Final report of European project number IST-1999-12324, named **New European Schemes for Signatures, Integrity, and Encryption**

April 19, 2004— Version 0.15 (beta)

Springer-Verlag

Berlin Heidelberg New York
London Paris Tokyo
Hong Kong Barcelona
Budapest

The work described in this report has been supported by the Commission of the European Communities through the IST program under contract IST-1999-12324. The information in this document is provided as is, and no warranty is given or implied that the information is fit for any particular purpose. The user thereof uses the information at its sole risk and liability.

B. Preneel[1], A. Biryukov[1], C. De Cannière[1], S. B. Örs[1],
E. Oswald[1,8], B. Van Rompay[1],
L. Granboulan[2], E. Dottax[2], G. Martinet[2],
S. Murphy[3], A. Dent[3], R. Shipsey[3], C. Swart[3], J. White[3],
M. Dichtl[4], S. Pyka[4], M. Schafheutle[4], P. Serf[4],
E. Biham[5], E. Barkan[5], Y. Braziler[5], O. Dunkelman[5],
V. Furman[5], D. Kenigsberg[5], J. Stolin[5],
J-J. Quisquater[6], M. Ciet[6], F. Sica[6],
H. Raddum[7], L. Knudsen[7,9], M. Parker[7].

[1] Katholieke Universiteit Leuven, Dept. Elektrotechniek-ESAT/SCD-COSIC, Kasteelpark Arenberg 10, B-3001 Leuven-Heverlee, Belgium

[2] École Normale Supérieure, Département d'Informatique, 45 rue d'Ulm, Paris 75230 Cedex 05, France

[3] Royal Holloway, Information Security Group, Egham, Surrey TW20 0EX, UK

[4] Siemens AG, Otto-Hahn-Ring 6, 81739 München, Germany

[5] Technion, Computer Science Dept., Haifa 32000, Israel

[6] Université catholique de Louvain, UCL Crypto Group, Laboratoire de Microélectronique (DICE), Place du Levant 3, B-1348 Louvain-la-Neuve, Belgium

[7] Universitetet i Bergen, Dept. of Informatics, PO Box 7800 Thormoehlensgt. 55, Bergen 5020, Norway

[8] A-SIT, Technologiebeobachtung, Inffeldgasse 16a, A-8010 Graz, Austria

[9] Tech. U. of Denmark, Dept.of Mathematics, Building 303, DK-2800 Lyngby, Denmark

Table of Contents

Book I. An Introduction to the NESSIE Project

Book II. Evaluation

Book III. The NESSIE portfolio

Book IV. Selected research papers

Book V. References

Book I

An Introduction to the NESSIE Project

1. Introduction

Abstract. In February 2000 the NESSIE project has launched an open call for the next generation of cryptographic algorithms. These algorithms should offer a higher security and/or confidence level than existing ones, and should be better suited for the constraints of future hardware and software environments. The NESSIE project has received 39 algorithms, many of these from major players. In October 2001, the project completed the first phase of the evaluation and has selected 24 algorithms for the second phase. The selection of the portfolio of 17 recommended algorithms has been announced in February 2003. This article presents an introduction to the NESSIE project.

NESSIE (New European Schemes for Signature, Integrity, and Encryption) is a research project within the Information Societies Technology (IST) Programme of the European Commission. The participants of the project are:

- Katholieke Universiteit Leuven (Belgium), coordinator;
- Ecole Normale Supérieure (France);
- Royal Holloway, University of London (U.K.);
- Siemens Aktiengesellschaft (Germany);
- Technion - Israel Institute of Technology (Israel);
- Université Catholique de Louvain (Belgium); and
- Universitetet i Bergen (Norway).

NESSIE is a 3-year project, which started on January 1, 2000. This paper presents the state of the project, and it is organized as follows. Section 2 discusses the NESSIE call and its results. Section 3 discusses the tools which the project is developing to support the evaluation process. Sections 4 and 5 deal with the security and performance evaluation respectively, and Sect. 6 discusses the selection of algorithms for the 2nd phase. Section 7 raises some intellectual property issues. The NESSIE approach towards dissemination and standardization is presented in Section 8. Finally, conclusions are put forward in Section 9.

Detailed and up to date information on the NESSIE project is available at the project web site `http://cryptonessie.org/`.

[†] Bart Preneel
Katholieke Univ. Leuven, Dept. Electrical Engineering-ESAT,
Kasteelpark Arenberg 10, B-3001 Leuven-Heverlee, Belgium
`bart.preneel@esat.kuleuven.ac.be`

2. NESSIE Call

In the first year of the project, an open call for the submission of cryptographic algorithms, as well as for evaluation methodologies for these algorithms has been launched. The scope of this call has been defined together with the project industry board (PIB) (cf. Sect. 8), and it was published in February 2000. The deadline for submissions was September 29, 2000. In response to this call NESSIE received 40 submissions, all of which met the submission requirements.

2.1 Contents of the NESSIE Call

The NESSIE call includes a request for a broad set of algorithms providing date confidentiality, data authentication, and entity authentication. These algorithms include block ciphers, stream ciphers, hash functions, MAC algorithms, digital signature schemes, and public-key encryption and identification schemes (for definitions of these algorithms, see [441]). In addition, the NESSIE call asks for evaluation methodologies for these algorithms. While key establishment protocols are also very important, it was felt that they should be excluded from the call, as the scope of the call is already rather broad.

The scope of the NESSIE call is much wider than that of the AES call launched by NIST [630], which was restricted to 128-bit block ciphers. It is comparable to that of the RACE Project RIPE (Race Integrity Primitives Evaluation, 1988-1992) [116] (confidentiality algorithms were excluded from RIPE for political reasons) and that of the Japanese CRYPTREC project [631] (which also includes key establishment protocols and pseudo-random number generation). Another difference is that both AES and CRYPTREC intend to produce algorithms for government standards. The results of NESSIE will not be adopted by any government or by the European commission. However, the intention is that relevant standardization bodies will adopt these results. As an example, algorithms for digital signature and hash functions may be included in the EESSI standardization documents which specify algorithms recommended for the European Electronic Signature Directive.

The call also specifies the main selection criteria which will be used to evaluate the proposals. These criteria are long-term security, market requirements, efficiency, and flexibility. Primitives can be targeted towards a specific environment (such as 8-bit smart cards or high-end 64-bit processors), but it is clearly an advantage to offer a wide flexibility of use. Security is put forward as the most important criterion, as security of a cryptographic algorithm is essential to achieve confidence and to build consensus.

For the *security requirements* of symmetric algorithms, two main security levels are specified, named *normal* and *high*. The minimal requirements for a symmetric algorithm to attain either the normal or high security level depend on the key length, internal memory, or output length of the algorithm. For block ciphers a third security level, *normal-legacy*, is specified, with a block size of 64 bits compared to 128 bits for the normal and high security level. The motivation for this request are applications such as UMTS/3GPP, which intend to use 64-bit block ciphers for the next 10-15 years. For the asymmetric algorithms, a varying security level is accepted, with as minimum about 2^{80} 3-DES encryptions.

If selected by NESSIE, the algorithm should preferably be available royalty-free. If this is not possible, then access should be non-discriminatory. The submitter should state the position concerning intellectual property and should update it when necessary.

The submission requirements are much less stringent than for AES, particularly in terms of the requirement for software implementations (only ‘portable C’ is mandatory).

2.2 Response to the NESSIE Call

The cryptographic community has responded very enthusiastically to the call. Thirty nine algorithms have been received, as well as one proposal for a testing methodology. After an interaction process, which took about one month, all submissions comply with the requirements of the call. There are 26 symmetric algorithms:

- seventeen block ciphers, which is probably not a surprise given the increased attention to block cipher design and evaluation as a consequence of the AES competition organized by NIST. They are divided as follows:
  - six 64-bit block ciphers: CS-Cipher, Hierocrypt-L1, IDEA, Khazad, MISTY1, and Nimbus;
  - seven 128-bit block ciphers: Anubis, Camellia, Grand Cru, Hierocrypt-3, Noekeon, Q, and SC2000 (none of these seven come from the AES process);
  - one 160-bit block cipher: Shacal; and
  - three block ciphers with a variable block length: NUSH (64, 128, and 256 bits), RC6 (at least 128 bits), and SAFER++ (64 and 128 bits).
- six synchronous stream ciphers: BMGL, Leviathan, LILI-128, SNOW, SOBER-t16, and SOBER-t32.
- two MAC algorithms: Two-Track-MAC and UMAC; and
- one collision-resistant hash function: Whirlpool.

Thirteen asymmetric algorithms have been submitted:

- five asymmetric encryption schemes: ACE Encrypt, ECIES, EPOC, PSEC, and RSA-OAEP (both EPOC and PSEC have three variants);

– seven digital signature algorithms: ACE Sign, ECDSA, ESIGN, FLASH, QUARTZ, RSA-PSS, and SFLASH; and
– one identification scheme: GPS.

Approximately[1] seventeen submissions originated within Europe (6 from France, 4 from Belgium, 3 from Switzerland, 2 from Sweden), nine in North America (7 USA, 2 from Canada), nine in Asia (8 from Japan), three in Australia and three in South America (Brazil). The majority of submissions originated within industry (27); seven came from academia, and six are the result of a joint effort between industry and academia. Note however that the submitter of the algorithm may not be the inventor, hence the share of academic research is probably underestimated by these numbers.

On November 13–14, 2000 the first NESSIE workshop was organized in Leuven (Belgium), where most submissions were presented. All submissions are available on the NESSIE web site [633].

[1] Fractional numbers have been used to take into account algorithms with submitters over several continents/countries – the totals here are approximations by integers, hence they do not add up to 40.

3. Tools

It is clear that modern computers and sophisticated software tools cannot replace human cryptanalysis. Nevertheless, software tools can play an important role in modern cryptanalysis. In most cases, the attacks found by the cryptanalyst require a large number of computational steps, hence the actual computation of the attack is performed on a computer. However, software and software tools can also be essential to find a successful way to attack a symmetric cryptographic algorithm; examples include differential and linear cryptanalysis, dependence tests, and statistical tests.

Within NESSIE, we distinguish two classes of tools. The general tools are not specific for the algorithms to be analyzed. Special tools, which are specific for the analysis of one algorithm, are implemented when, in the course of the cryptanalysis of an algorithm, the need for such a tool turns up.

For the evaluation of the symmetric submissions, a comprehensive set of general tools is available within the project. These tools are in part based on an improved version of the tools developed by the RIPE (RACE Integrity Primitives Evaluation) project [116]. These test include: the frequency test, the collision test, the overlapping m-tuple test, the gap test, the constant runs test, the coupon collector's test, Maurer's universal test [429], the poker test, the spectral test, the correlation test, the rank test, the linear, non-linear, and dyadic complexity test, the Ziv-Lempel complexity test, the dependence test, the percolation test, the linear equation, linear approximation and correlation immunity test, the linear factors test, and a cycle detection tool.

The NESSIE project is also developing a new generic tool to analyze block ciphers with differential [78] and linear cryptanalysis [422]. This tool is based on a general description language for block ciphers.

In September 2000, the US NIST published a suite of statistical tests for the evaluation of sequences of random or pseudo-random bits; this document has been revised in December 2000 [469]. A careful comparison has been made between the RIPE and NIST test suites.

The software for these tools will not be made available outside the project, but all the results obtained using these tools will be made public in full detail.

4. Security Evaluation

We first describe the internal process within NESSIE used to assess submissions. Initially each submission was assigned to a NESSIE partner, who performed basic checks on the submission, such as compliance with the call, working software, obvious weaknesses etc. The aim of this initial check was mainly to ensure that submissions were specified in a consistent and cogent form in time for the November 2000 workshop. It is vital for proper security assessments that the algorithms are fully and unambiguously described. This process required interaction with some submitters to ensure that the submissions were in the required form.

The next internal stage (November 2000) was to assign each submission to a pair of NESSIE partners for an initial detailed evaluation. Each submission has then been subject to two independent initial assessments. After the two initial assessments of a submission have taken place, the two NESSIE partners have produced a joint summary of their assessments concerning that submission. Based on this initial evaluation, algorithms were dismissed or subjected to further dedicated analysis.

Next, an open workshop was organized in Egham (UK) on September 12-13, 2001 to discuss the security and performance analysis of the submissions. The presenters include both researchers from the NESSIE project, but also submitters, members from the NESSIE PIB, and members from the cryptographic community at large.

Following this workshop, a comprehensive security evaluation report has been published [478]. The document gives an overview of generic attacks on the different type of algorithms. Moreover, for each symmetric algorithm it presents a short description, the security claims by the designers, and the reported weaknesses and attacks. The part on asymmetric algorithms contains a discussion of security assumptions, security models, and of the methodology to evaluate the security. For each algorithm, a short description is followed by a discussion of the provable security (which security properties are proved under which assumptions) and of the concrete security reduction.

5. Performance Evaluation

Performance evaluation is an essential part in the assessment of a cryptographic algorithm: efficiency is a very important criterion in deciding for the adoption of an algorithm.

The candidates will be used on several platforms (PCs, smart cards, dedicated hardware) and for various applications. Some applications have tight timing constraints (e.g., payment applications, cellular phones); for other applications a high throughput is essential (e.g., high speed networking, hard disk encryption).

First a framework has been defined to compare the performance of algorithms on a fair and equal basis. It will be used for all evaluations of submitted candidates. First of all a theoretical approach has been established. Each algorithm is dissected into three parts: setup (independent of key and data), precomputations (independent of data, e.g., key schedule) and the algorithm itself (that must be repeated for every use). Next a set of four test platforms has been defined on which each candidate may be tested. These platforms are smart cards, 32-bit PCs, 64-bit machines, and Field Programmable Gate Arrays (FPGAs).

Then rules have been defined which specify how performance should be measured on these platforms. The implementation parameters depend on the platform, but may include RAM, speed, code size, chip area, and power consumption. On smart cards, only the following parameters will be taken into account, in decreasing order of importance: RAM usage, speed, code size. On PCs, RAM has very little impact, and speed is the main concern. On FPGAs, throughput, latency, chip area and power consumption will be considered. Unfortunately, the limited resources of the project will not allow for the evaluation of dedicated hardware implementations (ASICs), but it may well be that teams outside the project can offer assistance for certain algorithms.

The project will also consider the resistance of implementations to physical attacks such as timing attacks [374], fault analysis [79, 104], and power analysis [375]. For non constant-time algorithms (data or key dependence, asymmetry between encryption and decryption) the data or key dependence will be analyzed; other elements that will be taken into account include the difference between encryption and decryption, and between signature and verification operation. For symmetric algorithms, the key agility will also be considered.

This approach will result in the definition of a platform dependent test and in several platform dependent rekeying scenarios. Low-cost smart cards will only be used for block ciphers, MACs, hash functions, stream ciphers, pseudo-random number generation, and identification schemes.

In order to present performance information in a consistent way within the NESSIE project, a performance ‘template’ has been developed. The goal of this template is to collect intrinsic information related to the performance of the submitted candidates. A first part describes parameters such as word size, memory requirement, key size and code size. Next the basic operations are analyzed, such as shift/rotations, table look-ups, permutations, multiplications, additions, modular reduction, exponentiation, inversion,.... Then the nature and speed of precomputations (setup, key schedule, etc.) are described. Elements such as the dependence on the keys and on the inputs determine whether the code is constant-time or not. Alternative representations of the algorithms are explored when feasible.

The result of the preliminary performance evaluation are presented in [477]. This document contains an overview of the performance claimed by the designers, a theoretical evaluation, and performance measurements of optimized C-code on a PC and a workstation. However, due to limited resources and the large number of algorithms, it was not possible to guarantee full optimization for all algorithms. Nevertheless, it was felt that these results provide sufficient information to make a selection of algorithms for the 2nd phase of the project.

6. Selection for the 2nd Phase

On September 24, 2001, the NESSIE project has announced the selection of candidates for the 2nd phase of the project. Central to the decision process has been the project goal, that is, to come up with a portfolio of strong cryptographic algorithms. Moreover, there was also a consensus that every algorithm in this portfolio should have a unique competitive advantage that is relevant to an application.

It is thus clear that an algorithm could not be selected if it failed to meet the security level required in the call. A second element could be that the algorithm failed to meet a security claim made by the designer. A third reason to eliminate an algorithm could be that a similar algorithm exists with better security (for comparable performance) or with significantly better performance (for comparable security). In retrospect, very few algorithms were eliminated because of performance reasons. It should also be noted that the selection was more competitive in the area of block ciphers, where many strong contenders were considered. The motivation for the decisions is given in [476].

Designers of submitted algorithms were allowed to make small alterations to their algorithms; the main criterion to accept these alterations is that they should improve the algorithm and not substantially invalidate the existing security analysis. More information on the alterations can be found on the NESSIE webpages [633].

The selected algorithms are listed below; altered algorithms are indicated with a *. Block ciphers:

- IDEA: MediaCrypt AG, Switzerland;
- Khazad*: Scopus Tecnologia S.A., Brazil and K.U.Leuven, Belgium;
- MISTY1: Mitsubishi Electric Corp., Japan;
- SAFER++64, SAFE++128: Cylink Corp., USA, ETH Zurich, Switzerland, National Academy of Sciences, Armenia;
- Camellia: Nippon Telegraph and Telephone Corp., Japan and Mitsubishi Electric, Japan;
- RC6: RSA Laboratories Europe, Sweden and RSA Laboratories, USA;
- Shacal: Gemplus, France.

Here IDEA, Khazad, MISTY1 and SAFER++64 are 64-bit block ciphers. Camellia, SAFER++128 and RC6 are 128-bit block ciphers, which will be compared to AES/Rijndael [181,470]. Shacal is a 160-bit block cipher based on SHA-1 [465]. A 256-bit version of Shacal based on SHA-256 [472] has also been introduced in the

second phase; this algorithm will be compared to an RC-6 and a Rijndael [181] variant with a block length of 256 bits (note that this variant is not included in the AES standard). The motivation for this choice is that certain applications (such as the stream cipher BMGL and certain hash functions) can benefit from a secure 256-bit block cipher.
Synchronous stream ciphers:

- SOBER-t16, SOBER-t32: Qualcomm International, Australia;
- SNOW*: Lund Univ., Sweden;
- BMGL*: Royal Institute of Technology, Stockholm and Ericsson Research, Sweden.

MAC algorithms and hash functions:

- Two-Track-MAC: K.U.Leuven, Belgium and debis AG, Germany;
- UMAC: Intel Corp., USA, Univ. of Nevada at Reno, USA, IBM Research Laboratory, USA, Technion, Israel, and Univ. of California at Davis, USA;
- Whirlpool*: Scopus Tecnologia S.A., Brazil and K.U.Leuven, Belgium.

The hash function Whirlpool will be compared to the new FIPS proposals SHA-256, SHA-384 and SHA-512 [472].
Public-key encryption algorithms:

- ACE-KEM*: IBM Zurich Research Laboratory, Switzerland (derived from ACE Encrypt);
- EPOC-2*: Nippon Telegraph and Telephone Corp., Japan;
- PSEC-KEM*: Nippon Telegraph and Telephone Corp., Japan (derived from PSEC-2);
- ECIES*: Certicom Corp., USA and Certicom Corp., Canada
- RSA-OAEP*: RSA Laboratories Europe, Sweden and RSA Laboratories, USA.

Digital signature algorithms:

- ECDSA: Certicom Corp., USA and Certicom Corp., Canada;
- ESIGN*: Nippon Telegraph and Telephone Corp., Japan;
- RSA-PSS: RSA Laboratories Europe, Sweden and RSA Laboratories, USA;
- SFLASH*: BULL CP8, France;
- QUARTZ*: BULL CP8, France.

Identification scheme:

- GPS*: Ecole Normale Supérieure, Paris, BULL CP8, France Télécom and La Poste, France.

Many of the asymmetric algorithms have been updated at the beginning of phase 2. For the asymmetric encryption schemes, these changes were driven in part by the recent cryptanalytic developments, which occurred after the NESSIE submission deadline [243, 414, 583]. A second reason for these changes is the progress of standardization within ISO/IEC JTC1/SC27 [584]. The standards seem to evolve towards defining a hybrid encryption scheme, consisting of two components: a KEM (Key Encapsulation Mechanism), where the asymmetric encryption is used to encrypt a symmetric key, and a DEM (Data Encapsulation Mechanism), which protects both secrecy and integrity of the bulk data with symmetric techniques (a “digital envelope”). This approach is slightly more complicated for encryption of a short plaintext, but it offers a more general solution with clear advantages. Three of the five NESSIE algorithms (ACE Encrypt, ECIES and PSEC-2) have been modified to take into account this development. At the same time some other improvements have been introduced; as an example, ACE-KEM can be based on any abstract group, which was not the case for the original submission ACE Encrypt. Other submitters decided not to alter their submissions at this stage. For further details, the reader is referred to the extensive ISO/IEC draft document authored by V. Shoup [584]. The NESSIE project will closely monitor these developments. Depending on the progress, variants such as RSA-KEM defined in [584] may be studied by the NESSIE project.

For the digital signature schemes, three out of five schemes (ESIGN, QUARTZ and SFLASH) have been altered. In this case, there are particular reasons for each algorithm (correction for the security proof to apply, improve performance, or preclude a new attack). The other two have not been modified. It should also be noted that PSS-R, which offers very small storage overhead for the signature, has not been submitted to NESSIE.

7. Intellectual Property

An important element in the evaluation is the intellectual property status. While it would be ideal for users of the NESSIE results that all algorithms recommended by NESSIE were in the public domain, it is clear that this is for the time being not realistic. The users in the NESSIE PIB have clearly stated that they prefer to see royalty-free algorithms, preferably combined with open source implementations. However, providers of intellectual property typically have different views.

One observation is that in the past, there has always been a very large difference between symmetric and asymmetric cryptographic algorithms. Therefore it is not so surprising that NIST was able to require that the designers of the block cipher selected for the AES would give away all their rights, if their algorithm was selected; it is clear that this is not a realistic expectation for the NESSIE project.

In this section we will attempt to summarize the intellectual property statements of the submissions retained for the 2nd phase. Note however that this interpretation is only indicative; for the final answer the reader is referred to the intellectual property statement on the NESSIE web page [633], and to the submitters themselves.

Twelve out of 24 algorithms are in the public domain, or the submitters indicate that a royalty-free license will be given. These are the block ciphers Khazad, Misty1, Shacal, Safer++, the stream ciphers BMGL, SNOW, Sober-t16 and Sober-t32, the MAC algorithms Two-Track-MAC and UMAC, the hash function Whirlpool, and the public-key algorithms RSA-OAEP[2] (public-key encryption) and RSA-PSS[1] (digital signature scheme).

Royalty-free licenses will be given for the block cipher Camellia, for the public-key encryption algorithms EPOC-2 and PSEC-KEM, and for the digital signature scheme ESIGN, provided that other companies with IPR to the NESSIE portfolio reciprocate.

The block cipher IDEA is free for non-commercial use only; for commercial applications a license is required.

Licenses under reasonable and non-discriminatory terms will be given for ACE-KEM (the detailed license conditions are rather complex). Additions to the 'reasonable and non-discriminatory' terms are required for the public-key algorithms ECDSA and ECIES; it is required that the license holder reciprocates some of his rights.

[1] This statement does not hold for the variants of RSA with more than two primes.

For the digital signature schemes SFLASH and QUARTZ the licensing conditions are expected to be non-discriminatory, but no decision has been made yet. A similar statement holds for the identification scheme GPS, but in this case certain applications in France may be excluded from the license.

Finally, the submitters of RC6 are willing to negotiate licenses on reasonable terms and conditions.

It is clear that intellectual property is always a complex issue, and it will not be possible to resolve this completely within the framework of NESSIE. However, IPR issues may play an important role in the final selection process.

8. Dissemination and Standardization

8.1 An Open Evaluation Process

The NESSIE project intends to be an open project, which implies that the members of the public are invited to contribute to the evaluation process. In order to facilitate this process, all submissions are available on the NESSIE website, and comments are distributed through this website. In addition, three open workshops are organized during the project: the first two workshops have taken place in November 2000 and September 2001; the third one has been scheduled for November 2002.

8.2 The Project Industry Board

The Project Industry Board (PIB) was established to ensure that the project addresses real needs and requirements of industry dealing with the provision and use of cryptographic techniques and cryptographic products. The goals for the Board may be summarized as follows:

– contribute to dissemination: outwards through a member's contacts with industry and users, and also through passing NESSIE information into the member's own organization influencing products and directions;
– collaboration with the Project in formulation of the call and its goals and requirements;
– contribution to consensus building through influence and contacts in the industry and marketplace;
– identification of industry requirements from market needs and corporate strategies;
– guidance and judgment on the acceptability and relevance of submissions and evaluation results;
– support in standardization of NESSIE results;
– contribution to Project workshops;
– practical contributions to analysis and evaluation of submissions;
– identification of gaps in the scope of the submissions;
– ongoing guidance during the evaluation of the processes and validity of results.

Two meetings are held per year, but the PIB may request additional meetings to address specific issues or concerns that may arise. Membership was originally by invitation, but subsequently a number of additional companies have requested and obtained membership. Currently the PIB consists of about twenty leading companies which are users or suppliers of cryptology.

8.3 Standardization

Together with the NESSIE PIB, the project will establish a standardization strategy. It is not our intention to establish a new standardization body or mechanism, but to channel the NESSIE results to the appropriate standardization bodies, such as, ISO/IEC, IETF, IEEE and EESSI. We believe that the NESSIE approach of open evaluation is complementary to the approach taken by standardization bodies. Indeed, these bodies typically do not have the resources to perform any substantial security evaluation, which may be one of the reasons why standardization in security progresses often more slowly than anticipated.

The NESSIE project will also take into account existing and emerging standards, even if these have not been formally submitted to the NESSIE project. Two recent examples in this context come from the standardization efforts run by NIST: AES/Rijndael [181,470] will be used as a benchmark for the other 128-bit block ciphers, and the NESSIE project will study the security and performance of the new SHA variants with results between 256 and 512 bits [472].

9. Conclusion

We believe that after two years, the NESSIE project has made important steps towards achieving its goals. This can be deduced from the high quality submissions received from key players in the community, and by the active participation to the workshops.

The first two years of the NESSIE project have also shown that initiatives of this type (such as AES, RIPE, CRYPTREC) can bring a clear benefit to the cryptographic research community and to the users and implementors of cryptographic algorithms. By asking cryptographers to design concrete and fully specified schemes, they are forced to make choices, to think about real life optimizations, and to consider all the practical implications of their research. While leaving many options and variants in a construction may be very desirable in a research paper, it is often confusing for a practitioner. Implementors and users can clearly benefit from the availability of a set of well defined algorithms, that are described in a standardized way.

The developments in the last years have also shown that this approach can result in a better understanding of the security of cryptographic algorithms. We have also learned that concrete security proofs are an essential tool to build confidence, particularly for public key cryptography (where constructions can be reduced to mathematical problems believed to be hard) and for constructions that reduce the security of a scheme to other cryptographic algorithms. At the same time, we have learned that it is essential to study proofs for their correctness and to evaluate the efficiency of such reductions.

Acknowledgments

I would like to thank all the members of and the contributors to the NESSIE project. The work described in this paper has been supported by the Commission of the European Communities through the IST Programme under Contract IST–1999–12324.
Bart Preneel, NESSIE coordinator

Book II

Evaluation

Part A

Submitted primitives

1. Call for Cryptographic Primitives

Version 2.2, 8th March 2000

Introduction

NESSIE (New European Schemes for Signature, Integrity, and Encryption) is a project within the Information Societies Technology (IST) Programme of the European Commission. The participants of the project are

Participant name	Country
Katholieke Universiteit Leuven	Belgium
École Normale Supérieure	France
Fondazione Ugo Bordoni	Italy
Royal Holloway, University of London	U.K.
Siemens Aktiengesellschaft	Germany
Technion - Israel Institute of Technology	Israel
Université Catholique de Louvain	Belgium
Universitetet i Bergen	Norway

NESSIE is a 3-year project, which started on 1st January 2000. Further information about NESSIE is available at `http://cryptonessie.org`.

The main objective of the project is to put forward a portfolio of strong cryptographic primitives for a number of different platforms. These primitives will be obtained after an open call and evaluated using a transparent and open process. They should be the building blocks of the future standard protocols for the information society.

The deadline for the submission of primitives will be 29th September 2000. A workshop will be organised for submitters to present their primitives on 13-14 November 2000 in Leuven, Belgium.

Background

In the information society, cryptology has become a key enabling technology to provide secure electronic commerce and electronic business, secure communica-

This call was widely dissemniated and published on the NESSIE web site `http://www.cosic.esat.kuleuven.ac.be/nessie/call/`.

tions, secure payments, and the protection of the privacy of the citizen. Cryptology is a field that evolves quickly, and society needs robust primitives that provide long term security (15 to 20 years or more), rather than ad hoc solutions that need to be frequently replaced. With the current state of the art in cryptology, it is not possible to have provably secure solutions, although there is a trend to prove more and more security properties of primitives. However, for use in real applications, sufficient confidence in a primitive can only be achieved when primitives have been subjected to an open and independent evaluation for a sufficient amount of time.

The procedure of an open call followed by an evaluation process has been previously used in the selection process for the DES, the RIPE project, and the AES. The scope of this call for primitives is wider than the NIST call for AES. The information society needs other cryptographic primitives than just block ciphers. Thus the NESSIE call seeks cryptographic primitives in many areas, such as:

- Stream ciphers: for applications with high throughput requirements or tight performance constraints etc.
- MACs: for high-speed authentication of packets etc.
- Families of Pseudo-random functions: for key derivation, entity authentication and encryption etc.
- Digital signatures and hash functions: for electronic commerce, business and payment etc.
- Asymmetric encryption schemes.
- Asymmetric identification schemes.

Furthermore, there is a wide range of environments in which cryptographic primitives are used. Thus the NESSIE project will consider primitives designed for use in specific environments (though flexibility is clearly desirable). The NESSIE call also asks for testing methodologies of these primitives (such as statistical tests).

The results of this call will then be subjected to a thorough and open evaluation process. In addition to the responses to the call, the project will also consider a selection from existing standards containing such primitives. The main selection criteria will be long-term security, market requirements, efficiency (performance), and flexibility.

It is also a goal of the project to disseminate widely the results of the project, and to build a consensus based on these results. In order to achieve this, an Industry Group has been established. The Industry Group consists of about twenty leading European companies in this area and will be consulted on a regular basis throughout the project. It is expected that the Industry Group will provide input concerning the nature of the final call (requirements and definitions for primitives), the relevance of the selection criteria, and the standardisation strategy. An important part of the dissemination will be the introduction of these primitives into standardisation bodies (ISO, ISO/IEC, CEN, IEEE, IETF), based in part on the consensus achieved within the project. It is anticipated that the results of the project will also be published in scientific publications.

General Requirements

This section discusses the general selection criteria, the type of primitives required, and the security requirements for each primitive.

Selection Criteria

The main selection criteria will be long-term security, market requirements, efficiency (performance) and flexibility.

- Security is the most important criterion, because security of a cryptographic primitive is essential to achieve confidence and to build consensus. It is anticipated that this evaluation process will be influenced by developments outside the project (such as new attacks or analysis techniques).
- A second criterion relates to market requirements. Market requirements are related to the need for a primitive, its usability, and the possibility for worldwide use.
- A third criterion is the performance of the primitive in the specified environment. For software, the range of environments considered include 8-bit processors (as found in inexpensive smart cards), 32-bit processors (e.g., the Pentium family) to the modern 64-bit processors. For hardware, both FPGAs and ASICs will be considered.
- A fourth criterion is the flexibility of the primitive. It is clearly desirable for a primitive to be suitable for use in a wide-range of environments.

Type of Primitives

The NESSIE project is seeking submissions of strong cryptographic primitives in the categories given below. The NESSIE project is particularly interested in receiving submissions in categories that have not received much standardisation effort.

1. Block ciphers
2. Synchronous stream ciphers
3. Self-synchronising stream ciphers
4. Message Authentication Codes (MACs)
5. Collision-resistant hash functions
6. One-way hash functions
7. Families of pseudo-random functions
8. Asymmetric encryption schemes
9. Asymmetric digital signature schemes
10. Asymmetric identification schemes

Definitions are broadly as given in the *Handbook of Applied Cryptography* (ISBN: 0-8493-8523-7).

Security Requirements for Each Primitive

Symmetric Primitives (Primitives 1-7)

There are two main security levels for symmetric primitives. These are named *normal* and *high*. For block ciphers, *normal-legacy* is also provided. The minimal requirements for a symmetric primitive to attain a security level are given below.

1. **Block ciphers.**
   a) *High.* Key length of at least 256 bits. Block length at least 128 bits
   b) *Normal.* Key length of at least 128 bits. Block length at least 128 bits.
   c) *Normal-Legacy.* Key length of at least 128 bits. Block length 64 bits
2. **Synchronous stream ciphers.**
   a) *High.* Key length of at least 256 bits. Internal memory of at least 256 bits.
   b) *Normal.* Key length of at least 128 bits. Internal memory of at least 128 bits.
3. **Self-synchronising stream ciphers.**
   a) *High.* Key length of at least 256 bits. Internal memory of at least 256 bits.
   b) *Normal.* Key length of at least 128 bits. Internal memory of at least 128 bits.
4. **Message Authentication Codes (MACs).** The primitive should support all output lengths (in multiples of 32 bits) up to the key length (inclusive).
   a) *High.* Key length of at least 256 bits.
   b) *Normal.* Key length of at least 128 bits.
5. **Collision-Resistant Hash functions.**
   a) *High.* Output length of at least 512 bits.
   b) *Normal.* Output length of at least 256 bits.
6. **One-Way Hash functions.** These hash functions shall be preimage resistant and second preimage resistant.
   a) *High.* Output length of at least 256 bits.
   b) *Normal.* Output length of at least 128 bits.
7. **Families of Pseudo-random functions.** Fixed block length of at least 128 bits.
   a) *High.* Key length of at least 256 bits.
   b) *Normal.* Key length of at least 128 bits.

Asymmetric Primitives (Primitives 8-10)

The security parameters should be chosen such that the most efficient attack on the primitive requires a computational effort of the order of 2^{80} 3-DES encryptions. Furthermore, a table giving the security levels in terms of the security parameters should be provided.

8. **Asymmetric encryption schemes (deterministic or randomised).** The minimal computational effort for an attack should be of the order of 2^{80} 3-DES encryptions.

9. **Asymmetric digital signature schemes.** The minimal computational effort for an attack should be of the order of 2^{80} 3-DES encryptions.
10. **Asymmetric identification schemes.** The minimal computational effort for an attack should be of the order of 2^{80} 3-DES encryptions. The probability of impersonation should be smaller than 2^{-32}.

Evaluation of Proposals

Detailed Evaluation Criteria

Security Criteria

- An attack should be at least as difficult as the generic attacks against the type of primitive (exhaustive search, birthday attack etc.).
- Primitives will be evaluated against the security claims of the submitter. An attack requiring lower computing resources than claimed would usually disqualify the submission.
- Primitives will be evaluated within the stated environment. Thus, consideration of vulnerability to side channel attacks (e.g., timing attacks, power analysis) may be appropriate.

Implementation Criteria

- Software and hardware efficiency will be compared with similar submissions and existing primitives.
- Execution code and memory sizes will be assessed according to their relevance in different contexts. Special attention will be paid to smart cards.
- Submitted primitives will be assessed against claimed performance, though it is clearly preferable for primitives to offer wide flexibility of use.

Other Criteria

- Simplicity and clarity of design are important considerations. Variable parameter sizes are less important.

Licensing Requirements

- Submitted primitives should, if selected by NESSIE, be available royalty-free. If this is not possible, then access is non-discriminatory.
- The submitter should state the position concerning intellectual property. This statement should be updated when necessary.

The Evaluation Process

The NESSIE project reserves the right to reject submitted primitives that are not clearly specified and easily comprehensible or that fail to meet the NESSIE requirements in some way.

The NESSIE evaluation process is an open process. Thus as part of the evaluation process, the NESSIE project welcomes comments about both submitted primitives and the evaluation process, including evaluation methodologies. To facilitate this process, three NESSIE workshops will be organised; the first will be on 13-14 November 2000, shortly after the deadline for submission of primitives. As part of the evaluation process, the NESSIE partners will give security and performance assessments of the submitted primitives. At the end of the evaluation process, the NESSIE project may recommend certain submissions for standardisation. The NESSIE project is to have two phases. A timetable for the NESSIE project is given below.

2000	January	Beginning of first phase of NESSIE
2000	January	Creation of Industry Board
2000	March	Call for Cryptographic Primitives
2000	September	Submission deadline
2000	November	First NESSIE workshop
2001	June	Preliminary assessment of submissions
2001	June	End of first phase of NESSIE
2001	July	Beginning of second phase of NESSIE
2001	September	Second NESSIE workshop
2002	February	Preliminary selection of submissions
2002	February	Standardisation Plan
2002	October	Third NESSIE workshop
2002	December	Final selection of submissions
2002	December	Final report of NESSIE project
2002	December	End of second phase of NESSIE

Formal Submission Requirements

The following are to be provided with any submission:

A. Cover sheet with the following information

1. Name of submitted algorithm
2. Type of submitted algorithm, proposed security level, and proposed environment.
3. Principal submitter's name, telephone, fax, organization, postal address, e-mail address
4. Name(s) of auxiliary submitter(s)
5. Name of algorithm inventor(s)/developer(s)
6. Name of owner, if any, of the algorithm (normally expected to be the same as the submitter)
7. Signature of submitter
8. (optional) Backup point of contact (telephone, fax, postal address, e-mail)

B. Primitive specification and supporting documentation

1. A complete and unambiguous description of the primitive in the most suitable forms, such as a mathematical description, a textual description with diagrams, or pseudo-code. The specification of a primitive using code is not

permitted. Input and output should be in the form of binary strings. For asymmetric algorithms, a method for key generation and parameter selection needs to be specified.

2. A statement that there are no hidden weaknesses inserted by the designers.
3. A statement of the claimed security properties and expected security level, together with an analysis of the primitive with respect to standard cryptanalytic attacks. Weak keys should also be considered.
4. A statement giving the strengths and advantages of the primitive.
5. A design rationale explaining design choices.
6. A statement of the estimated computational efficiency in software. Estimates are required for different sub-operations like key setup, primitive setup, and encryption/decryption (as far as applicable). The efficiency should be estimated both in cycles per byte and cycles per block, indicating the processor type and memory. If performance varies with the size of the inputs, then values for some typical sizes should be provided. Optionally the designers may provide estimates for performance in hardware (area, speed, gate count, a description in VHDL).
7. A description of the basic techniques for implementers to avoid implementation weaknesses.

C. Implementations and test values

1. A reference implementation, in portable C. For symmetric primitives, NESSIE has specified an API, published on the NESSIE web site. For asymmetric primitives, it is allowed to use a 'standard' library for mathematical functions, such as multi-precision arithmetic (e.g., Lydia). See here for more information.
2. A sufficient number of test vectors. The NESSIE project will supply software to generate test vectors for symmetric primitives.
3. Optionally, an optimized implementation for some architectures, a JAVA implementation, an assembly language implementation.

D. Intellectual property statement

– A statement that gives the position concerning intellectual property position and the royalty policy for the primitive (if selected). This statement should include an undertaking to update the NESSIE project when necessary.

Requirements:

– Items A, B, and D shall be supplied in paper form and in electronic form (Adobe PDF or PostScript on one or more diskettes).
– Item C shall be supplied in electronic form only (diskette).
– Item A, B, C and D shall be supplied on separate 3,5" 1.44 MB floppy diskettes, formatted for use on an IBM-compatible PC. Every diskette shall be labeled with the submitter's name, the name of the primitive, the number, and the date. Every diskette shall contain an ASCII file labeled "README", that lists all files included on the diskette and provides a brief description of the content of each file.
– All submissions must be in English.
– Classified and/or proprietary submissions shall not be considered.

- Paper submissions and diskettes shall be sent to the following address:

  Prof. Bart Preneel
  NESSIE Project Coordinator
  Katholieke Universiteit Leuven
  Dept. Electrical Engineering-ESAT/COSIC
  Kard. Mercierlaan 94,
  B-3001 Heverlee
  BELGIUM

  They should arrive on or before 29th September 2000.
- Optionally, an electronic version may be sent to `submissions@cryptonessie.org`.

An acknowledgment will be sent by email and by regular mail within 5 working days.

General questions for clarification of the formal submission requirements and of the evaluation criteria can be sent to `info@cryptonessie.org`. Answers to questions that are relevant to other submissions will be made available at `http://www.cryptonessie.org/`. The NESSIE project will endeavour to answer all relevant questions in a timely manner.

Call for Evaluation Criteria

In addition to the criteria given above, the NESSIE project also invites suggestions for Evaluation Criteria, such as testing methodologies (eg. statistical testing).

Further Information

Email: `info@cryptonessie.org`.
Website: `http://www.cryptonessie.org/`.

2. The submissions

Block ciphers

- **Anubis.** [37]
  Submitted by Paulo S.L.M. Barreto and Vincent Rijmen.
- **Camellia.** [24]
  Submitted by Kazumaro Aoki, Tetsuya Ichikawa, Masayuki Kanda, Mitsuru Matsui, Shiho Moriai, Junko Nakajima and Toshio Tokita, on behalf of NTT Corp.
- **CS-cipher.** [234]
  Submitted by Pierre-Alain Fouque, Jacques Stern and Serge Vaudenay, on behalf of CS Communication & Systèmes.
- **Grand Cru.** [112]
  Submitted by Johan Borst.
- **Hierocrypt.** [493] – Hierocrypt-L1 and Hierocrypt-3.
  Submitted by Kenji Ohkuma, Fumihiko Sano, Hirofumi Muratani, Masahiko Motoyama and Shinichi Kawamura, on behalf of Toshiba Corp.
- **IDEA.** [387]
  Submitted by Richard Straub, on behalf of MediaCrypt AG.
- **Khazad.** [39]
  Submitted by Paulo S.L.M. Barreto and Vincent Rijmen.
- **MISTY1.** [426]
  Submitted by Eisaku Takeda, on behalf of Mitsubishi.
- **Nimbus.** [410]
  Submitted by Alexis W. Machado.
- **Noekeon.** [180]
  Submitted by Joan Daemen, Michael Peeters, Gilles van Assche and Vincent Rijmen.
- **NUSH.** [391]
  Submitted by Anatoly N. Lebedev and Alexey A. Volchkov, on behalf of LAN Crypto, Int.
- **Q.** [434]
  Submitted by Leslie McBride, on behalf of Mack One Software.
- **RC6.** [324]
  Submitted by Jakob Jonsson and Burton S. Kaliski, Jr, on behalf of RSA.

- **SAFER++.** [420]
  Submitted by James L. Massey, Gurgen Khachatrian and Melsik K. Kuregian, on behalf of Cylink Corp.
- **SC2000.** [569]
  Submitted by Takeshi Shimoyama, Hitoshi Yanami, Kazuhiro Yokoyama, Masahiko Takenaka, Kouichi Itoh, Jun Yajima, Naoya Torii and Hidema Tanaka, on behalf of Fujitsu Laboratories LTD.
- **SHACAL.** [281] – SHACAL-1 and SHACAL-2.
  Submitted by Helena Handschuh and David Naccache, on behalf of Gemplus.

Stream ciphers and pseudo-random number generators

- **BMGL.** [285]
  Submitted by Johan Håstad and Mats Nāslund.
- **SNOW.** [214]
  Submitted by Patrick Ekdahl and Thomas Johansson.
- **SOBER.** [289] – SOBER-t16 and SOBER-t32.
  Submitted by Philip Hawkes and Gregory G. Rose, on behalf of Qualcomm International.
- **LEVIATHAN.** [435]
  Submitted by David McGrew and Scott R. Fluhrer, on behalf of Cisco Systems, Inc.
- **LILI-128.** [187]
  Submitted by Ed Dawson, William Millan, Lyta Penna, Leone Simpson and Jovan Dj. Golic.

Hash functions

- **Whirlpool.** [38]
  Submitted by Paulo S.L.M. Barreto and Vincent Rijmen.

Message authentication codes

- **UMAC.** [379]
  Submitted by Ted Krovetz, John Black, Shai Halevi, Alejandro Hevia, Hugo Krawczyk and Phillip Rogaway.
- **Two-Track-MAC.** [609]
  Submitted by Bart Van Rompay and Bert Den Boer.

Asymmetric encryption schemes

- **ACE-KEM.** [563] – Upgrade of ACE-Encrypt.
  Submitted by Thomas Schweinberger and Victor Shoup, on behalf of IBM.

- **EPOC.** [239] – EPOC-1, EPOC-2 and EPOC-3.
  Submitted by Eiichiro Fujisaki, Tetsutaro Kobayashi, Hikaru Morita, Hiroaki Oguro, Tatsuaki Okamoto, Satomi Okazaki, David Pointcheval and Shigenori Uchiyama, on behalf of NTT Corp.
- **ECIES.** [320]
  Submitted by Don B. Johnson and Simon Blake-Wilson, on behalf of Certicom.
- **PSEC.** [238] – PSEC-1, PSEC-2, PSEC-3 and PSEC-KEM
  Submitted by Eiichiro Fujisaki, Tetsutaro Kobayashi, Hikaru Morita, Hiroaki Oguro, Tatsuaki Okamoto, Satomi Okazaki, David Pointcheval and Shigenori Uchiyama, on behalf of NTT Corp.
- **RSA-OAEP.** [325] – Revised to RSA-KEM [584]
  Submitted by Jakob Jonsson and Burton S. Kaliski, Jr, on behalf of RSA.

Digital signature schemes

- **ACE Sign.** [563]
  Submitted by Thomas Schweinberger and Victor Shoup, on behalf of IBM.
- **ECDSA.** [319]
  Submitted by Don B. Johnson and Simon Blake-Wilson, on behalf of Certicom.
- **ESIGN.** [244]
  Submitted by Eiichiro Fujisaki, Tetsutaro Kobayashi, Hikaru Morita, Hiroaki Oguro, Tatsuaki Okamoto and Satomi Okazaki on behalf of NTT Corp.
- **FLASH and SFLASH.** [514]
  Submitted by Jacques Patarin and others, on behalf of Schlumberger.
- **QUARTZ.** [161]
  Submitted by Jacques Patarin and others, on behalf of Schlumberger.
- **RSA-PSS.** [326]
  Submitted by Jakob Jonsson and Burton S. Kaliski, Jr, on behalf of RSA.

Digital identification schemes

- **GPS.** [525]
  Submitted by Guillaume Poupard and others, on behalf of France Telecom, La Poste and ENS.

Evaluation methodologies

- **General Next Bit Predictor.** [294]
  Submitted by Julio César Hernández, José María Sierra, J. C. Mex-Perera, Daniel Borrajo, Arturo Ribagorda and Pedro Isasi.

Part B

Security evaluation

Book II Part B of this final report was first published under the name ”NESSIE security report” as Deliverable D20 of NESSIE.

Table of Contents

List of Tables

List of Figures

1. Introduction

1.1 NESSIE project

The NESSIE project is a three year project (2000-2002) that is funded by the European Union's *Fifth Framework Programme*. The main objective of the NESSIE project is to put forward a portfolio of strong cryptographic primitives of various types. Further details about the NESSIE project can be found at the NESSIE website `http://www.cryptonessie.org/`.

The start of the NESSIE project was an open call [475] for the submission of cryptographic primitives as well as for evaluation methodologies for these primitives. This call includes a request for the submission of block ciphers (as for the AES call), but also of other cryptographic primitives including hash functions, stream ciphers, and digital signature algorithms. The call also asked for evaluation methodologies for these primitives. The scope of the call was defined in conjunction with the project industry board, and was published in March 2000. This call resulted in forty submissions. The NESSIE project aims to assess these submissions with the goal of producing a portfolio of cryptographic primitives for use in different environments. The NESSIE project proposes to disseminate the project results widely and to build consensus based on these results by using the appropriate bodies: a project industry board, NESSIE workshops, the 5th Framework programme, and various standardisation bodies.

The NESSIE project has been divided into two phases. In the first phase of the security evaluation, the submissions were analysed by the NESSIE partners. The NESSIE partners also received external comments for some submissions. The outcome of the first phase was a preliminary assessment of all submissions: a security evaluation [478] and a performance evaluation [477]. This was used to decide which of the submissions were to be considered in the second phase [476]. In the second phase, the remaining submissions were subject to more detailed scrutiny to produce the portfolio. This report summarises all the security evaluation of the submissions and is the conclusion of the second phase of security evaluation. The NESSIE project also compiles a performance evaluation of the submissions. This security report together with the performance report form the basis of the decision as to which primitives should be part of the NESSIE portfolio.

The NESSIE evaluation process is an open process. Thus as part of the evaluation process, the NESSIE project welcomes comments about the submitted primitives and the evaluation process, including this report. To facilitate the open evaluation process, there are to be four NESSIE workshops. The first NESSIE workshop, in which the submitted primitives were presented, took place on 13-14 November 2000 at Katholieke Universiteit Leuven (Belgium). The second NESSIE workshop, in which early results concerning the primitives were presented, took place on 12-13 September 2001 at Royal Holloway, University of London (UK). The third NESSIE workshop, in which new results concerning the primitives were presented, took place on 6-7 November 2002 at Siemens AG, Munich (Germany). The fourth NESSIE workshop, which disseminated the results of the project, took place on 26 February 2003 in Lund (Sweden).

1.2 Security evaluation methodology

The NESSIE project has attempted to define a high–level methodology to compare in a fair and acceptable way the submitted primitives. This methodology may evolve according to technical advances, remarks of the NESSIE members, Industry Board or cryptographic experts, and with problems encountered. However it should be noted that it is impossible to produce a definitive security methodology. Cryptographic primitives with completely inadequate security can often be identified, but for other cryptographic primitives, the situation is much less clear-cut. There is neither an automatic method of assessing the security of such a primitive nor a general consensus on the relative importance of different security criteria. The few previous initiatives that have undertaken a similar task to the NESSIE project, such as AES, have been more limited in scope and have reached a subjective judgement by experts on the security of such primitives. We first give the evaluation criteria specified in the NESSIE call [475] and then a list of important issues that NESSIE has considered in making its security evaluations of a submitted primitive. This list is extensive but not complete.

1.2.1 Evaluation criteria in NESSIE call

1. An attack should be at least as difficult as the generic attacks against the type of primitive (exhaustive search, birthday attack, etc.)
2. Primitives will be evaluated against the security claims of the submitter. An attack requiring lower computing resources than claimed would usually disqualify the submission.
3. Primitives will be evaluated within the stated environment. Thus, consideration of vulnerability to side channel attacks (e.g. timing attacks, power analysis) may be appropriate.

1.2.2 Methodological issues

Resistance to cryptanalysis. Clearly, any submission should be resistant at the relevant security level to cryptanalytic attacks. Indeed, in the NESSIE call for submissions [475], it is stressed that failure to be resistant to such an attack would usually disqualify a submission. However, when assessing the relevance of a cryptanalytic attack, other factors such as the volume and type of data required to mount the attack will be considered.

Design philosophy and transparency. An important consideration when assessing the security of a cryptographic primitive is the design philosophy and transparency of the design of that primitive. It is easier to have confidence in the assessment of the security of a primitive if the design is clear and straightforward, and is based on well-understood mathematical and cryptographic principles. This is particularly relevant when making relative comparisons between primitives (see below).

Strength of modified primitives. One common technique used to assess the strength of a primitive is to assess a modified primitive, for example by changing or removing a component or reducing the number of rounds. Conclusions about the original primitive based on an assessment of the modified primitives have to be carefully considered as the inference may or may not be straightforward.

Relative security. When assessing primitives designed to operate to the same security level in similar environments, it is natural to wish to compare their security. However, care has to be taken when making such comparisons. One measure that has been suggested for primitives based on an iterative algorithm is the *security margin*, which measures the gap between the maximum number of broken rounds and the total number of rounds, but there is no general consensus about its definition or use. Furthermore, whilst the NESSIE project tries to ensure that each submitted primitive receives equivalent cryptanalysis, it is the case that some designs are easier to analyse than others (as discussed above). However, it is felt that there should be some security margin to protect against cryptanalytic advances.

Cryptographic environment. In certain cryptographic environments, a cryptographic primitive may have been designed to possess intrinsic security advantages or disadvantages. An example would be a primitive that is resistant to power or timing attacks when implemented on a smart card. Such properties would be considered when assessing the security of a primitive.

Statistical testing. The NESSIE project is carrying out statistical testing of submitted primitives (where relevant). The purpose of this statistical testing is to highlight anomalies in the operation of the primitive that may indicate cryptographic weakness and require further investigation.

1.3 Structure of the Report

This report is split into distinct chapters for distinct categories of primitives. For a reader with some knowledge in cryptology, each chapter is intended to be self-contained. However, this report is not a course on cryptology; the reader should refer to textbooks [265, 270, 441, 598]. Each chapter has the following sections:

1. **Introduction.**
   The introduction defines what is the category of primitives that is considered.

2. **Security requirements.**
   The security model and common attacks are explained. The assessment process by NESSIE is described.
3. **Overview of the common designs.**
   Common designs are described, and how the primitives submitted to NESSIE fit in with these. Current standards are reviewed.
4. **Analysis of all primitives submitted to NESSIE.**
   The analysis of each primitive submitted to NESSIE is summarised. Primitives that were not selected for Phase II have a shorter review than the ones that were studied during the whole NESSIE process.
5. **Conclusion.**
   Recommendations are made for a choice of primitives that should be part of the NESSIE portfolio.

The categories of primitives considered in this report are slightly different from the categories defined in the call. This is more consistent with the list of primitives submitted to NESSIE and with the way in which these primitives were analysed. They are:

Ch. 2. *Block ciphers*
Ch. 3. *Stream ciphers and pseudo-random number generators*
Ch. 4. *Hash functions*
Ch. 5. *Message authentication codes*
Ch. 6. *Asymmetric encryption*
Ch. 7. *Digital signature schemes*
Ch. 8. *Digital identification schemes*

An appendix on side-channel attacks and the bibliography conclude this report.

1.4 The submissions received by NESSIE

- **Block ciphers**
  - **Anubis.** [37] – has been tweaked
  - **Camellia.** [24]
  - **CS-cipher.** [234]
  - **Grand Cru.** [112]
  - **Hierocrypt.** [493] – Hierocrypt-L1 and Hierocrypt-3
  - **IDEA.** [387]
  - **Khazad.** [39] – has been tweaked
  - **MISTY1.** [426]
  - **Nimbus.** [410]
  - **Noekeon.** [180]
  - **NUSH.** [391]
  - **Q.** [434]
  - **RC6.** [324]
  - **SAFER++.** [420]

  - **SC2000.** [569]
  - **SHACAL.** [281] – SHACAL-1 and SHACAL-2.

- **Stream ciphers and pseudo-random number generators**
  - **BMGL.** [285]
  - **SNOW.** [214]
  - **SOBER.** [289] – SOBER-t16 and SOBER-t32
  - **LEVIATHAN.** [435]
  - **LILI-128.** [187]

- **Hash functions**
  - **Whirlpool.** [38] – has been tweaked

- **Message authentication codes**
  - **UMAC.** [379]
  - **Two-Track-MAC.** [609]

- **Asymmetric encryption**
  - **ACE-KEM.** [563] – Upgrade of ACE-Encrypt
  - **EPOC.** [239] – EPOC-1, EPOC-2 and EPOC-3
  - **ECIES.** [320]
  - **PSEC.** [238] – PSEC-1, PSEC-2, PSEC-3 and PSEC-KEM
  - **RSA-OAEP.** [325] – Revised to RSA-KEM [584]

- **Digital signature schemes**
  - **ACE Sign.** [563]
  - **ECDSA.** [319]
  - **ESIGN.** [244]
  - **FLASH family.** [514] – FLASH and SFLASH (has been tweaked)
  - **QUARTZ.** [161]
  - **RSA-PSS.** [326]

- **Digital identification schemes**
  - **GPS.** [525]

- **Evaluation methodologies**
  - **General Next Bit Predictor** [294]
    This evaluation methodology appeared to be useless [200], it will not be further mentioned in this report.

1.5 Some mathematical notations

$\mathbb{Z}$	the set of integers
$\mathbb{Z}/n\mathbb{Z}$	the set of integers modulo n
$(\mathbb{Z}/n\mathbb{Z})^*$	the non-zero elements of $\mathbb{Z}/n\mathbb{Z}$
$(\mathbb{Z}/n\mathbb{Z})^\times$	the invertible elements of $\mathbb{Z}/n\mathbb{Z}$, which form a multiplicative group with $\phi(n)$ elements
QR_n	the squares in $(\mathbb{Z}/n\mathbb{Z})^\times$ (quadratic residues)
$\mathbb{F}_q$	the finite field with q elements
$(\mathbb{F}_p)^n$	the n-dimensional vector space on $\mathbb{F}_p$
$\langle G \rangle$	the cyclic group generated by G
1^λ	a bitstring of length λ of all 1, used as a security parameter
φ	Euler phi function, defined by $\varphi(\prod_i p_i^{n_i}) = \prod_i (p_i - 1) p_i^{n_i - 1}$
$f = O(g)$	f is dominated by g, which means that $\lvert f/g \rvert$ is upper bounded
$f = o(g)$	f is negligible in g, which means that $f/g \to 0$
$f \ll g$	alternate notation for $f = o(g)$

NB: if p is prime then $\mathbb{F}_p = \mathbb{Z}/p\mathbb{Z}$ and $(\mathbb{Z}/p\mathbb{Z})^* = (\mathbb{Z}/p\mathbb{Z})^\times$.

2. Block ciphers

2.1 Introduction

Block cipher encryption provides confidentiality by transforming a plaintext message into a secure ciphertext message, where the precise function implemented by the block cipher is determined by a secret key. This secret key, or series of keys, is only known by legitimate users of the block cipher. Whereas a stream cipher contains a memory, embodied in its current state, a block cipher is memoryless outside its current block and therefore has no current state. A mode of operation of a block cipher partitions a plaintext message into a series of blocks which are then encrypted one block at a time, although a block cipher can be used as a component in a stream cipher, and also for pseudorandom number generators, MACs, hash functions, and signature schemes. Block cipher encryption is the most well-known form of symmetric-key encryption — the word *symmetric* implies that both transmitter and receiver of the ciphertext have knowledge of the secret key. Block ciphers have been around in one form or another for a very long time, for instance the substitution cipher, and the transposition cipher. However, many of the ideas that permeate modern block cipher design were inspired by the work of Shannon [567] around 1949. He first elucidated the concepts of confusion and diffusion that are still primary design criteria for any state-of-the-art block cipher. The rate at which cryptographic and information theory have developed has accelerated over the last thirty years or so, with the result that many *rules* for block cipher design are now well-accepted amongst the majority of cryptographers [354]. However, it remains the case that no practical block cipher is provably secure and, consequently, new design criteria are still being discovered, these often as a response to emerging novel attacks on block ciphers. Typically, a block cipher design is proposed according to well-accepted and well-founded *rules*, and this inevitably forces the cryptanalyst to attempt to attack the cipher in some unforeseen way. These unforeseen attacks, if successful, lead in turn to the extending of the canon of design criteria — and so the discipline progresses. It should therefore come as no surprise that, during the three-year lifespan of NESSIE, new designs and new attack methods have been proposed, and new questions raised. To some extent, some of the block cipher designs submitted to NESSIE have been influenced by the knowledge gained from the search for the new Advanced Encryption Standard (AES), immediately prior to NESSIE. In

[0] Coordinator for this chapter: UiB — Matthew Parker

particular the inclusion of high-diffusion ciphers in both AES and NESSIE has encouraged the development of algebraic and non-statistical attacks on block ciphers. It should be emphasised that most of the block ciphers submitted to the NESSIE project appear to be secure, acceptably efficient, and practical for use in real systems. The criteria for selection for the NESSIE portfolio therefore rests, to some extent, on secondary considerations, in particular on the identification of potential weaknesses and expected performance on different platforms. In this chapter, attacks on the various block ciphers submitted will be described, and potential security weaknesses of the candidates identified.

It should also be noted that there are four usual modes of operation for a block cipher. The most common mode is called Electronic Codebook Mode (ECB) which takes disjoint plaintexts and outputs disjoint ciphertexts. There is also Cipher Block Chaining (CBC) mode, where the encryption of a block depends on the encryptions of previous blocks. Then there is Cipher Feedback (CFB) mode which is the first of the two stream cipher modes, where one m-bit character at a time is encrypted. Finally there is Output Feedback (OFB) mode where, in contrast to CFB mode, the stream bits are not dependent on the previous plaintexts, i.e. only the stream bits are fed back, not the ciphertext as in CFB mode. NIST has now published a new standard for block cipher modes [632], and Counter Mode (CTR) has been added.

2.1.1 Block Cipher — A Formal Definition

Before discussing security aspects we first give a more precise definition of a block cipher [270]. A block cipher is a function $E : \{0,1\}^K \times \{0,1\}^N \to \{0,1\}^N$ that takes two inputs, a K-bit key k and an N-bit plaintext P, to return an N-bit ciphertext $C = E(k,P)$. For any block cipher, and any key k, the function E_k is a *permutation* on $\{0,1\}^N$. This means that it is a *bijection*, i.e. a *one-to-one* function of $\{0,1\}^N$ to $\{0,1\}^N$. Accordingly, it has an inverse, E_k^{-1}. Both the cipher and its inverse E^{-1} should be easily computable, meaning that given k, P we can compute $E(k,P)$, and given k, C we can compute $E^{-1}(k,C)$.

2.2 Security requirements

Block cipher encryption is a method to transform a plaintext message of blocklength N bits by encrypting it to a ciphertext message of blocklength N bits, where the encryption operation is determined by a secret key string of length K bits, where the key is often chosen uniformly at random. The inverse operation, block cipher decryption, takes the N-bit ciphertext and decrypts it back to the N-bit plaintext using the same secret key string of length K bits. The aim is to make it practically impossible to retrieve the plaintext from the ciphertext without knowledge of the K-bit secret key. Decryption is only possible if the encryption function is invertible (i.e. if it is a bijection) and this restricts the choice of possible N-bit block ciphers to one of $(2^N)!$ block ciphers. However, parameterisation by a secret key of length K bits further restricts the set of block ciphers realised by a particular design to a maximum of 2^K block ciphers (which, for any reasonable N and K, is an infinitesimally small fraction of the complete space of $(2^N)!$ block ciphers). The problem of block cipher design is to determine which set of 2^K block ciphers to choose such that, for an unknown fixed key, it is virtually impossible to say anything about the ciphertext resulting from a known or chosen plaintext, or to say anything about the fixed key, given prior knowledge of a few plaintext/ciphertext pairs. Note that, if the plaintext contains exploitable redundancy, then one may be able to attack the cipher using ciphertext only. Any effective block cipher scheme must be realised efficiently in time and space, with as little implementation cost as possible. The practical trade-off is therefore to design a block cipher which is both sufficiently secure, and satisfactorily efficient in terms of hardware/software space and time resources. It should be emphasised that the complete design of the block cipher, along with all ciphertexts, is considered public knowledge, with only the secret key remaining unknown to attackers of the system. Clearly, knowledge of the secret key implies knowledge of the plaintexts that were encrypted using that secret key. A block cipher with a secret key is considered perfect if, for all plaintexts, P, and ciphertexts, C, it holds that $\Pr(P) = \Pr(P|C)$ [567]. If, for a fixed K-bit key, an N-bit block cipher is used to encrypt $\lfloor \frac{K}{N} \rfloor$ plaintexts, then the cipher can always be chosen to be the one-time pad so that, in this special case, the encryption is provably secure and the block cipher perfect — a one-time pad is a symmetric key block cipher where K key bits are used, only once, to encrypt K plaintext bits, where the K corresponding ciphertext bits are the XOR of the plaintext bits with the key bits. In such a situation the ciphertext and plaintext are statistically independent. However, in most situations the one-time pad is impractical as far too many secret keys must be used. Therefore it is highly desirable to securely encrypt T plaintexts using the same, fixed K-bit secret key, where $T \gg \lfloor \frac{K}{N} \rfloor$. Most modern block ciphers seek to maximise T, whilst still achieving an acceptable security, via a combination of confusion, which makes the relationship between key and ciphertext as complicated as possible, and diffusion which seeks to eliminate any redundancy in the plaintext. Diffusion also makes it difficult for any attacker to partially approximate the cipher.

Theoretically the ideal block cipher, from a security viewpoint, would involve one very large, well-chosen N-bit Substitution Box (S-Box), keyed by K key bits and, ideally, it would be impossible to decompose this S-box into smaller sub-units. However this immediately implies a huge implementation complexity, so any practical block cipher will, instead, combine relatively small sub-units to *confuse* (e.g. S-boxes) and *diffuse* (e.g. linear transformation layers) the plaintext. Moreover, these sub-units will be applied iteratively as keyed rounds, parameterised by sub-keys which are derived from the master K-bit key. This decomposition into practical sub-units constitutes a trade-off between security and acceptable complexity. All the block ciphers submitted to NESSIE are iterated over multiple rounds, and all of them utilise a key-schedule to derive round keys from a master key. It is this decomposition into sub-processes that provides the cryptanalyst with ammunition for an attack. In spite of the above compromises, it is an accepted design principle that encryption using a block cipher, selected via a randomly-chosen key, should look like encryption by a randomly-chosen invertible function over N bits.

2.2.1 Security model

It is the nature of scientific discovery that the initial models are, to a large extent, heuristic — intended security against well-known attacks. These heuristics later give way to formal proof of security versus resources needed. What is certain is that, with respect to practical block cipher security, very little can (yet) be proved to any high-degree of accuracy — hence the existence of NESSIE. However, there exist a number of accepted security models which can tell us something about the block cipher under consideration:

- **Unconditional Security (Perfect Secrecy).** Shannon assumed that an adversary has unlimited computational resources. In his model, secure encryption only exists if the size of the key is as large as the number of secret bits to be exchanged remotely using the encryption system. Perfect secrecy is possible only if no more than $\lfloor \frac{K}{N} \rfloor$ plaintexts are enciphered using a fixed key (e.g. the one-time pad), so unconditional security is not a useful model for practical block ciphers.
- **Security Against Polynomial Attack.** In contrast to Condition Security, modern cryptography assumes the adversary's computation is resource-bounded. Specifically, it is assumed that the adversary is a probabilistic algorithm which runs in polynomial time, and security is claimed with respect to the feasibility of breaking the cryptosystem. This model arises out of complexity theory considerations where adversaries are assumed to possess only polynomial computational resources — polynomial in the size of the input to the cipher in bits. The model typically conducts worst-case and asymptotic analyses to determine whether polynomial attacks on a cipher exist. Even if they do exist, it is not guaranteed that such attacks are practical. This security model tends to provide an understanding as to the type (class) of problem embodied by a block cipher, without providing exact figures.
- **"Provable" Security.** Typically this can mean one of two things. Firstly, if it can be shown that breaking a block cipher is as difficult as solving some well-known hard problem (e.g. discrete log or factoring) then the cipher is considered provably secure. This is, of course, misleading as the hard problem on which it is based is usually not provably hard. This relates to a very fundamental open question in computer science as to whether these hard problems are in P or in NP. In fact, provable security requires a proof that P $\neq$ NP, and the existence of one-way functions which are hard *on the average*, but which can be solved quickly given some extra information [267] (pages 27-28). Note that these are *asymptotic* complexity measures — one is assessing the level of complexity as the input size, in bits, asymptotes to infinity. The strategy of mapping cryptosystems to hard problems is very useful for practical analysis of the cipher, although this model is more often applied to public-key cryptosystems. Secondly, a block cipher may be shown to be provably secure against a known

sub-class of attacks. One example of this is the provable security against linear and differential cryptanalysis used, for example, by the designers of MISTY1. It should be emphasised however that this obviously does not mean that the cipher is secure against all attacks.

- **Practical Security.** In this model a block cipher is considered computationally secure if the best-known attack requires too much resource by an acceptable margin. This is a very practical model as one can test the cipher with different known attacks, probing for weakness, and then give an assessment of the cipher's strength against such attacks in terms of time/space resources needed. This model tends to provide the most answers, and most of the analysis in NESSIE was of this type. However, it says nothing about the security level with respect to yet unknown attacks.
- **Historical Security.** It is quite useful to assess the security level of a block cipher according to how much cryptanalytic attention the cipher has attracted over the years. For example, both Cipher A and Cipher B could be considered excellent cipher designs. But Cipher A may be ten years older and has therefore been under scrutiny for many more years than Cipher B without any serious security flaws found in it. This inevitably inspires a certain confidence in the older cipher and suggests that the time-scale over which projects such as NESSIE and AES operate is only sufficient to draw preliminary conclusions as to the security of a completely new cipher. However, it should also be noted that the effort spent on breaking a cipher cannot always be measured reliably from the time passed.

2.2.2 The Block Cipher as a Pseudorandom Permutation

It is natural to consider a block cipher as a set of permutations. In this context we can consider a *distinguisher* which differentiates between a randomly-selected pseudorandom permutation and a permutation which is randomly selected from the set of permutations generated by the block cipher. This section investigates this approach in more detail, considering the asymptotic limit as the size of the input and output to and from the block cipher approaches infinity. (We here summarise and paraphrase some of the definitions given by Goldwasser and Bellare [270] and others [265].)

Definition: Let U_N denote a random variable uniformly distributed over $\{0,1\}^N$. A *pseudorandom function* is one of an infinite set of functions with increasing input sizes, $\{f(U_N, K)\}, N = 1, \ldots, \infty$, with the property that the input-output behaviour of a random instance of the set is *computationally indistinguishable* from that of a random function. *Pseudorandom permutations* can be described similarly.

A block cipher, $E(k, P)$, can be considered as a set of 2^K permutations on the message space, each instance, $E_k(P)$, of the set being a distinct permutation and obtained by fixing the key k. In this setting one can model the following attacks on the cipher. Let g be a function drawn at random from the set of all N-bit permutations, as $N \to \infty$. The adversary gets an oracle for g, and can also get an oracle for g^{-1}, these relating to chosen plaintext and chosen ciphertext attacks, respectively.

Definition: A *one-way function*, $f : \{0,1\}^* \rightarrow \{0,1\}^*$, is a function which is *easy* to compute but *hard* to invert. By *easy* we mean that f can be computed by a deterministic polynomial time algorithm, and by *hard* we mean that any probabilistic polynomial time algorithm attempting to invert f will succeed with *negligible* average probability (where the average probability is taken over the elements in the domain of the function f). More formally, for every probabilistic polynomial time algorithm, A', every polynomial $p(\cdot)$, and all sufficiently large N's,

$$\Pr(A'(f(U_N), 1^N) \in f^{-1}f(U_N)) < \frac{1}{p(N)}$$

where U_N denotes a random variable uniformly distributed over $\{0,1\}^N$, and 1^N is some auxiliary input to the algorithm A'.

The block cipher can be considered secure if it can be shown to be equivalent, asymptotically, to a set of pseudorandom functions or pseudorandom permutations. In other words, we cannot distinguish the output ciphertext bits from random output. This allows us to relate block ciphers to one-way functions as follows. Given an N-bit plaintext message, P, and a K-bit key, k, the block cipher function, E, produces an N-bit ciphertext output message, C, where $C = E(k, P)$. Then, for P fixed to p, we can define f such that $f(k) = E(k, P = p)$. Luby and Rackoff show how to build a pseudorandom permutation from a pseudorandom function using a few rounds of a Feistel construction [404], and in [405] they prove that, asymptotically, f is a one-way function on the assumption that E is a set of pseudorandom functions. Therefore, retrieving the key, k, using $k = f^{-1}$, is proven to be hard. However, the notion of a one-way function is weaker than the notion of a secure block cipher. For example, a one-way function may leak half of its input and still be one-way (non-invertible). This demonstrates the need to introduce the idea of '*hard-core*' bits of a function. Given an efficient algorithm to predict the value of a hard-core bit, one can construct an algorithm inverting the one-way function. For a secure block cipher all bits have to be hard-core bits.

Distinguishing Attacks. By viewing a block cipher as a set of permutations we can develop a *distinguisher* which compares and differentiates between the block cipher and the 'ideal' set of random permutations. This is done as follows.

Let g be a function drawn at random from the set, $\mathbf{D}$, of all N-bit permutations. Let g' be a function drawn at random from the set of N-bit permutations, $E(k, P)$. In practice this is achieved by drawing a K-bit key, k, at random from the set of all 2^K K-bit keys. The adversary, A, is a given a series of N-bit permutations, g'', and its job is to determine whether its series of g'' are a series of permutations, g, or a series of permutations, g'. To achieve this the adversary uses an algorithm $A^{g''}$ that takes a function, g'', as input, and returns a bit, d. The *advantage* of the adversary who uses algorithm $A^{g''}$ is given by,

$$\text{advantage}_{E,A} = |\Pr(A^g = 1) - \Pr(A^{g'} = 1)|$$

and this *advantage* measures the ability of algorithm $A^{g''}$ to distinguish between a function g taken at random from $\mathbf{D}$ and a function g' taken at random from the set of N-bit permutations, $E(k, P)$. If we now consider the set of all algorithms $\{A^{g''}\}_{t,q,\mu}$ having time complexity at most t, making at most q oracle queries, such that the sum of the lengths of these queries is at most μ bits, then we can define the pseudorandom permutation advantage (prpadvantage) of E as follows,

$$\text{prpadvantage}_E(t, q, \mu) = \max_{\{A^{g''}\}_{t,q,\mu}} \{\text{advantage}_{E,A}\}$$

where the maximum is over the set $\{A^{g''}\}_{t,q,\mu}$. We can say that a set, $E(k, P)$, which represents a block cipher, is a secure pseudorandom function if prpadvantage$_E(t, q, \mu)$ is *small* for *practical* values of the resource parameters, t, q, μ.

In this section we have defined the security of a block cipher in terms of a secure pseudorandom function. The scenario is that of a *chosen-plaintext attack* where the attacker is assumed to have control over the input plaintext, P. A similar scenario can be developed for a *chosen-ciphertext attack*, and in [270], the chosen-ciphertext attack is also assumed to give the adversary more power: not only can it query g, but it can directly query g^{-1}. In the above we have assumed that an ideal block cipher is a set of permutations (more generally, functions) with the property that the input-output behaviour of a random instance of the family is, asymptotically, *computationally indistinguishable* from that of a random permutation (function). This is a much stronger assumption than just assuming the block cipher is secure against key recovery. However, distinguishing attacks typically use a *distinguisher* similar to that defined above to recover a subset of key bits. But there could exist better attacks that break the cipher as a pseudorandom permutation (function) without recovering the key, although no such attacks are currently known. Conversely, [270] proves that any function family that is insecure under key-recovery is also insecure as a pseudorandom permutation (function).

2.2.3 Classification of attacks

As stated previously, most of the analysis done by NESSIE falls into the Practical Security model. The success-level of an attack is usually measured according to time, memory, and data complexities needed:

- Time complexity needed for an attack on a block cipher is the number of steps required for the attack algorithm. This often reduces to the number of decryptions. However this unit is not always appropriate, for instance the Gaussian elimination used to solve a system of equations describing the cipher will use very different units of time complexity.
- Memory complexity needed for an attack on a block cipher is typically measured in terms of the amount of storage required for the attack algorithm.
- Data complexity is the number of texts the attacker gets from the encryption oracle.

Typically, Memory is much more expensive than Time, for example an attack that requires 2^{64} Memory is very expensive in comparison to 2^{64} steps of an algorithm. Data is also considered expensive and should be minimised. Various tradeoffs are possible, and the complexity of the attack is usually taken as the max(Time,Data). For more practical attacks it may be useful to trade Data for Time. An attack is considered successful in theory, and the block cipher considered *broken* if the Time complexity required is of order $< 2^K$, where K is the number of key bits used to parameterise the cipher. In this case one says that the block cipher is broken if the K key bits can be guessed in time faster than exhaustive key search, and partially broken if some of the plaintext bits can be discovered in time faster than exhaustive key search. Also, for a fixed key, the complete cipher can be characterised if all 2^N different plaintexts are encrypted. This puts an upper bound on the Data Complexity required to mount an attack to 2^N. A block cipher is considered secure if no attack requires both Time and Data complexity significantly less than 2^K and 2^N, respectively. Very often it is difficult to mount an attack on the complete R-round cipher so many (most) attacks break *reduced-round* versions of the cipher. Therefore another way to assess the security of a cipher is to quote the maximum number of rounds of the cipher that have currently been broken. It is the aim of the block cipher designer to make the cipher look as much like a random bijection as possible. Therefore any process which can, for a fixed key, distinguish the cipher from a random cipher, constitutes an attack on the cipher. Such attacks are called Distinguishing attacks, which encompass many of the most effective attacks used today.

The types of attack that can be performed depend on what resources are available to the adversary. They can be classified as follows:

- Ciphertext-only attacks. The adversary has access to a set of ciphertexts and also knows something about the nature of the plaintext.
- Known plaintext attack. The adversary has access to a set of plaintext-ciphertext pairs.
- Chosen plaintext attack. The adversary is able to choose a series of plaintexts and has access to the resultant ciphertexts.
- Adaptively chosen plaintext attack. The adversary is able to choose a series of plaintexts, where the choice of each new plaintext is influenced by the ciphertexts obtained from the previous plaintexts.
- Chosen ciphertext attacks. Similar to chosen and adaptively chosen plaintext attacks, but with the roles of plaintext and ciphertext reversed.
- Combined chosen plaintext/ciphertext attacks. In this case the adversary is able to choose both plaintexts and ciphertexts. The Boomerang attack is an example of such an attack.

The chosen text attacks are always the most powerful attacks but, of course, are often less realistic in a practical context. Ideally the designer or analyst should try to prove that the cipher is secure against adaptively chosen attacks.

Because of the birthday attack on, for instance, Cipher Block Chaining, and Accumulated Block Chaining modes, it is recommended that a single key is used to encrypt at most $2^{N/2}$ ciphertexts [354]. This is because one only requires about

$2^{N/2}$ ciphertexts to obtain a matching pair of ciphertexts with probability $> \frac{1}{2}$. It should be noted that this restriction is independent of key size.

Attacks typically fall into two main groups, firstly those that are statistical in nature, and secondly those that are largely non-statistical. However we do not distinguish these two types of attacks here as most attack strategies that work with probability one can also be envisaged in a scenario with probability strictly less than one. We now briefly describe the best-known attacks.

2.2.3.1 Exhaustive Key Search

Applicable to all ciphers, the attack only needs a few known plaintext-ciphertext pairs. The adversary tries all keys one by one to check whether the given plaintext encrypts to the given ciphertext. The attack requires only approximately $\lceil \frac{K}{N} \rceil$ pairs to determine the key.

2.2.3.2 Differential Cryptanalysis

This chosen plaintext attack was first applied to DES by Biham and Shamir [77]. It was the first attack which could (theoretically) recover DES keys in time less than exhaustive search. Typically, pairs of chosen plaintexts, (P_0, P_1) with a fixed difference, $\Delta = P_0 - P_1$ are chosen, where the difference operation "$-$" is usually chosen to be the group operation that is used to add the fixed round key. Thus $\Delta = (P_0 + k) - (P_1 + k)$, so the important point is that the difference, Δ, is independent of the key, k, chosen. Moreover, the difference is chosen so that with some acceptably high probability, this difference, Δ, propagates to an output difference Δ' at the output of the round. If one can propagate differences through all rounds with sufficiently high probability, then this allows one to devise a key-invariant approximation to the core (central) rounds of the cipher which can establish a non-random difference relationship between bits specified by the mask, I, near to the input, and bits specified by the mask, O, near to the output of the cipher. This in turn enables the adversary to guess round key bits of the first and last round, and compute the differences at bits specified by I and O according to this guess. If the guess is correct then the differences at pair (I, O) will agree with the guess for the differences at (I, O) with probability p. If p is large enough then choosing enough plaintext-ciphertext pairs will enable the adversary to determine whether the guess was correct. In this way the adversary can determine the round key bits of outer layers of the cipher and work inwards. The processing complexity of a differential attack is approximately $\frac{c}{p}$, where c is a small constant. For many block ciphers the round key is added using XOR. In this case the notion of difference between plaintexts also uses XOR, thereby ensuring key-independence for the approximation. However, other notions of difference can also be useful. One should distinguish between a characteristic and a differential. Whereas a characteristic specifies one particular evolution of differences through the cipher, a differential takes into account all possible paths through the cipher that would yield the same output difference and sums their combined probabilities [388]. Therefore Differential Cryptanalysis using differentials as opposed to just characteristics always achieves higher probabilities. However, the gain is not always significant. Maximum Average Differential Probability is also sometimes used to assess the goodness of Differential Cryptanalysis. Differential Cryptanalysis typically uses the concept of a Markov Cipher [388] where the characteristic probability is independent of the actual round inputs and is computed over all possible choices of the round key. Moreover, it was shown by Lai *et al.* [388] that, if the round keys are independent, then so are the characteristic probabilities, and this allows these probabilities to be combined relatively simply. However, typically the attacker gets multiple encryptions under the same key, which means the round keys may appear dependent. Fortunately the Markov assumption can be shown to hold approximately for virtually all keys [383].

2.2.3.3 Truncated Differential Cryptanalysis

Instead of trying to propagate a complete difference through every cipher round, one can aim to propagate only part of the difference — hence truncated differentials [352]. The advantage here is that the partial difference propagations may occur with much higher probability than full difference propagations.

2.2.3.4 Impossible Differential Cryptanalysis

Here the aim is to find a differential that occurs with zero probability over a number of rounds of the cipher. Then by key guessing outside the core approximated rounds one can rule out certain key guesses if they allow the forbidden differential to occur. Such attacks have been applied by Knudsen [353], and by Biham *et al.* [65]. One particularly useful fact in [65,353] is that any Feistel cipher with bijective round functions has an impossible differential after 5 rounds (see Sect. 2.3.1 for a definition of Feistel).

2.2.3.5 Higher Order Differential Cryptanalysis

This is a recursive extension of differential cryptanalysis where one looks to establish probabilistic nested differences across rounds of the cipher [385]. Thus an s-th order differential requires sets of 2^s chosen plaintexts with fixed pairwise difference between members of the set [352,384]. It follows that an $(s+1)$-th order differential of a function of nonlinear order d is zero. This attack was successfully applied by Nyberg and Knudsen against the cipher of [488] which is *provably secure* against differential attack. The boomerang attack of Wagner [616] can be viewed as a special type of second-order differential attack, and is suited to ciphers where one first-order differential applies to the first half of the cipher, and another first-order differential applies to the second half of the cipher.

2.2.3.6 Linear Cryptanalysis

This known plaintext attack was first applied to FEAL by Matsui and Yamagishi [427] and to DES by Matsui [421]. It is conceptually similar to differential cryptanalysis as the aim is to establish essentially key-invariant approximations. This follows because if the linear relationship $x = y$ holds with probability $\frac{1}{2} + \delta$, then $x = y + k$ holds with probability $\frac{1}{2} \pm \delta$. Thus the bias of the approximation away from $\frac{1}{2}$ is still δ and is unaffected by key change. One approximates subunits of the cipher by linear approximations which hold with some probability different from $\frac{1}{2}$. If these approximations can be connected up over all core rounds of the cipher then one can establish linear relationships between bits specified by the mask, I, near the input, and bits specified by the mask, O, near the output of the cipher. This in turn enables the adversary to guess round key bits of the first and last round, and compute bits at I and O according to this guess. If the guess is correct then the pair (I, O) will agree with the guess at (I, O) with probability $\frac{1}{2} \pm b$. If the bias, b, is large enough then choosing enough plaintext-ciphertext pairs will enable the adversary to determine whether the guess was correct. In this way the adversary can determine the round key bits of outer layers of the cipher and work inwards. The complexity of a linear attack is approximately $c \cdot \frac{1}{|b|^2}$ for some constant, c, where b is called the bias. Linear hulls are an extension of the above technique [487] and exploit the fact that there may be more than one linear characteristic (propagation route) through a cipher that begins and ends at the same place. This allows one to sum up the individual biases for each characteristic, although it is possible that the individual characteristics can cancel each other out. The Maximum Average Linear Hull Probability is also sometimes used as a metric to assess the goodness of linear cryptanalysis. A similar development of linear cryptanalysis is to re-use the data one already possesses in different ways, by applying different linear approximations and pooling the information one receives about the key. This technique is referred to as Multiple Linear Approximations [143, 335]. The technique is particularly useful if two or more of the best linear approximations have comparable biases. A further generalisation of linear cryptanalysis looks for nonlinear approximations through parts of the cipher. Although these nonlinear approximations typically exist with higher probability than linear approximations they do not usually preserve key-invariance, so are typically used at either end of the cipher to positively enhance linear biases [361, 370].

2.2.3.7 Differential-Linear Cryptanalysis

Many attack techniques can be combined to form a hybrid attack on a cipher. An example of this is Differential-Linear cryptanalysis. In [389] a differential is established through part of the cipher and used to create a linear approximation with probability 1. This technique is also used in [70, 75] where the differential part is used to create linear approximations which have probability strictly less than 1.

2.2.3.8 Boomerang Attacks

This attack is based on differential techniques and was first described by Wagner [616]. *Boomerang attacks* allow for a more extensive use of structures than is available in conventional differential attacks. More specifically, the boomerang attack is a differential attack that attempts to generate a quartet structure at an intermediate value halfway through the cipher. This quartet structure is illustrated in Fig 2.1.

The aim, as shown in Fig 2.1, is to cover the plaintext pair P, P' with the differential characteristic for E_0, and to cover the plaintext pairs P, Q and P', Q' with the differential characteristic for E_1^{-1}. In this case it can be shown that the plaintext pair Q, Q' is perfectly set up to use the differential characteristic $\Delta^* \rightarrow \Delta$ for E_0^{-1}. We can summarise this scenario as follows:

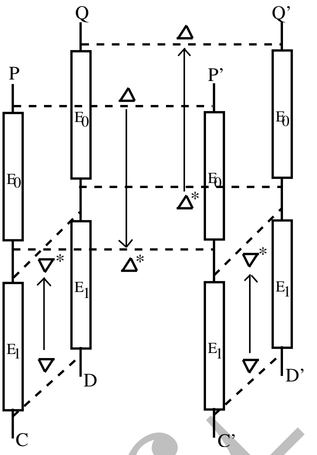


Fig. 2.1. A Typical Boomerang Attack

$$\begin{aligned}
&E_0(Q) \oplus E_0(Q') \\
&\quad = E_0(P) \oplus E_0(P') \oplus E_0(P) \oplus E_0(Q) \oplus E_0(P') \oplus E_0(Q') \\
&\quad = E_0(P) \oplus E_0(P') \oplus E_1^{-1}(C) \oplus E_1^{-1}(D) \oplus E_1^{-1}(C') \oplus E_1^{-1}(D') \\
&\quad = \Delta^* \oplus \nabla^* \oplus \nabla^* \\
&\quad = \Delta^*
\end{aligned}$$

Thus one will have the same difference in the plaintexts Q, Q' as found in the original plaintexts, P, P', which is why the attack is called the boomerang attack. To set up this quartet one can generate $P' = P \oplus \Delta$, then obtain the encryptions C, C' of P, P' with two chosen-plaintext queries. Then one generates D, D' as $D = C \oplus \nabla$ and $D' = C' \oplus \nabla$. Finally D, D' are decrypted to obtain the plaintexts Q, Q' with two adaptive chosen-ciphertext queries.

2.2.3.9 Mod n Cryptanalysis

This attack by Kelsey *et al.* [346] is a generalisation of a linear attack, and uses the property that some bit groupings (words) within the cipher may be biased modulo n, where n is typically some small integer. It has been shown that ciphers that use only bitwise rotations and additions, mod 2^{32} are particularly vulnerable to such attacks.

2.2.3.10 Weak-Key Classes

A weak-key class refers to any subset of size 2^s of the key space such that, for this class, an attack is known requiring fewer key guesses than exhaustive search, where exhaustive search here means 2^{s-1} key guesses. For instance, for some block ciphers the round key is added using integer addition or even multiplication of some form. And some block ciphers use even more general keyed nonlinear components. In these cases, differential and linear cryptanalysis may benefit from a consideration of a sub-class of the complete space of keyed block ciphers for this particular design — a weak-key class. Typically this class will be defined by fixing a well-chosen subset of the key bits so that the residual cipher is then open to attack. This is a useful way of assessing whether the cipher is secure for all possible keys. Block ciphers which add the key using XOR are more immune to weak-key class attacks than block ciphers which add some or all of the key nonlinearly [453]. A stronger form of the attack demands that the attacker can identify whether the key used is in the weak-key class. This is known as the key-class membership test, and a practical attack should use a low complexity membership test. For instance, Wagner [616] uses a boomerang attack as a weak-key class identifier.

2.2.3.11 Related-Key Attacks

There are several variants of this attack depending on the privileges of the adversary. Either the adversary obtains encryptions under one fixed key, or he obtains encryptions under several keys where there is either a known or a chosen relation between the keys. The fixed key variant was first used by Knudsen in [350] to establish a chosen plaintext attack, reducing an exhaustive key search by four times. The version using several keys was developed in [59, 345, 351]. It is noted by Biham [59] that the related-key attacks discussed in that paper are completely independent of the number of rounds of the cipher. Slide attacks can be a variant of related-key attacks, and these are discussed later.

2.2.3.12 Interpolation Attack

The interpolation attack was proposed by Jakobsen and Knudsen in [315, 316]. This attack is interesting in that one need not recover the key to break the cipher as the attack seeks to construct a polynomial relationship between plaintext and ciphertext given a set of known plaintext-ciphertext pairs. The attack is often easier if components of the cipher have a straightforward mathematical description or if the polynomial to be approximated is of relatively low degree and with relatively few non-zero coefficients. The number of plaintext-ciphertext pairs required depends directly on the degree of the constructed polynomial — the lower the better. The technique of choice is Lagrange Interpolation. The attack also enables the recovery of round keys once the polynomial is constructed. For instance, although an interpolation attack can predict the ciphertext from the plaintext *without* knowledge of the key, it can also be used to predict the output from the last but one round of the cipher and this, in turn, allows recovery of the last round key. Meet-in-the-middle techniques can be used to reduce the degree of the polynomial to be interpolated [315].

A probabilistic version of the interpolation attack has also been proposed by Jakobsen [314], which views the attack as a problem in coding theory and applies Sudan's algorithm for decoding Reed-Solomon codes [600]. The paper [628] further investigates ways of minimising the degree of the polynomial to be interpolated by changing the irreducible polynomial over which the S-boxes of the block cipher are described.

It is shown by Kurosawa *et al.* [382] how to use Rabin's root finding algorithm in conjunction with the Interpolation Attack so as to find all the possible last round keys that satisfy the interpolated polynomial.

2.2.3.13 Non-Surjective Attack

This attack [541] is applicable to Feistel ciphers where the round function is non-surjective. (See Sect. 2.3.1 for a definition of Feistel). It uses the possibility that the F-function of the Feistel cipher may be non-bijective. This is true for DES, and this attack breaks DES faster than exhaustive search [64, 185].

2.2.3.14 Slide Attack

Slide attacks [90] developed out of related-key attacks [59, 350] and exploit a weakness in ciphers that use identical or periodic round functions. These attacks are, in many cases, independent of the number of rounds of a cipher, and independent of the exact properties of the iterated round function. Let $F_r \circ F_{r-1} \circ \ldots \circ F_1$ denote an r-round iterated cipher, where the F_is are identical or periodically related through the rounds. The adversary looks for pairs of plaintexts P, P^* and their corresponding ciphertexts C, C^* such that $F_1(P) = P^*$ and $F_r(C) = C^*$. This gives an adversary two input-output pairs of one round of the cipher. One can expect, by the birthday paradox, to find such pairs of texts after about $2^{N/2}$ plaintexts. But, for Feistel ciphers, where the round function modifies only half the block, there is also a chosen-plaintext variant which can often cut the complexity down to $2^{N/4}$ plaintexts. The basic slide attack is shown in Fig 2.2.

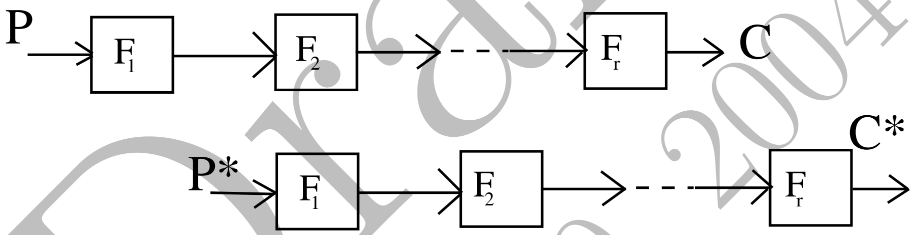


Fig. 2.2. A Typical Slide Attack

For the slide attack to work, F_i should be very weak against known-plaintext attack with two plaintext-ciphertext pairs. Some ciphers include the addition of randomised constants into the key schedule and/or round irregularities, and these protect, to some extent, against a slide attack. For instance, a slide attack by Biryukov and Wagner on MISTY1 is prevented by the inclusion of the nonlinear FL layers after every six rounds of MISTY1 [90].

2.2.3.15 Integral/Multiset Cryptanalysis

There are, in fact, a number of different attack techniques which fall under the umbrella of Integral or Multiset attacks, these being Square attacks [178,179], Integral attacks [371], Multiset attacks [88], and Saturation attacks [408]. They also appeared in the birthday cryptanalysis of Ladder-DES [61]. Saturation attacks assume only permutations are used, and Multiset attacks cover both Saturation and Integral attacks. In fact, Gilbert-Minier's collision attack is an example of a Multiset attack [258]. Whereas in differential cryptanalysis one considers differences of pairs of plaintexts, in integral cryptanalysis [371] one considers sums of plaintexts (integrals) to exploit the degree of balance of the output. The Square attack, which was first applied to the Square cipher by Daemen *et al.* [178, 179] and then to Rijndael [181], is an example of integral cryptanalysis. Integrals are particularly suited to the analysis of ciphers with mainly bijective components. Until now, most integrals that have been developed have probability one, and probabilistic integrals have not been examined in depth. Typically, integral attacks set up a path through the cipher where at any position in the path the collection of texts produces either a set of words which are all different (A), or all the same (C), or such that the sum of the words is S (where $S = 0$ is common), or indeterminate (?). One can then follow interacting paths of A, C, S, and ? through the cipher and predict the form of the set of words after as many rounds as possible. It is interesting to note that [371] successfully combines integral attacks with interpolation attacks, where one half of the cipher is covered by an integral, and the other half is approximated by a low-degree polynomial. In [88], Multiset attacks were used by Biryukov and Shamir to structurally cryptanalyse a block cipher. In this context, one does not exploit the weaknesses of a particular cipher, (such as bad differentials) but, instead, studies the security of cryptosystems described by generic block diagrams. Such attacks are therefore applicable to a large class of cryptosystems, and multiset strategies are particulary suited to such attacks.

2.2.3.16 χ^2 Attack

The χ^2 attack quantifies the statistical significance of certain input-output dependencies for a certain cipher approximation. It was probably first suggested in the context of a statistical cryptanalysis of DES by Vaudenay [611]. The idea has been developed as an extension of linear cryptanalysis with a more sensitive distinguisher at the output, where quantification is achieved by means of a χ^2 analysis [42, 359, 366]. If the block cipher is truly random then no set of plaintext-ciphertext pairs will produce any significant deviation from a random relationship between plaintext and ciphertext. Conversely, any deviation from random acts as a distinguisher that can then form the basis for an attack on the cipher.

2.2.3.17 Attacks Using Exact Systems of Multivariate Equations

Recent block cipher designs have discouraged the application of linear and differential cryptanalysis by deliberately designing against them. This has prompted cryptanalysts to look elsewhere for effective attack methods. One interesting new direction is also one of the most direct. Every component of a cipher can be described by means of a set of algebraic equations. These component descriptions can then be collected together to form a large system of equations which define the complete cipher. If this system of equations can be solved faster than exhaustive search then the cipher is broken. Clearly an arbitrary construction of the system will be impossible to solve as it will contain far too many variables and equations of too high a degree. However, by careful search one can find systems of low degree equations in relatively few variables. In particular, the critical component for many ciphers is the S-box, and careful search of state-of-the-art S-boxes has found that the S-box can be exactly represented by surprisingly few low-degree equations, and recent research has led to constructions of sparse systems of quadratic equations. Whereas, say, linear cryptanalysis uses approximations with relatively low probability, the equation system holds with probability one. These algebraic attacks differ in several respects from the standard statistical approaches to cryptanalysis. In particular, these new attacks require only a few known-plaintext queries and, also, the attack complexity does not seem to grow exponentially with the number of rounds of the cipher. It has further been found that well-chosen, sparse, overdefined systems are often easier to solve than critically defined systems. Cryptanalytic methods have been developed by using such systems of equations, where nonlinear terms are treated as independent linear variables in a process called relinearisation [566] (by Shamir and Kipnis — a development from linearisation), and Extended Linearisation (XL) by Courtois *et al.* [164]. These methods, and a variant of XL called Extended Sparse Linearisation (XSL) may perhaps lead to a successful attack on Rijndael (amongst others) — in theory at least, although this is by no means clear at the moment [165, 449]. The main strategy relating to these developing techniques appears to be to find as many equations in as few variables as possible, and of as low degree as possible [82]. The aim is to minimise the number of *free terms*, where a set of free terms is a set of monomials of any degree that are linearly independent. The number of free terms is given as the number of distinct terms minus the number of equations — a term in an equation can be of any degree, in particular for quadratic multivariate equation systems, a term is of degree one or two. Recently, Murphy and Robshaw [452] have embedded Rijndael in a large cipher called the Big Encryption System (BES), which expands Rijndael by its conjugates. The work of Murphy and Robshaw [452, 454] found a system of equations from BES that is simpler than the one developed by Courtois and Pieprzyk [165] for Rijndael, and this suggests that the technique of embedding a cipher in a larger cipher can sometimes lead to simpler equation systems. The paper [165] has also observed that, for the block ciphers they analysed, it is possible that the security of these ciphers does not grow exponentially with the number of rounds. However, it should be noted that some experts do not consider this to be possible.

2.2.3.18 Exploiting Relations Between the Bit-Functions of S-Boxes

A very recent paper by Fuller and Millan [245] has observed that all output bits of the Rijndael S-box, written as Boolean functions of the input bits, can be transformed to one another by means of an affine transformation of the input bits. In other words, let b_i and b_j be two distinct output functions of the Rijndael S-box. Then we can always find a Boolean matrix, $\mathbf{A}$, and a Boolean vector, $\mathbf{B}$, such that,

$$b_i(x) = b_j(\mathbf{A}x + \mathbf{B})$$

This surprising result means that the Rijndael block cipher only uses *one* S-box from eight bits to one bit (used 128 times in each round). At the time of writing this symmetry has not led to an attack on Rijndael but we include this observation in this section as it is possible that it may lead to attacks in the future. Note however that, if this symmetry does not lead to an attack, then it could in fact be beneficial, leading to more efficient software and hardware implementations of the cipher. Since [245] was posted, Youssef and Tavares [629] proved this result by making use of dual bases over GF(2^n) and trace functions, and generalised the observation to include all S-boxes that utilise bijective monomials. They also extended the result to show that all *coordinate* (bit) functions of the Rijndael round function are equivalent under affine transformation of the input to the round function. Further to this, Biham [62] has shown that such bit-affine relations exist for many of the S-boxes of the block ciphers submitted to NESSIE. In particular, for the S-box, S9, of MISTY1, the rotation of the input by any number of bits does not affect the least significant bit of the output. Tables which summarise these results can be found in Sect. 2.9. Also, Biryukov, De Cannière, Braeken, and Preneel [83] have developed an algorithm to solve the linear and affine equivalence problem for pairs of arbitrary S-boxes where, for $n \times n$ S-boxes, the affine algorithm hascomplexity $O(n^3 2^{2n})$, allowing a comparison of S-boxes up to about size 32×32. This algorithm uses an exponential amplification of the guesses by exploiting the linearity of the potential affine mapping. The algorithm has a certain similarity to the ’to-and-fro’ algorithm of Patarin, Goubin, and Courtois [516] used to solve polynomial isomorphisms. The affine equivalence algorithm of [83] can also be used to decompose an S-box in terms of SPNs that use layers of smaller S-boxes and, for an 8×8 S-box it is found that, typically, at least 20 layers of 4×4 S-box SPNs are necessary.

2.2.3.19 Exploiting the Permutation Cycle Structure of a Cipher

This cannot yet be considered an attack technique on a cipher. However, it suggests that one may be able to use the cycle structure of the permutation that defines part or all of the cipher to distinguish the cipher from a cipher chosen at random. In [81] Biryukov and Preneel show that such a technique is particularly suited to ciphers which utilise many involutional elements. The observation was motivated by the submission to NESSIE of Khazad. Khazad is built entirely from involutions which means that one can choose to replace a constituent permutation by its inverse, so allowing the conjugation of some permutations, where conjugation preserves isomorphism of permutations and, therefore, the cycle structure of a permutation. This conjugation can be used to determine sections of the cipher over which the permutation cycle structure is invariant. It follows that a significant number of rounds of the cipher can take on the same cycle structure as the linearly keyed S-box, and, in the future, it may be possible to use this fact to distinguish the cipher from a random cipher.

2.2.3.20 Side-Channel Attacks

Side-channel attacks are a major thread for all implementations of cryptographic algorithms. Annex A is devoted to this subject and addresses the application of side-channel attacks and their countermeasures for block ciphers. Summarizing annex A and [509], algorithms such as Rijndael, Khazad and Camellia seem to be the most suited for implementations resistant to side-channel attacks.

2.2.4 Assessment process

The block cipher submissions were assessed with reference to the above generic common block cipher attacks. Clearly, although some of these attacks are universal, some of them have more relevance to certain block ciphers than others. Thus the block ciphers were assessed against the attacks that seemed most relevant. Furthermore, some block ciphers were analysed using techniques specific to that primitive.

Local statistical testing was applied to components of the block ciphers in order to demonstrate that they have good statistical properties. Global statistical testing was applied to the input-output of the block cipher submissions to show that the data demonstrated good statistical properties. In addition, these tests were also applied on reduced round versions of the block ciphers in order to determine the number of rounds needed to achieve good statistical properties. Further details can be found in [479]. None of the block ciphers tested exhibited any anomalous behaviour. We now summarise the two toolboxes developed by NESSIE.

2.2.4.1 The NESSIE statistical toolbox for block ciphers

This toolbox is part of the general NESSIE test suite for the evaluation of statistical properties of the submissions. We summarise the available tests as follows:

NESSIE Stream Cipher Tests. The block cipher can be used in OFB Mode or Counter Mode. In such cases it produces a stream output and can be viewed as a stream cipher. In these modes it can therefore be tested using the NESSIE stream cipher tests. In both modes the following tests are applied:

– Frequency Test. Splits up the bit sequence into disjoint m-tuples whose distributions are then evaluated statistically.
– Collision Test. The collision test splits up the bit sequence into blocks of a fixed size. A collision occurs if the same block appears more than once. The test statistically evaluates the number of collisions.
– Overlapping m-tuple Test. Shifted windows of m-tuples of words of fixed wordlength are examined and statistically tested along with cyclic shifts of the original sequence.
– Gap Test. Splits up the bit sequence into disjoint m-tuples which are then interpreted as binary integer representations. The length of gaps, where the numbers are not within a numerical range given as a parameter of the test, are evaluated statistically. This test is also applied to cyclic shifts of the original sequence.
– Run Test. Splits up the bit sequence into disjoint m-tuples which are then interpreted as binary integer representations. The lengths of subsequences of consecutive, strictly increasing numbers are evaluated statistically.
– Coupon Collector's Test. Splits up the bit sequence into disjoint m-tuples. The number of subsequence m-tuples it takes until all possible 2^m m-tuples have appeared, is evaluated statistically. The test is also applied to cyclic shifts of the original sequence.
– Universal Maurer Test. Splits up the bit sequence into disjoint m-tuples and evaluates statistically how many m-tuples later an m-tuple re-appears in the sequence. The result of this test is closely related to the entropy of the bit sequence.

- Poker Test. Splits up the bit sequence into disjoint m-tuples, and this sequence of m-tuples is then split up into subsequent disjoint k-tuples of m-tuples. The poker test statistically evaluates how many of the m-tuples in a k-tuple are equal. The test is also applied to cyclic shifts of the original sequence.
- Fast Spectral Test. Applies the fast Walsh transform to the given sequence and uses the spectral results to assess the randomness of the sequence.
- Correlation Test. Determines in how many places the original sequence and the sequence shifted by n bits have the same value for shifts up to the length of the original sequence.
- Rank Test. Sequence bits are used to fill square matrices and the rank of the matrices over GF(2) is evaluated statistically.
- Linear Complexity Test. The Berlekamp Massey algorithm is used to determine the length of the shortest linear feedback shift register which can produce the given bit sequence. For the linear complexity profile, this is done for the first 1,2,3.. bits of the sequence.
- Maximum Order Complexity (MOC) Test. Determines the length of the shortest possibly non-linear feedback shift register which can produce the given bit sequence For the MOC profile, this is done for the first 1,2,3.. bits of the sequence.
- Ziv Lempel Complexity Test. Measures how well a bit sequence can be reconstructed from earlier parts of the bit sequence.
- Dyadic Complexity Test. The Dyadic Complexity Test is an implementation of the complexity measure suggested by Goresky and Klapper [348] for sequences of bits. It determine the length of the shortest feedback shift register with carry which can produce the given bit sequence.
- The Percolation Test is the simulation of a forest fire. The bit sequence to be tested determines where trees are standing in the simulated forest. The test evaluates statistically how fast a fire propagates in the simulated forest.
- Constant Runs Test. For the constant runs test, the sequence of bits is subdivided into runs, that is maximal disjoint subsequences of consecutive 0s and 1s. The frequencies of these runs of the various lengths are evaluated statistically.

These stream cipher tests were applied to the block cipher submissions, and none of the block ciphers exhibited any anomalous behaviour. The detailed results of the statistical tests are available as NESSIE public reports.

NESSIE Block Cipher Tests. NESSIE has also developed a test suite to test the block ciphers as block ciphers. Details can be found in [479,565]. The following tests are applied:

- Dependence Test. Evaluates the dependence matrix and the distance matrix of the cipher. The degree of completeness, the avalanche effect, and the Strict Avalanche Criterion (SAC) are also computed. We now define these criteria:
  - A function is complete if each output bit depends on each input bit.
  - The avalanche effect occurs when, on average, approximately one half of the output bits change whenever a single input bit is complemented.
  - The SAC is satisfied if each output bit changes with probability one half whenever a single bit is complemented.

– Linear Factors Test. Used to determine whether there are any linear combinations of output bits which, for all keys and plaintexts, are independent of one or more key or plaintext bits. Such a combination is called a linear factor. It is impossible to do this for all keys and plaintexts, so the test is only applied to a sufficiently large number of pairs of random keys and plaintexts.

The above block cipher tests were applied to the full cipher, and then to reduced-round versions to determine the minimum number of rounds for which the various criteria were all satisfied.

These block cipher tests were applied to the block cipher submissions, and none of the block ciphers exhibited any anomalous behaviour. The detailed results of the statistical tests are available as NESSIE public reports.

2.2.4.2 The NESSIE toolbox for differential and linear cryptanalysis

NESSIE has developed a software toolbox to aid in the analysis of block ciphers. This is described in detail in [57]. The system currently examines the block cipher on three levels.

1. Statistical properties of the basic building blocks are found.
2. Properties of complete rounds are found.
3. Properties are found over several rounds.

It is intended to extend the system to include aspects of cryptanalysis of the full cipher at a later date. We now describe the analysis for each of the above three categories.

Building Blocks. Any re-arrangement mapping and any nonlinear S-box mapping between an input vector of bits and an output vector of bits can be analysed, where the input and output vectors need not be of the same size. The mapping can be validated for one-to-one mapping, tested for the period of inherent permutation cycles and the existence or otherwise of fixed-points in any inherent permutations, and checked for output balance/distribution.

The system also provides difference distribution tables and linear approximation tables for the inherent S-boxes, for use in differential and linear cryptanalysis, respectively.

One Round. The system identifies dependencies between the output bits of round i and the output bits of round $i+1$, and provides an example for linear and differential one-round characteristics and their probability.

Several Rounds. The system identifies the most important 3-round characteristics, and suggestions for good 5-round characteristics, with corresponding probabilities.

2.3 Overview of the common designs

Modern-day ciphers are designed to withstand known cryptanalytic attacks. However there is not complete agreement as to the best design philosophy to use, hence the variety of designs submitted to NESSIE. We now summarise some of the most important of these design philosophies, where any given cipher may utilise one or more of these philosophies.

2.3.1 Feistel ciphers

A basic Feistel cipher takes $2t$ plaintext bits, and is a permutation, F, which uses m round permutations, F_i, so that,

$$F = F_0 \circ F_1 \circ \ldots \circ F_{m-1}$$

where '$\circ$' means composition.

Round i acts on half the input bits, the t bits, R, by means of the keyed function, f_i, and XORs the result with the other half of the bits, the t bits, L. It then swaps the left and right halves. Thus we have,

$$[L', R'] = F_i[L, R] = [R, L \oplus f_i(R)]$$

where $[L'\, R']$ becomes the new input $[L\, R]$ to round $i + 1$. Although F and the F_i must be permutations, the f_i need not be. It takes two rounds before all plaintext bits have been acted on in a nonlinear way. DES, MISTY1, and Camellia are all essentially Feistel ciphers. The Feistel structure allows decryption to be accomplished using the same process, but with the sub-keys used in reverse order. Sometimes the Feistel structure may be nested such that individual components also have a Feistel structure. This is the case for MISTY1.

2.3.2 Substitution-Permutation Networks (SPNs)

A substitution-permutation network (SPN) separates the role of confusion (substitution) and diffusion (permutation) in the cipher. As with most ciphers, the cipher is decomposed into iterative rounds where each round comprises a layer of S-boxes, followed by a permutation/diffusion layer. The S-box layer provides the nonlinearity (confusion), and the permutation layer provides the rapid diffusion. In fact, the SPN has more recently come to include ciphers where the diffusion layer is not a permutation (re-wiring) but is a linear or affine transformation. Moreover these linear transformations are often derived from Maximum-Distance-Separable (MDS) error-correcting codes, where the high minimum distance of the code implies a high diffusion rate. The separating of the tasks of confusion and diffusion allows the designer to maximise nonlinearity for the S-box, and maximise information spread for the diffusion layer. Rijndael, KHAZAD, Hierocrypt, and SAFER++are all examples of SPNs. In fact a Feistel cipher is also a type of SPN. Sometimes the SPN structure may be nested, such that the individual S-boxes are themselves SPNs. Hierocrypt is an example of such a cipher, and Rijndael can also be described in this way [41]. In fact, one can always compose a large S-box as a number of layers of smaller S-boxes — and this is the case for the S-boxes of KHAZAD and ANUBIS.

2.3.3 Resistance against differential and linear cryptanalysis

Many recent block cipher designs have attempted to maximise the resistance of the cipher to differential and linear attack. This is usually achieved in two ways:

- Firstly, the elemental S-boxes are designed to be highly nonlinear, i.e. so that they can only be poorly approximated by linear equations, and so that differential characteristics can only be propagated through the S-box with small probability. These design criteria are evident in the parameters of low linear bias and low differential probability associated with the cipher.
- Secondly, the diffusion components are designed to spread the information to the whole cipher block as rapidly as possible. This rapidity is sometimes measured in terms of the branch number or minimum distance of the diffusion layer. The resilience of the component functions of the S-box is also a measure of the diffusion properties of the S-box, where Resilience t indicates that the component function is completely independent of any t or fewer of its constituent variables. The result of rapid diffusion is that, in any linear or differential attack, which essentially uses elemental approximations, too many of the elements of the block cipher must be approximated, thereby rendering the attack useless.

For some recent ciphers, e.g. Khazad, the emphasis on extremely rapid diffusion has reduced the perceived need for optimal nonlinearity of the S-box — reasonably good nonlinearity is considered good enough. This has enabled the S-box to be chosen randomly from a suitable subspace of the set of S-boxes, unlike, for instance, the S-boxes of Rijndael, Camellia, or MISTY1, which optimise nonlinearity at the price of, in the case of Rijndael and Camellia, a non-random, algebraic S-box which may lay itself open to algebraic attack.

2.3.4 Mini-ciphers and reduced rounds

Although the attacks on block ciphers described in this chapter provide a means of assessing the security of a cipher, it is still the case that cryptanalysis of a real cipher is an extremely big task. As an aid to analysis, some cipher designs naturally allow the description of mini-versions of their cipher over a reduced wordsize and reduced plaintext and key blocksize. One can then analyse these small versions of the cipher and extrapolate the conclusions to the larger real cipher. Ciphers which naturally support mini-versions include RC6, SAFER++, IDEA, Khazad, Rijndael, and numerous others. In fact, most ciphers already incorporate this philosophy in the sense that they decompose the cipher into iterated round functions. This leads to attacks on reduced rounds which can then be extrapolated to the full-round cipher. Recently, [357] has suggested a *very* rough *rule-of-thumb* computation for the minimum number of rounds, R, required to make a block cipher secure, given by $R \geq \frac{dN}{w}$ where w is the minimum word size input to a confusion stage in the cipher (for instance this could be the size of the smallest S-box used), d is the maximum number of rounds it takes for one word to be input to a confusion stage (for a Feistel cipher, $d = 2$), and N is the size of the plaintext input to the cipher, in bits.

2.3.5 Simple as opposed to complicated designs

Some ciphers are deliberately very difficult to analyse. Others are deliberately relatively simple to analyse, being built out of conceptually simple primitives. For example, ciphers that add in the key via XOR are often easier to analyse than ciphers that use nonlinear key input. The security of ciphers that use nonlinear keying can often be far more key-dependent than the security of ciphers that use linear keying. This makes them harder to analyse but also makes it more likely that there are *weak* keys for which the enciphering function is weak [453]. The current trend is towards ciphers that are simple to describe and analyse, as a cipher that cannot be analysed cannot be declared *secure*. Of course there are different notions of simplicity, for instance, although RC6 and KHAZAD are both *simple* designs, they are not simple in the same way.

2.3.6 A separate key-schedule

Virtually all modern block cipher designs separate the key-schedule from the enciphering/deciphering process. Typically the cipher will take in a master key and, from this key, generate round keys which will then be used to key the rounds of the encryption process. Thus there are usually two parallel processes in a block cipher. One encrypts, whilst the other generates the key-schedule.

2.3.7 The use or otherwise of S-boxes

The use of S-boxes in block ciphers is mainly as a means of introducing nonlinearity. They are usually envisaged as large look-up tables, substituting one value for another. Examples of the type of cipher that uses S-boxes are DES, MISTY1, Rijndael, Camellia. In contrast, some ciphers do not use explicit S-boxes but introduce nonlinearity by means of well-known algebraic operations such as integer addition, integer multiplication, log, or exponentiation. Examples of these ciphers are RC6, IDEA, and SAFER++. However, this distinction is a bit tenuous as, for example, a log function or a multiplication can be viewed as an S-box, and the S-box used by Rijndael and Camellia is essentially x^{-1} over GF(2^8). A more significant distinction is between key-dependent and non-key-dependent S-boxes — for instance, whereas multiplication by a constant can be implemented using a fixed look-up table, multiplication by a key cannot. Another useful design distinction is that some ciphers use large S-boxes (e.g. 8 by 8 or bigger), and some ciphers use smaller S-boxes (e.g. 4 by 4). It has recently been argued that the use of S-boxes that are too small is not a good idea [165].

2.3.8 Ciphers which are developed from well-studied precursors

It is an accepted fact that block ciphers need many years of analysis before they can be labelled *secure*. It is therefore common for block cipher designers to incrementally improve their own previous designs and/or incorporate aspects of previous block cipher designs. This is a cautious strategy which seeks to add to the perceived security of the new design. For instance, an attack on the key-schedule of a previous version of SAFER [351, 356], and an improved diffusion layer, have led to upgrades which have been incorporated in SAFER++. (Note, however, that it could be argued that the designers have made drastic changes in designing SAFER++from previous designs which, to some extent, invalidate previous analysis — for instance they completely changed and reduced the complexity of the mixing layers). As another example, RC6 uses exactly the same key schedule as RC5.

2.3.9 Making encryption and decryption identical

It is clearly desirable for the encryption and decryption processes of a cipher to be identical in as many ways as possible, as this enables the re-use of hardware or software resources. Feistel ciphers are designed in this way. Moreover, any cipher component which is an involution will be, by definition, the same in reverse. Khazad uses only involutions so encryption and decryption are identical for such a cipher (apart from the order of the sub-keys). In contrast, Rijndael uses different encryption and decryption operations, although the encryption and decryption speeds for Rijndael are virtually identical in software, differing only slightly for 8-bit machines. In hardware, the speed of both operations for Rijndael is the same, decryption requiring just a little bit more hardware.

2.3.10 Hash functions as block ciphers

A hash function is usually not intended to be used as a block cipher, but it can be. The hash function will take a K-bit message and hash an initial N-bit value to a final N-bit value which is called the hash of the message. The hash function is designed so that it is impossible to ascertain the K-bit message from the N-bit hash. K is always bigger than N. If we redefine the K-bit message as a K-bit key, and the N-bit initial value as the input plaintext, then the output hash becomes the N-bit output ciphertext. For example, SHACAL-1 and SHACAL-2 are block cipher submissions to NESSIE and are block ciphers built from the hash functions, SHA-1 and SHA-256, respectively. A secure hash function should ideally minimise the number of (inevitable) N-bit output hash collisions for different initial K-bit messages, given that the initial N-bit value is fixed. This is called collision-resistance. Transferring this criterion to the block cipher regime implies that, given a fixed input plaintext, a secure block cipher should seek to minimise the number of ciphertext collisions for different key inputs. However, it is not at all clear at present whether weaknesses in collision-resistance for the hash function translate into attacks on the associated block cipher. It is an open problem.

2.3.11 Current standards

The Data Encryption Standard, DES, is now considered to have too small a key space for today's security requirements. Triple-DES (DES encryption, followed by DES decryption, followed by DES encryption) uses a $3 \times 56 = 168$-bit key and is considered to be a practical solution to the security weakness of DES. Triple-DES has been included in the NESSIE evaluation as a benchmark cipher with which to compare the other block cipher submissions, although it is somewhat inefficient in comparison.

The new Advanced Encryption Standard, AES [470], also known as Rijndael, has also been included in the NESSIE evaluation, both Rijndael-128 and Rijndael-256, once again to act as a benchmark with which to compare the other block cipher submissions. Note also that RC6, one of the five AES finalists, has also been submitted to NESSIE, as has SAFER++which is a modified version of SAFER+ which was in the AES.

Currently there is an ongoing assessment of new cryptographic primitives to be adopted as the new ISO standard. The block ciphers being considered are as follows: For 64-bit plaintext, TDEA (Triple-DES) — 128 or 192-bit key, MISTY1 — 128-bit key, and Khazad — 128-bit key. For 128-bit plaintext, the AES (Rijndael) — 128, 192, or 256-bit key, Camellia — 128, 192, or 256-bit key, SEED — 128-bit key, RC6 — 16 - 256-byte key, and CAST-128.

2.3.12 Block cipher primitives

The deadline for submissions to the NESSIE project was September 29, 2000, just before NIST's announcement that the AES block cipher was to be Rijndael. Nevertheless, there were 17 block ciphers submitted to NESSIE.

The NESSIE call for primitives specified the following security levels for block ciphers.

– *High.* Key length of at least 256 bits. Block length at least 128 bits.
– *Normal.* Key length of at least 128 bits. Block length at least 128 bits.
– *Normal-Legacy.* Key length of at least 128 bits. Block length 64 bits.

There have been two phases to the evaluation process, Phase I and Phase II. In total, 8 ciphers were selected for Phase II, with SAFER++and RC6 both being selected for two security levels.

The submissions are classified in the table below according to the security levels specified in the NESSIE call, and those selected for Phase II of NESSIE are indicated.

Name of cipher	Block Size (bits)	High	Normal	Normal-Legacy	Phase II
CS-Cipher	64			X	
Hierocrypt-L1	64			X	
IDEA	64			X	√
Khazad	64			X	√
MISTY1	64			X	√
Nimbus	64			X	
NUSH	64 128 256	X	X	X	
SAFER++	64 128	X	X	X	√ √
Grand Cru	128		X		
Noekeon	128		X		
Hierocrypt-3	128	X	X		
Q	128	X	X		
RC6	128	X	X		√ √
SC2000	128	X	X		
Anubis	128	X			
Camellia	128	X			√
SHACAL-1	160		X		√
SHACAL-2	256		X		√

NUSH was designed with three different block sizes: 64 bits, 128 bits, and 256 bits.
RC6 has a variable block length $4w$ bits, where $w \geq 32$ is recommended by the designers.
SAFER++has two variants, one with 64-bit blocks and one with 128-bit blocks.

2.4 64-bit block ciphers considered during Phase II

The 64-bit block ciphers selected for Phase II of NESSIE were IDEA, Khazad, MISTY1, SAFER++ (64-bit block), and Triple-DES. None of these ciphers has been broken so the following security evaluation identifies weaknesses that occur in reduced-round versions of the ciphers, and identifies weaknesses that may lead to more effective attacks in the future. We first describe each cipher in some detail, along with the most important attacks known on each cipher. Note that the algorithms given here are not complete specifications, but references are given to complete specifications which may be found on the NESSIE website. After discussing each cipher we summarise and compare some of the distinguishing features of the ciphers, identifying potential weaknesses and noting the best-known attacks, as shown in Tables 2.14 and 2.15 of Sect. 2.9.1.

2.4.1 IDEA

2.4.1.1 The design

IDEA [387] operates on 64-bit blocks of plaintext and ciphertext and is controlled by a 128-bit key. It is specified to require 8.5 encryption rounds to achieve an acceptable security margin. It claims to achieve high security by concatenated use of three arithmetic operations from two dissimilar algebraic groups (two of the groups are isomorphic), namely:

– Addition mod 2^{16}.
– Multiplication mod $2^{16} + 1$ (where the all 0 bit word is interpreted as 2^{16}).

– Bitwise exclusive OR.

Note that (since $2^{16} + 1$ is prime) the multiplication operation is isomorphic to the addition operation. The combined use of these operations is used to achieve high nonlinearity and completely replace the more conventional use of S-boxes for this task. This can often result in relatively efficient implementations, in software because many processors have special-purpose multiplication operators, and in certain hardware scenarios where memory is at a premium, as S-boxes usually require large look-up tables. Although the software realisation of modular multiplication can be optimised [63], hardware implementation of multiplication will always cost many gates, and this means that hardware implementations of IDEA are not compact. The cipher is designed so that encryption and decryption are identical apart from key input. The encryption operation is shown in Fig. 2.3, and comprises 8 identical steps (rounds) followed by an output transformation. Round 1 is shown in detail in Fig. 2.3.

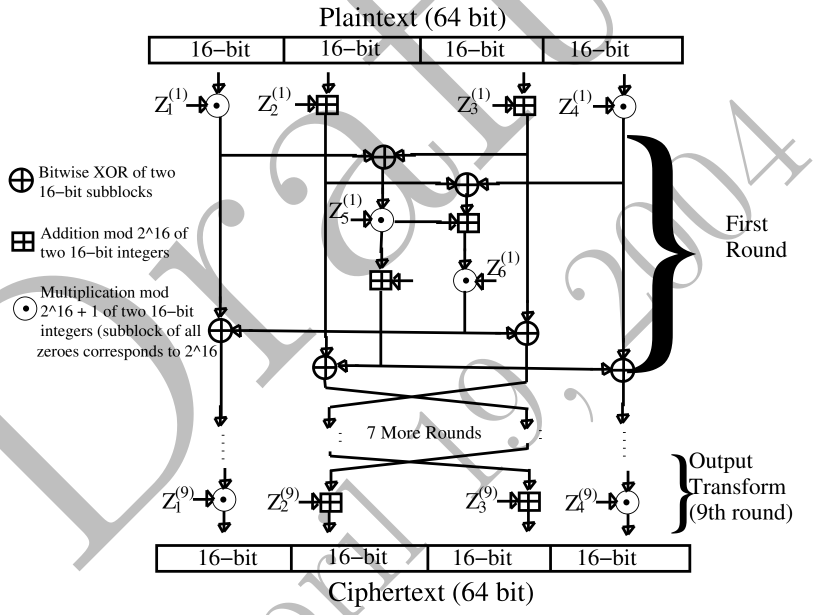


Fig. 2.3. The IDEA Block Cipher

The 64 bits of plaintext are partitioned into four 16-bit subblocks and all encryption operations take 16-bit inputs and produce 16-bit outputs. The Z's of Fig. 2.3 refer to six 16-bit key subblocks per round (with four 16-bit key subblocks for the subsequent output transformation). A total of $8 \cdot 6 + 4 = 52$ different 16-bit subblocks are generated from the 128-bit key. We describe this key schedule below.

It should be noted from Fig. 2.3 that after each round the 16-bit partitions are re-ordered before commencing the next round. The process of round one is repeated for 7 more rounds, and after 8 rounds the output transformation is combined with the final four of the key subblocks, as shown in Fig. 2.3.

A fundamental design criterion is the following:
At no point during the encryption process is the same algebraic group operation used contiguously.

It has been shown by Wernsdorf [619] that the multiply-addition box at the centre of the round of IDEA generates the alternating group on $\{0,1\}^{32}$ and [619] conjectures that the alternating group is also generated by the complete round of IDEA. These large groups exclude the possibility of several types of regularity for IDEA.

The key schedule takes a 128-bit key and turns it into 52 16-bit key subblocks, as follows:

- First, the 128-bit key is partitioned into eight 16-bit subblocks and these form the first eight key subblocks.
- The 128-bit key is then cyclically shifted to the left by 25 positions, after which the resulting 128-bit block is once again partitioned into eight 16-bit subblocks and these form the next eight key subblocks.
- The above process is then repeated until all required 52 16-bit key subblocks have been generated.

Decryption is identical to encryption apart from the use of different key subblocks, where these 52 16-bit key subblocks are as follows: each subblock used for decryption is the inverse of the key subblock that was used for encryption, where the inverse is taken with respect to the algebraic group operation associated with that particular key subblock. Moreover, these key subblocks must be applied in reverse order to the order in which they were used for encryption.

2.4.1.2 Security analysis

IDEA has been widely studied for over a decade and few security flaws have been found. In fact, no attack against 5 or more of its 8.5 rounds has been found. However, IDEA does exhibit weak key classes of substantial size, and it can be argued that a flawless cipher should have no weak keys at all. IDEA has been included in the popular cryptographic package, PGP, although not for some time, and is one of the best known and most widely used ciphers. The following attacks are some of those that have been applied to IDEA prior to and during NESSIE. Meier proposed a differential attack on 2.5 rounds [437], where it was observed that, if $a+b<2^{16}$, then $a+b \mod 2^{16} = a+b \mod 2^{16}+1$, and this in turn implies that that multiply-addition part of IDEA is linear about $\frac{1}{4}$ of the time. The paper of Borst *et al.* [113] developed a truncated differential attack on 3.5 rounds, and a miss-in-the-middle attack on 4.5 rounds was proposed by Biham *et al.* [66]. Moreover, it has been shown [86, 176, 288, 383] that certain weak-key classes exist for IDEA. For instance, [176] exploited the observation that $-x \mod 2^{16}+1 = x \oplus 11\ldots101$ whenever x_1, the second least significant bit of x, is 1. In fact, attacks which exploit potential weaknesses in the modular multiplication, mod $2^{16}+1$, are of particular interest for IDEA, for instance, Harpes *et al.* [283] attack IDEA by applying a quadratic residue homomorphism between $(\mathbb{Z}/n\mathbb{Z})^*$ and $\mathbb{Z}/2\mathbb{Z}$, where $n = 2^{16} + 1$, and a recent paper by Borisov *et al.* [111] further exploits homomorphisms between modular multiplication and XOR to identify large weak key classes for certain variants of IDEA where addition has been replaced with XOR. They show that for 2^{112} of the keys there exists a multiplicative differential characteristic over 8 rounds that holds with probability 2^{-32}. Recently, Raddum [538] has demonstrated a better attack on this variant of IDEA, called IDEA-X, using a similar technique, but this time using XOR-differentials. The advantages of the attack in [538] are that it works for all keys, and it is only necessary to change the first two additions in each round to XORs. In [538] a differential characteristic has been found that holds with probability 2^{-30} over 8 rounds and exploits the fact that $Z_{2^{16}}$ and $\mathrm{GF}(2^{16}+1)^*$ are both cyclic groups, and therefore isomorphic. Essentially, each element a in $\mathrm{GF}(2^{16}+1)^*$ can be written uniquely as,

$$a = g_{15}^{x_{15}} \cdot g_{14}^{x_{14}} \cdot \ldots g_0^{x_{15}}$$

where $\cdot$ means field multiplication, $x_i \in \{0, 1\}$, g_0 is a primitive element of $\mathrm{GF}(2^{16} + 1)$, and $g_i = g_{i-1}^2$. We write this as $a = \mathbf{g}^{\mathbf{x}}$. Then $\phi(a) = \mathbf{x}$ is an isomorphism, mapping multiplication, mod $2^{16} + 1$, for a to addition, mod 2^{16}, for $\mathbf{x}$. For $a, b \in$ $\mathrm{GF}(2^{16} + 1)$, $ab = \phi^{-1}(\phi(a) + \phi(b))$. The above isomorphism is then used to pass an input XOR difference through the mod $2^{16} + 1$ multiplier, then unchanged through a subsequent (integer) additive key bit, and finally back to an XOR difference. The complete XOR to XOR difference holds with probability $\frac{1}{4}$. This characteristic is used to attack IDEA-X as defined by [111]. However, the attack of [538] in fact requires fewer of the integer additions to be changed to XORs so the cipher it attacks is closer to the real IDEA than the cipher of [111].

Until the commencement of NESSIE the best attack on IDEA was by Biham *et al.* [66] on 4.5 out of 8.5 rounds of IDEA.

Nakahara *et al.* [457] report on variants of the SQUARE attack applied to reduced-round versions of the PES and IDEA block ciphers (PES is a forerunner of IDEA). Attacks on 2.5 rounds of IDEA require $3 \cdot 2^{16}$ chosen-plaintexts and recover 78 key bits. A SQUARE related-key attack is applied on 2.5 rounds of IDEA and recovers 32 key bits, with 2 chosen-plaintexts and 2^{17} related keys. Implementations of the attacks on 32-bit block mini-versions of both ciphers confirmed the expected computational complexity. Although the attacks do not improve on previous approaches, this report shows new variants of the SQUARE attack on word-oriented block ciphers like IDEA and PES.

Biryukov *et al.* [86] present a large collection of new weak-key classes on the full 8.5 round IDEA cipher proposed, using the property that some multiplicative keys which are 0 or 1 turn modular multiplication into a linear operation. The weak-key classes in this paper contain 2^{53} - 2^{64} weak keys. The novelty of the approach is in the use of the boomerang distinguisher for the weak-key class membership test, as developed by Wagner [616]. Large weak-key classes for reduced-round versions of IDEA are also shown. It appears that the existence of relatively large weak key classes for IDEA is due to the use of modular multiplication as the main nonlinear part of the cipher, and because the key schedule is also a linear process. The results suggest a redesign of the key schedule of IDEA.

Finally, statistical evaluation of IDEA [537], following the recommendations of the NESSIE statistical evaluation process for block cipher submissions, does not indicate a deviation from random behaviour.

Table 2.1. Comparison of attack requirements on reduced-round IDEA

Attack Type	Year	Reference	#Attacked Rounds	Data † Complexity	Time Complexity
Differential	1993	[437]	2	2^{10}	2^{42}
Improved-Square	2002	[191]	2	23	2^{64}
Differential	1993	[175]	2.5	2^{10}	2^{32}
Differential	1993	[437]	2.5	2^{10}	2^{106}
Square attack	2000	[457]	2.5	$3 \cdot 2^{16}$	2^{58}
Square attack	2000	[457]	2.5	2^{32}	2^{59}
Square	2000	[457]	2.5	2^{48}	2^{79}
Related-Key Square	2001	[457]	2.5	2	2^{33}
Improved-Square	2002	[191]	2.5	55	2^{81}
Differential-Linear	1996	[113]	3	2^{29}	$0.75 \cdot 2^{44}$
Improved-Square	2002	[191]	3	71	2^{71}
Improved-Square	2002	[191]	3	2^{33}	2^{82}
Truncated Differential	1997	[369]	3.5	2^{56}	2^{67}
Miss-in-the-middle	1999	[66]	3.5	$2^{38.5}$	2^{53}
Improved-Square	2002	[191]	3.5	103	2^{103}
Improved-Square	2002	[191]	3.5	2^{34}	2^{82}
Miss-in-the-middle	1999	[66]	4	2^{38}	2^{70}
Related-Key ‡ Differential-Linear	1998	[288]	4	38.3	38
Improved-Square	2002	[191]	4	2^{34}	2^{114}
Miss-in-the-Middle	1999	[66]	4.5	2^{64}	2^{112}

† number of chosen texts.
‡ the differential-linear attack requires two related keys.

In view of the comparatively large amount of cryptanalysis undertaken on IDEA, Tables 2.1 and 2.2 summarise many of the known chosen-plaintext and weak-key attacks on IDEA, respectively. The most effective of these attacks are then included in the summary table, Table 2.14, in Sect. 2.9.1 at the end of this chapter.

2.4.2 Khazad

2.4.2.1 The design

Khazad [39] is a 64-bit Substitution-Permutation (SPN) block cipher with a 128-bit key and is designed to operate over 8 rounds. Khazad has many similarities to Rijndael, but works on 64-bit plaintext blocks. A primary *selling-point* of Khazad is that **all** components of the algorithm use involutions. This involutional structure is important for implementations and to make encryption and

Table 2.2. Comparison of attack requirements on IDEA for weak-key classes.

Attack Type	Year	Reference	#Attacked Rounds	**Weak-key Class Size**	Data † Complexity	Time Complexity
Differential	1993	[176]	8.5	2^{51}	2	2^{12}
Differential-Linear	1998	[288]	8.5	2^{63}	20	4
Boomerang	2002	[86]	4.5	2^{101}	2^{18}	2^{18}
Boomerang	2002	[86]	5	2^{97}	2^{10}	2^{10}
Boomerang	2002	[86]	5	2^{95}	4	4
Boomerang	2002	[86]	8.5	2^{53}	4	4
Boomerang	2002	[86]	8.5	2^{57}	2^{11}	2^{11}
Boomerang	2002	[86]	8.5	2^{64}	2^{16}	2^{16}

† number of chosen texts.

decryption equivalent operations apart from the order of the key schedule. This also implies equal security for encryption and decryption. These involutions include the S-box, S, (i.e. $S[S[x]] = x$), and the 64-bit linear diffusion mapping, based on an MDS code and represented by a matrix H, (i.e. $H^2 = I$). The matrix H that realises this diffusion is also chosen to have lowest possible Hamming Weight and, for smart-card implementation, lowest possible integer weight over 8-bit words. The cipher also emphasises the Wide-Trail Design Strategy [174] as the linear diffusion layer is based on a Maximum-Distance-Separable (MDS) code with a high branch number of 9. This ensures that there is full mixing after a single round. This, in combination with a reasonably nonlinear S-box ensures strong resistance to linear and differential attacks, and to related-key attacks. In the original submission the 8 × 8-bit S-box was randomly chosen to avoid any obvious internal structure. However, this made the S-box costly to implement in hardware so the authors have modified the submission by replacing this S-box with another S-box which is more amenable to hardware implementation. This replacement was also made in order to correct a small security flaw where the maximum bias of a linear characteristic through the S-box was slightly underestimated (this did not lead to any security failure, as the high diffusion of the round more than compensates for this). The S-box has also been chosen to have no fixed point and is made out of six 4 × 4-bit smaller S-boxes, P and Q, which are involutions with optimal nonlinearity characteristics, as shown in Fig. 2.4. The designers expect that the hardware required to implement this S-box should be around $\frac{1}{5}$ of that for the Rijndael S-box, which is also an 8 × 8-bit S-box. Essentially, this potential reduction in implementation cost is bought at the price of slightly weaker nonlinear characteristics. However it should be noted that, unlike in Camellia or Rijndael, the avoidance of an S-box using x^{-1} *may* increase the minimal size of any potential set of sparse quadratic equations on which to base an attack of the form proposed by Courtois and Pieprzyk in [165] or by Murphy and Robshaw in [452]. Preliminary evidence for this claim is given by Biryukov and de Cannière [82], where a description of the 8 × 8 Khazad S-box requires 28 quadratic equations, but a description of the 8 × 8 Rijndael S-box only requires 23 quadratic equations

Key addition is achieved via XOR (which is also an involution). This has the advantage that no weak keys exist as the nonlinearity operations are key-

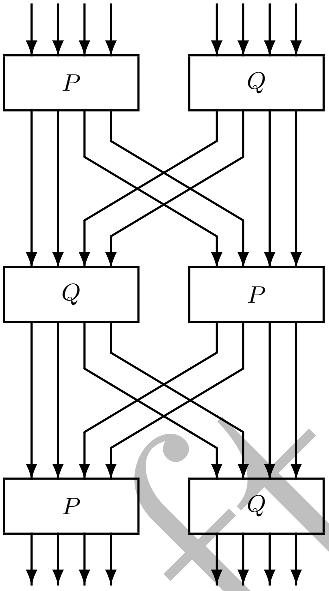


Fig. 2.4. Structure of the Khazad S-box. Both P and Q are pseudo-randomly generated involutions; the output from the upper and middle nonlinear layers are mixed through a simple linear shuffling.

independent. The Khazad key schedule uses a 9-round 128-bit Feistel scheme with an internal F-function being a round of Khazad with constants used as the 64-bit key. The user-specified 128-bit key is used as a plaintext and the intermediate 64-bit values after each round become the subkeys of Khazad. The key schedule expands the 128-bit key into 64-bit round keys, $K^0, K^1, \ldots, K^R$, plus two initial round keys K^{-2} and K^{-1}, and these round keys are generated as follows:

$$K^r = \rho[c^r](K^{r-1}) \oplus K^{r-2} \quad 0 \leq r \leq R$$

where $\rho[c^r](K)$ is the round function of Khazad which takes as input parameters the pre-chosen constant c^r and round key input, K.

Running R rounds of the complete cipher apply the operation $\alpha_R[K^0 \ \ldots \ K^R]$ to the plaintext, where,

$$\alpha_R[K^0 \ \ldots \ K^R] = \sigma[K^R] \circ \gamma \circ (\bigcirc_1^{r=R-1} \rho[K^r]) \circ \sigma[K^0]$$

where σ, γ, and ρ are the operations of key addition, nonlinear substitution and the full round function, respectively.

2.4.2.2 Security analysis

The designers [39] state that, for Khazad, there is no 2-round differential characteristic with probability higher than 2^{-45} and no 2-round linear approximation with bias of more than approximately $2^{-20.7}$. Related-key attacks were also claimed to be infeasible. A SQUARE attack on 3 rounds of Khazad is described by Barreto and Rijmen in [39], requiring 2^8 key guesses × 2^8 chosen plaintexts

$= 2^{16}$ S-box look-ups to recover one key-byte. A variant on this requires 2^9 chosen plaintexts, 2^{16} S-box look-ups, and 64-bit key guessing, which can be extended to an attack on 4 rounds requiring 2^{80} S-box look-ups. An extension of the Biham-Keller impossible differential attack on 5 rounds of Rijndael [74] can be applied to 3 rounds of Khazad, requiring 2^{13} chosen plaintexts and 2^{64} encryptions. The designers [39] also claim security against truncated differential attacks after 4 rounds, and against Interpolation attacks and Boomerang attacks. Analysis by Biham *et al.* [67] supports these claims. However, further analysis reveals 4 linear characteristics with bias $\simeq \frac{17}{256}$, whereas [39] originally claimed a maximal absolute bias of $\frac{13}{256}$. This resulted in the tweak to the S-box mentioned earlier. The Gilbert-Minier collision attack, which works better than the SQUARE attack on Rijndael, will not work for Khazad since it requires full 64-bit collisions, whereas Rijndael only requires 4-byte collisions owing to slower mixing (diffusion).

Generalised linear characteristics for Khazad with maximum and minimum bias are also found by Parker in [511]. It is found that Khazad already has a moderately low-bias characteristic with Peak-to-Average Power Ratio = 16.0 with respect to the Walsh-Hadamard Transform, and this confirms the bias quoted by the designers. But, as with all S-boxes, this bias increases significantly when approximated by more general linear expressions, where a more general linear expression is here meant to mean $\omega^{f(\mathbf{x})}$, where ω is an rth complex root of 1, and $f(\mathbf{x})$ is a linear expression in the variables, x_i, mod r. For $r \gg 2$ this set of linear expressions is much larger than for binary linear expressions, so much closer approximations can be found.

One advantage that Khazad may have over Rijndael is that the S-box of Khazad does not depend on a simple mathematical function, namely x^{-1}, which is potentially open to attack. However, the nonlinearity of the Rijndael S-box is stronger. Moreover, although the S-box of Khazad is claimed by the designers to be implementable with about $\frac{1}{5}$ of the hardware of that needed for Rijndael, a recent paper by Fuller and Millan [245] shows that the output functions of Rijndael are all affine transformations of the same function. This suggests a potential extra hardware saving for the Rijndael S-box, although this simplification may perhaps later be exploited as a security weakness, as all the output bits of the S-box are simply affine relations of one another. In contrast, the Khazad S-box is relatively unstructured and therefore less of a candidate for potential algebraic attacks. Furthermore, Biham [62] has shown that no such affine relationships exist for the Khazad S-box (see Sect. 2.9.1). However, using the affine-equivalence algorithm developed by Biryukov, De Cannière, Braeken, and Preneel [83], it was found that the P and Q S-boxes of Khazad are both self and mutually-affine equivalent. This implies that the complete Khazad cipher can be described using affine mappings and a single non-linear 4×4-bit look-up table. Note that this is not necessarily as bad as it sounds: each cipher can be described with affine mappings and a single non-linear 2×1 bit AND.

One of the major hindrances to cryptanalysis of Khazad is the extremely high rate of diffusion for the cipher and it may be that new attacks on such ciphers have to exploit more global *iterative* properties of the cipher, such as its permutation structure. Recently some new techniques which analyse the permutation cycle structure of KHAZAD have been proposed by Biryukov and Preneel [81], suggesting promising avenues for future attacks on such ciphers. The techniques exploit the involution structure of KHAZAD by showing that the involutions maintain the permutation cycle structure through successive rounds of KHAZAD. We can write 5-round KHAZAD as follows:

$$k_0 SMk_1 SMk_2 SMk_3 SMk_4 Sk_5$$

where the k_i are the key XORs, S is the nonlinear S-box layer, and M is the MDS linear diffusion layer. By passing the k through M we can rewrite the above as

$$k_0 Sk'_1[MSM]k_2 Sk'_3[MSM]k_4 Sk_5.$$

[81] first investigates the permutation cycle structure of MSM, noting that, as $M = M^{-1}$ (an involution), $MSM = MSM^{-1} = A$, where A is a permutation isomorphic to S. Secondly, [81] shows that the permutation cycle structure of $k_1 Sk_2$, for arbitrary k_1 and k_2, depends only on the difference $\Delta_{k_1 k_2} = k_1 \oplus k_2$. Moreover each cycle length appears in the permutation $k_1 Sk_2$ an even number of times. It follows that $k_1 Sk_2$ consists of more than $2^8 \prod_i l_i$ cycles, where $2l_i$ is the number of cycles of the ith permutation of the eight parallel 8-bit permutations which comprise the $k_1 Sk_2$ layer. Using these results it can be shown that, for two randomly chosen points $x = (x_1, x_2, \ldots, x_8)$ and $y = (y_1, y_2, \ldots, y_8)$, the GCD of the cycle sizes has a good chance of being big (unlike a random permutation). The probability that the x_is and y_is will belong to the same cycle or to two different cycles of the same size is more than $\frac{2l_i}{256}$. By collecting and factoring cycle lengths we can obtain information about $\Delta_{k_1 k_2}$. Next, [81] shows that,

$$[MSM]k_1 Sk_2[MSM] = Ak_1 Sk_2 A = B \qquad \text{(covering 3.5 rounds)}$$

Because of the involutional structure of KHAZAD, B is an isomorphic permutation of A, so it has the same cycle structure as discussed earlier. In particular, it has the same cycle structure as $k_1 Sk_2$. Therefore it is enough to study the fixed points of $S(x) \oplus \Delta_{k_1 k_2}$. It follows that, for a randomly chosen k_1, k_2, $Ak_1 Sk_2 A$ has no fixed points with probability greater than $1 - 2^{-8}$. However, if it has a single fixed point then it must have more than 2^8 fixed points. Finally, [81] examines

$$kSk[MSM]kSk[MSM]ksk \qquad \text{5 rounds.}$$

It is possible to show that,

$$(KH_5(x) \oplus \Delta_k)^n = k_0 SM(KH_3)^n MSk_5$$

where $\Delta_k = k_0 \oplus k_5$ and $KH_j(x)$ means j rounds of KHAZAD. In other words we have a very strong relationship, due to the involutional structure, between 3 rounds and 5 rounds of KHAZAD. It follows that if one can detect peculiarities in the cycle structure of 3-round KHAZAD in less than 2^{64} steps, then this will provide a distinguishing attack on 5-round KHAZAD faster than exhaustive key search.

The paper by Biryukov and de Cannière [82] compares minimal systems of multivariate polynomials which completely define certain block ciphers, including Khazad. The work is motivated by the recent papers [165] and [452, 454] which propose attacks on block ciphers using overdefined systems of linear, quadratic and low degree equations. For Khazad the P S-box consists of 4 quadratic equations in 16 terms, and the Q S-box consists of 6 equations in 18 terms. These equations are used to define 30 equations in 32 linear and 28 quadratic terms for the 8-bit S-box of Khazad and, along with a set of linear equations to define the linear layers, the whole of the block cipher can be described by 2496 equations in 2048 variables using 3840 linear and quadratic terms. Similarly, the key schedule can be described by 2672 equations in 2638 variables using 4384 linear and quadratic terms. As the cipher takes a 64-bit plaintext input and a 128-bit key, 2 plaintext/ciphertext pairs are required to solve the system, implying a doubling of the number of state equations, variables, and terms for the cipher. However the number of equations for the key schedule remains unchanged, as the same schedule is used for both encryptions. In total [82] estimates that 7664 equations in 6464 variables using 12064 linear and quadratic terms are required. Roughly speaking it is desirable to keep the number of free terms as low as possible so as to maximise the solution speed for such a system (see Sect. 2.2.3.17). For Khazad there are 4400 free terms. It is found that twice as many free terms are required for MISTY1, suggesting that MISTY1 is more secure than Khazad [82]. However, this is perhaps misleading because, as [82] points out, they restrict themselves to quadratic equations for their count whereas S-box S7 of MISTY1 has a much more concise representation using cubic equations. If cubic equations are included in the count then MISTY1 may appear less secure than Khazad.

2.4.3 MISTY1

2.4.3.1 The design

MISTY1 was first published in 1996, is a Feistel network based on a 32-bit nonlinear function, takes 64-bit plaintext and a 128-bit key, and is recommended for 8 rounds (more generally a multiple of 4 rounds). Moreover, each pair of rounds is separated by a layer of two 32-bit FL-blocks. MISTY1 can be implemented in situations where resources are heavily constrained, and the constituent lookup tables are optimised for hardware performance. The entire algorithm is built from recursive components such that at each level the structure is again a secure Feistel-like structure. The recursive design adds a lot of complexity to the cipher, making analysis hard. The additional FL or FL^{-1} functions every odd round, for encryption or decryption respectively, take 32-bit input and output as well as taking a 32-bit subkey as input. The FL layers are added to avoid attacks other than differential and linear cryptanalysis. The modified Feistel structure uses an FO function which has 32-bit input and output as well as taking a 64-bit subkey and another 48-bit subkey. This FO function contains an FI function which has 16-bit input and output as well as taking a 16-bit subkey. The FI function contains 7 × 7-bit, and 9 × 9-bit S-boxes. This encryption structure is shown in

Fig. 2.5, where the 64-bit plaintext is first split into two 32-bit left and right parts, and then converted to ciphertext via bitwise XOR, FO and FL/FL^{-1} operations.

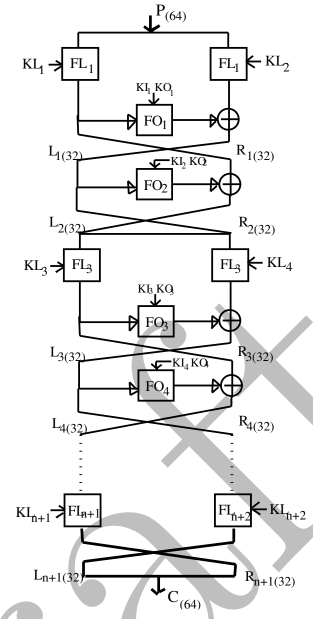


Fig. 2.5. Encryption for MISTY1

The rounds are summarised algebraically as follows:

$$\begin{array}{lll}
\text{odd rounds} & R_i = \mathrm{FL}_i(L_{i-1}, KL_i) & L_i = \mathrm{FL}_{i+1}(R_{i-1}, KL_{i+1}) \oplus \mathrm{FO}_i(R_i, KO_i, KI_i) \\
\text{even rounds} & R_i = L_{i-1} & L_i = R_{i-1} \oplus FO_i(R_i, KO_i, KI_i)
\end{array}$$

with a final FL operation after the last round, to ensure that decryption is like encryption apart from a reverse of the subkey order and the interchange of FL and FL^{-1}. The KL's, KO's and KI's in the above round expressions are subkeys which are derived from the 128-bit key by using the following key schedule. We first partition the 128-bit key into eight consecutive 16-bit key values $K_1, \ldots, K_8$. We then generate K_i' subkeys by using the FI function as follows,

$$K_i' = FI(K_i, K_{i+1})$$

where the indices are taken cyclically. Then using these subkeys we can derive the subkeys used in the round functions by applying the subkey mapping table of Table 2.3:

The subkeys of Table 2.3 are then used in the functions FL, FO, and FI, as shown in Figs 2.6, 2.7, and 2.8 (the FL^{-1} function is similar to the FL function).

The input to the FL function comprises a 32-bit data input $X_{(32)}$ and a 32-bit subkey $KL_{i(32)}$. Note that $\cap$ means bitwise AND, and $\cup$ means bitwise OR. The input data is split into two 16-bit halves, $X_{L(16)}$ and $X_{R(16)}$ where,

Table 2.3. Subkey Mapping Table

Round	KO_{i1}	KO_{i2}	KO_{i3}	KO_{i4}	KI_{i1}	KI_{i2}	KI_{i3}	KI_{iL}	KI_{iR}
Actual	K_i	K_{i+2}	K_{i+7}	K_{i+4}	K'_{i+5}	K'_{i+1}	K'_{i+3}	$K_{\frac{i+1}{2}}$ odd i $K'_{\frac{i}{2}+2}$ even i	$K'_{\frac{i+1}{2}+6}$ odd i $K_{\frac{i}{2}+4}$ even i

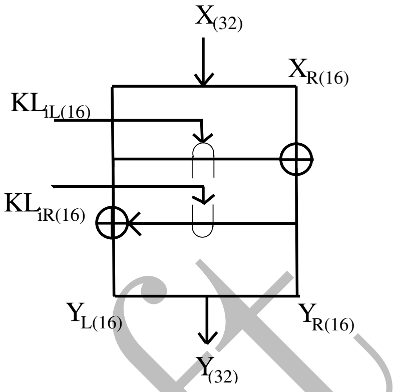


Fig. 2.6. FL function for MISTY1

$$X_{(32)} = X_{L(16)} | X_{R(16)}$$

The subkey is split into two 16-bit subkeys, $KL_{iL(16)}$ and $KL_{iR(16)}$ where,

$$KL_{i(32)} = KL_{iL(16)} | KL_{iR(16)}$$

The FL function is then defined as,

$$\begin{aligned} Y_{R(16)} &= (X_{L(16)} \cap KL_{iL(16)}) \oplus X_{R(16)} \\ Y_{L(16)} &= (Y_{R(16)} \cup KL_{iR(16)}) \oplus X_{L(16)} \end{aligned}$$

where the output is $Y_{(32)} = Y_{L(16)} | Y_{R(16)}$.

The input to the FO function comprises a 32-bit data input $X_{(32)}$ and two sets of subkeys, a 64-bit subkey $KO_{i(64)}$ and a 48-bit subkey $KI_{i(48)}$. The input data is split into two 16-bit halves, L_0 and R_0 where,

$$X_{i(32)} = L_{0(16)} | R_{0(16)}$$

The subkeys are divided into 16-bit subkeys where

$$\begin{aligned} KO_{i(64)} &= KO_{i1(16)} | KO_{i2(16)} | KO_{i3(16)} | KO_{i4(16)} \\ KI_{i(48)} &= KI_{i1(16)} | KI_{i2(16)} | KI_{i3(16)} \end{aligned}$$

Then for each integer j with $1 \leq j \leq 3$ we define:

$$\begin{aligned} R_j &= FI_{ij}(L_{j-1} \oplus KO_{ij}, KI_{ij}) \oplus R_{j-1} \\ L_j &= R_{j-1} \end{aligned}$$

Finally, $L_{3(16)}$ is XORed with KO_{i4} and concatenated with $R_{3(16)}$. The function returns $Y_{i(32)} = (L_{3(16)} \oplus KO_{i4}) | R_{3(16)}$.

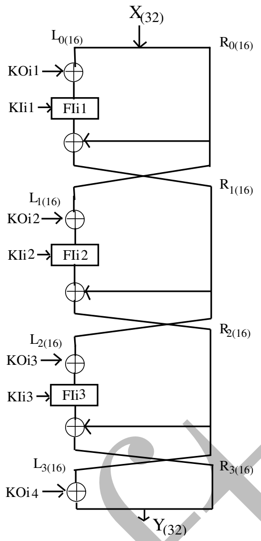


Fig. 2.7. FO function for MISTY1

The function FI_j takes a 16-bit data input $X_{j(16)}$ and a 16-bit subkey $KI_{ij(16)}$. The input data is split into two unequal components, a 9-bit left half $L_{0(9)}$ and a 7-bit right half $R_{0(7)}$ where $X_{j(16)} = L_{0(9)}|R_{0(7)}$. The key $KI_{ij(16)}$ is similarly split into two unequal components. FI uses two S-boxes, S7 and S9, mapping 7 bits to 7 bits, and 9 bits to 9 bits respectively. FI also uses two additional functions $ZE()$ and $TR()$. These are defined as follows,

$y_{(9)} = ZE(x_{(7)})$ ZE takes the 7-bit $x_{(7)}$ and converts it to a 9-bit value $y_{(9)}$ by adding two zero bits to the most significant end.

$y_{(7)} = TR(x_9)$ TR takes the 9-bit $x_{(9)}$ and converts it to a 7-bit value $y_{(7)}$ by discarding the two leftmost bits.

Then FI is defined by the following operations.

$$\begin{array}{ll} L_{1(7)} = R_{0(7)} & R_{1(9)} = S9(L_{0(9)}) \oplus ZE(R_{0(7)}) \\ L_{2(9)} = R_{1(9)} \oplus KI_{ijR(9)} & R_{2(7)} = S7(L_{1(7)}) \oplus TR(R_{1(9)}) \oplus KI_{ijL(7)} \\ L_{3(7)} = R_{2(7)} & R_{3(9)} = S9(L_{2(9)}) \oplus ZE(R_{2(7)}) \end{array}$$

Finally, $L_{3(7)}$ and $R_{3(9)}$ are concatenated to give $Y_{(16)} = L_{3(7)}|R_{3(9)}$.

The S-boxes which are contained in the FI function are designed to be efficient for both combinational logic and look-up table implementations, owing to the relatively small numbers of terms in the Algebraic Normal Forms (ANFs) of the constituent functions of the S-boxes. This results in small hardware implementation cost and short delay time.

Both 7 and 9-bit S-boxes are chosen to optimise the *provable security* of MISTY1 against differential and linear cryptanalysis, and both S-boxes achieve the minimum possible differential and linear biases for S-boxes of their size.

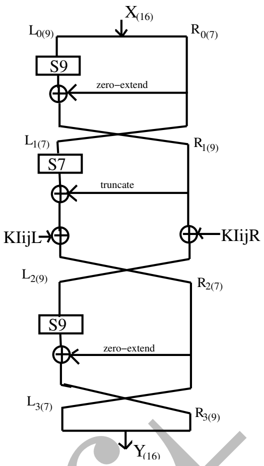


Fig. 2.8. FI function for MISTY1

2.4.3.2 Security analysis

MISTY1 [426] has been widely studied for five years and no serious security flaws have been found. It should be noted that a variant of MISTY1, namely KASUMI, has been chosen for the 3GPP standard. Therefore many attacks on MISTY1 may also be relevant to KASUMI, and vice versa. Both MISTY1 and KASUMI use nonlinear invertible FL functions to introduce AND and OR operations to the cipher. However, unlike MISTY1, KASUMI is a pure 8-round Feistel cipher, where in the odd rounds we first apply FO then FL, and in the even rounds we first apply FL then FO. Moreover, unlike KASUMI, in MISTY1 after the final swap there is an additional XOR with the subkey on the left-hand side. The S7 and S9 S-boxes of MISTY1 and KASUMI are not identical but are very similar and both exhibit affine relationships between the bit outputs [62] (see Sect. 2.9.1). In particular, for the S-box, S9, of MISTY1, the rotation of the input by any number of bits does *not* affect the least significant bit of the output (see Sect. 2.9.1) — this is, perhaps, a surprising result. It is considered that KASUMI has a weaker key schedule than MISTY1, as the key schedule of MISTY1 is nonlinear whereas that of KASUMI is linear. Further details of the differences between MISTY1 and KASUMI can be found in [210]. There is also a block cipher called MISTY2 by the same designers. MISTY2 is also a 64-bit block cipher using 128-bit key. This cipher has a newer structure than MISTY1 and recommends the use of 12 rounds as opposed to 8 rounds for MISTY1.

Attacks on MISTY1 without the FL operations have been accomplished up to five rounds. The low algebraic degree of the constituent functions of the MISTY1 S-boxes has invited higher order differential attacks by Lai [385] and Knudsen [352] on MISTY1 without FL functions. A higher order differential attack on MISTY1 without FL functions has also been presented by Tanaka *et al.* [604]. However, it appears that the key action in the FL function can significantly modify the algebraic degree of MISTY1. The Slide attack has been proposed against

MISTY1 by Biryukov and Wagner [90] where the same subkey is applied to every nth round, and this is appropriate to MISTY1 because of its simple key schedule. It turns out that without the FL functions, the slide attack works when one of 65536 keys is used. However, MISTY1 maintains some resistance to slide attacks, and this is because one of the design criteria for MISTY1 was resistance to related-key attacks. It has been shown by Biham *et al.* [65] and Knudsen [353] that any Feistel cipher with a bijective round function has impossible differentials in 5 rounds, and MISTY1 without FL layers falls into this category. However, Impossible differential attacks appear to be inappropriate for MISTY1 as the FL functions add extra dependency on the particular key at each round. MISTY1 is designed to have provable security against differential and linear cryptanalysis, and this proof is achieved by bounding the average differential/linear probabilities for the recursive layers of MISTY1; if the average differential/linear probability of each layer is p then the complete cipher has probability upper-bounded by p^4. It is claimed by the designers that the unequal division of the S-boxes into 7 bits and 9 bits has an advantage against differential and linear cryptanalysis, as the probability bound can be made lower for S-boxes that use odd as opposed to even numbers of bits. But there are hardware and software penalties resulting from this asymmetry. Knudsen and Moen have recently applied Integral Cryptanalysis [371, 448] to MISTY1 including FL functions. This includes 4 round and 5 round attacks. The integral attacks exploit the Sakurai-Zheng property that was initially applied to MISTY2. This property is as follows. Let $F(x, y)$ denote the left half and right half of the output after three rounds of MISTY2 on plaintext $< x, y >$. Then $F(x, y) = f(x) \oplus g(y)$ where f and g are key-dependent bijective mappings. Therefore, for any two arbitrary sets of 32-bit values, $\mathbf{S}$ and $\mathbf{T}$, we have, $\sum_{<x,y>\in \mathbf{S}\times\mathbf{T}} F(x, y) = 0$. This is a three-round integral for MISTY2. This property can also be applied to MISTY1. The 5-round attack by Knudsen and Wagner in [371] uses the Sakurai-Zheng property once, and the 4-round attack uses the property twice. It requires a data complexity of 2^{34} and a time/memory complexity of 2^{48}. Also a new attack, the Slicing Attack by Kuhn [380, 381] has been applied to the 4-round version of MISTY1, making use of the special structure and position of the key-dependent linear FL functions. These FL functions present a subtle weakness in the 4-round version of the cipher. Both this attack and that of [371] are particularly interesting as they include the FL layers, unlike many other attacks which ignore the FL layer.

Generalised linear characteristics through both the 7-bit and 9-bit S-boxes of MISTY1 with maximum and minimum bias are searched for in [511]. It is found that, although both S-Boxes have an optimally low bias relative to the Walsh-Hadamard Transform (WHT), with PAR = 2.0, the bias increases significantly with respect to many generalised linear approximations, in particular those covered by the **HI** transform, where the PAR = 16.0 and 32.0 for 7 and 9-bit S-boxes respectively. This suggests that, whereas the odd-length S-box width (7 and 9) minimises the possible linear characteristic with respect to the WHT, the odd-length restriction in fact weakens the S-box with respect to certain generalised linear approximations, to give a PAR of $2^{\lceil \frac{n}{2} \rceil}$, where $n = 7$ or 9. This is a counter-argument to the argument proposed by the designer for using odd-length

S-boxes — more general linear approximations suggest that even-length S-boxes may be better (although such generalised linear approximations have not yet led to an attack).

As discussed in the security analysis for KHAZAD, Biryukov and De Cannière [82] compare minimal systems of multivariate polynomials which completely define certain block ciphers, including MISTY1. For MISTY1 S9 is designed as a system of 9 quadratic equations in 54 terms. S7 is designed as a system of 7 cubic equations in 65 terms, but can also be defined by 11 quadratic equations in 93 terms. The quadratic representations are utilised and, along with a set of linear equations to define the linear layers, the whole of the block cipher can be described by 1824 equations in 1664 variables using 5848 linear and quadratic terms [82]. Similarly, the key schedule can be described by 432 equations in 528 variables using 1848 linear and quadratic terms. Two plaintext/ciphertext pairs are required to solve the system, implying a doubling of the cipher count, but not for the key schedule. In total [82] estimates that 4080 equations in 3856 variables using 13544 linear and quadratic terms are required. This gives the number of free terms as the number of terms minus the number of equations (see Sect. 2.2.3.17). For MISTY1 this is 9464 free terms. When compared with KHAZAD this seems large, but the count may decrease significantly if cubic representations are considered for S7 of MISTY1.

The main advantage of MISTY1 is its provable security against differential and linear cryptanalysis.

2.4.4 SAFER++ (64-bit block)

2.4.4.1 The design

SAFER++ [420] is a development from the existing SAFER family of ciphers and uses a combination of substitution and linear transformation to achieve confusion and diffusion, respectively. Recently, the 128-bit plaintext version has been adopted for use in the authentication scheme in Bluetooth, the wireless communication protocol. We consider here the legacy version which takes in 64-bit plaintext and 128-bit key, and outputs 64-bit ciphertext. The designers recommend this version be used with 8 rounds to achieve sufficient security. SAFER+ was a submission to the AES, and it is claimed that SAFER++is simpler and faster than SAFER+, and at least as strong, cryptographically. The main new feature in SAFER++, as compared to all earlier versions of SAFER (including SAFER+), is the use of a *4-point* Pseudo-Hadamard Transform (PHT), instead of the *2-point* PHT. SAFER++uses the 4-point PHT to achieve fast, rapid diffusion at low complexity. One 16-byte subkey is used with each round, along with one post-cipher *output transformation* which is a final 8-byte subkey *addition.* Another aspect of the cipher is its use of two incompatible group additions to achieve key addition, namely bitwise XOR (mod 2), and bytewise addition, mod 256. Moreover, the S-boxes are exponential and logarithmic functions, mod 257, which are combined with the two addition operations in such a way as to thwart potential homomorphisms. One round of the encryption is shown in Fig. 2.9, and an alternative view of the encryption round is given in Fig. 2.10. The decryption round is similar; however even if one does not consider the key schedule, decryption is not identical to encryption for SAFER++.

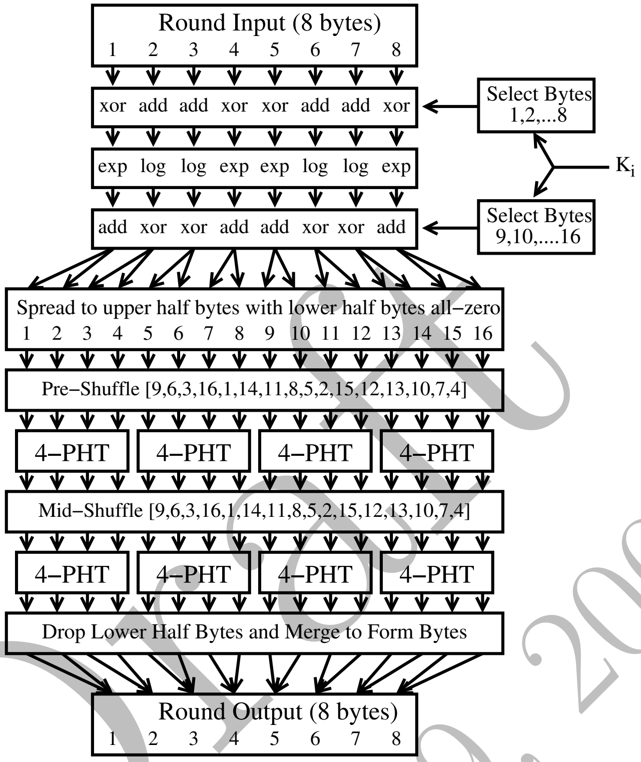

Round Input (8 bytes)
1 2 3 4 5 6 7 8
xor add add xor xor add add xor
Select Bytes 1,2,...8
exp log log exp exp log log exp
K_i
add xor xor add add xor xor add
Select Bytes 9,10,....16
Spread to upper half bytes with lower half bytes all−zero
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
Pre−Shuffle [9,6,3,16,1,14,11,8,5,2,15,12,13,10,7,4]
4−PHT
4−PHT
4−PHT
4−PHT
Mid−Shuffle [9,6,3,16,1,14,11,8,5,2,15,12,13,10,7,4]
4−PHT
4−PHT
4−PHT
4−PHT
Drop Lower Half Bytes and Merge to Form Bytes
Round Output (8 bytes)
1 2 3 4 5 6 7 8

Fig. 2.9. Encryption Round for SAFER++ (64-bit block)

Key addition for a round is bytewise and uses two different (incompatible) group operations, where subkey bytes 1,4,5,8 are added using bitwise XOR, and subkey bytes 2,3,6,7 are added using bytewise addition, mod 256. After 8 rounds of Fig. 2.9 there is a final 8-byte key addition whose operations are identical to the first addition in each round. The decryption process follows the same structure as encryption with the round keys used in reverse order, the PHT replaced with the inverse PHT, the shuffle replaced with the inverse shuffle, addition replaced with subtraction, and exp (log) replaced with log (exp). The 4-PHT matrix, H_4, which is used to achieve linear diffusion is as follows,

$$H_4 = \begin{pmatrix} 2 & 1 & 1 & 1 \\ 1 & 2 & 1 & 1 \\ 1 & 1 & 2 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix} \qquad \bmod 256$$

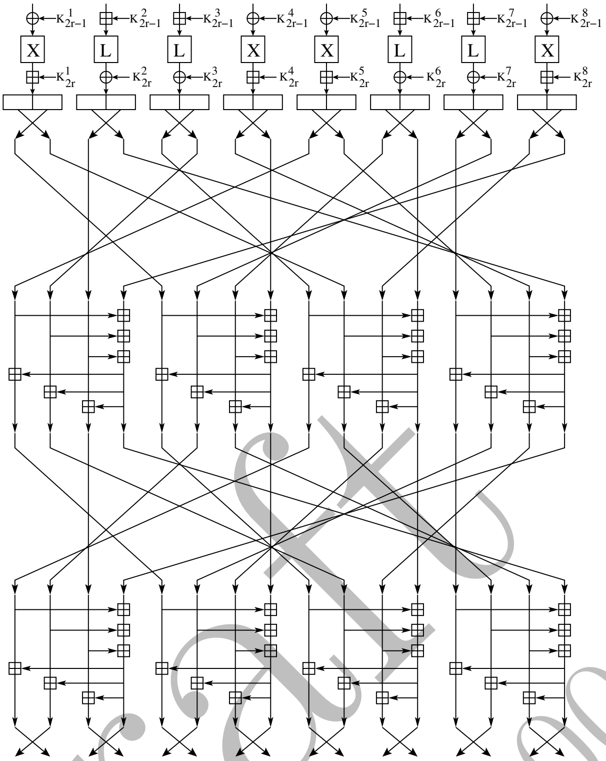


Fig. 2.10. Alternative View of Encryption Round for SAFER++ (64-bit block)

where each element of the matrix represents an 8-bit byte, and H_4 acts on a vector with 8-bit byte entries, mod 256. This PHT matrix can be implemented efficiently, as H_4 is a matrix of low weight. The shuffle permutation has also been chosen to maximise diffusion in conjunction with the PHT. H_4 is not an involution and the inverse of H_4, which is used for decryption to implement the inverse PHT, is as follows,

$$H_4 = \begin{pmatrix} 1 & 0 & 0 & -1 \\ 0 & 1 & 0 & -1 \\ 0 & 0 & 1 & -1 \\ -1 & -1 & -1 & 4 \end{pmatrix} \qquad \text{mod } 256$$

H_4^{-1} is also easy to implement, as it also has low weight. The advantage of the PHT method of diffusion is that a linear diffusion matrix, M, of dimensions 16×16 bytes is implemented by means of smaller, simpler 4×4 byte 4-PHT matrices. Moreover, the designers argue that because each row of M contains at least ten 1's, the diffusion of the cipher is wide and fast.

The nonlinear layer of the cipher is achieved by means of two S-boxes, exp and log. exp realises the function $y = 45^x \mod 257$ for bytes 1,4,5,8, and log realises the function $y = \log_{45}(x) \mod 257$ for bytes 2,3,6,7, where $\log_{45}(0) =$

128 by convention. The exp operation is juxtaposed with XOR, and these two operations do not admit a homomorphism, explicity $45^{x_1 \oplus x_2} \neq 45^{x_1} 45^{x_2}$. This incompatibility between exp and XOR enhances the security of the system. A similar incompatibility exists between log and add, which are also juxtaposed in the cipher. One reason for choosing S-boxes based on exp and log is that their associated ANF (Algebraic Normal Form - i.e. boolean expression) descriptions relating input and output look random, are of high degree, and contain many terms. This is a different philosophy than that used to design the S-boxes for MISTY1, where the ANF functions contain relatively few terms so as to simplify implementation complexity.

The key schedule for SAFER++ (64-bit block) uses 9 16-byte bias words in order to randomize the produced subkeys so as to help to avoid weak keys. These bias words, B_j, are determined by,

$$B_{i,j} = 45^{\left(45^{17i+j} \mod 257\right)} \mod 257$$

where $B_{i,j}$ is the ith byte of B_j.

The odd-index 16-byte subkeys are generated using the method outlined in Fig. 2.11, and given in detail in Fig. 2.12. A similar strategy holds for the even-index subkeys.

For SAFER++ (64-bit block), only subkeys $K_1, \ldots, K_9$ are used. Another interesting aspect of SAFER++is that it is possible to develop mini-versions of the cipher with, e.g. 4-bit nibble wordlengths, which still retain the main features of the cipher, and this enables the thorough testing of different attack strategies before extrapolating them to the larger, real cipher. Also one should note that the spreading of 64 bits to 128 bits, and the subsequent dropping of 128 bits down to 64 bits is an unusual feature of SAFER++which distinguishes it from many other block ciphers.

2.4.4.2 Security analysis

No security flaws have been found with SAFER++ (64-bit block) [420] and it has many similarities to SAFER++ (128-bit block). One weakness found in previous versions of the SAFER family by Knudsen was in the key-schedules [351, 356], but these weaknesses have been dealt with in SAFER++. Other pre-NESSIE attacks on the SAFER family include truncated differentials by Knudsen and Berson [362], and Murphy [451] identifies a potential algebraic weakness regarding the existence of invariant $\mathbb{Z}$-modules within the PHT layer. These modules and their cosets are not diffused by the PHT layer, and so provide a way to cope with diffusion in SAFER, regardless of the key schedule. One attack in [451] which used this property enabled a projection of the message/ciphertext space onto a 4-byte $\mathbb{Z}$-submodule so that the probability of any message projection giving any ciphertext projection is independent of $\frac{1}{4}$ of the key bytes. This result, along with the results of [351] led to a change in the SAFER key schedule. In [619] Wernsdorf shows that the round functions of SAFER++ (64-bit block) generate the alternating group over the set $\{0,1\}^{128}$, eliminating some potential weaknesses of SAFER++ (64-bit block), e.g. non-trivial factor groups (the alternating group is

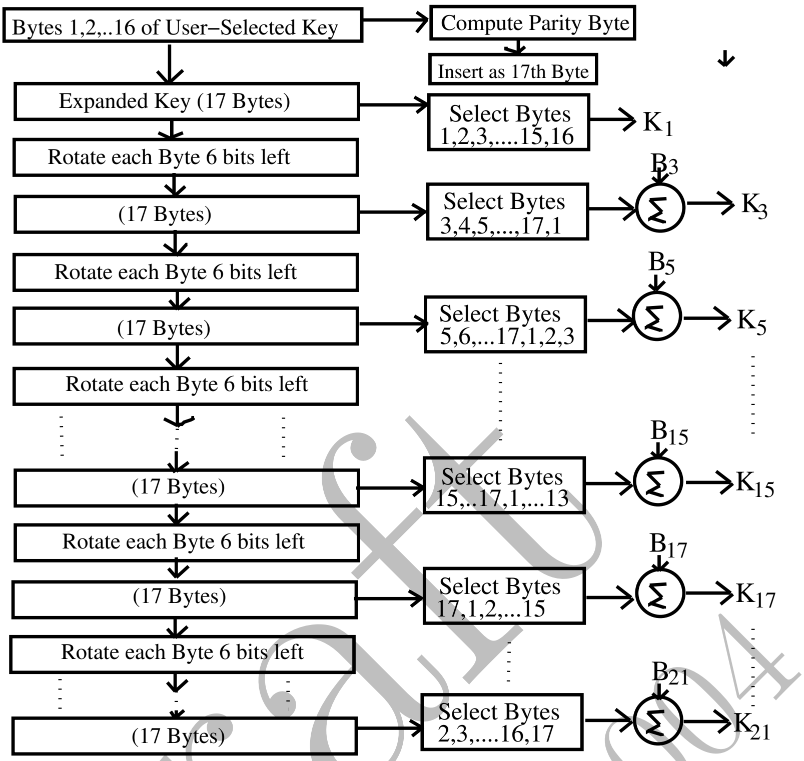

Bytes 1,2,..16 of User−Selected Key
Compute Parity Byte
Insert as 17th Byte
Expanded Key (17 Bytes)
Select Bytes 1,2,3,....15,16
K1
Rotate each Byte 6 bits left
(17 Bytes)
Select Bytes 3,4,5,...,17,1
B3
K3
Rotate each Byte 6 bits left
(17 Bytes)
Select Bytes 5,6,...17,1,2,3
B5
K5
Rotate each Byte 6 bits left
(17 Bytes)
Select Bytes 15,..17,1,...13
B15
K15
Rotate each Byte 6 bits left
(17 Bytes)
Select Bytes 17,1,2,...15
B17
K17
Rotate each Byte 6 bits left
(17 Bytes)
Select Bytes 2,3,....16,17
B21
K21

Fig. 2.11. Generation of Odd-Index Subkeys in the SAFER++ (64-bit block) Key Schedules. $\sum$ denotes bytewise summation, mod 256

a large, simple, primitive and $(2^{128} - 2)$-transitive group). The designers recommend 8 rounds to ensure security for the cipher and not fewer than 7 rounds, and they claim that one of the main reasons for the security of the cipher against differential and linear cryptanalysis is the high diffusion PHT layer. The designers conclude that SAFER++with six or more rounds is secure against differential cryptanalysis, and with two and a half or more rounds is secure against linear cryptanalysis. Note also that, in [356], Knudsen identified a differential such that, if X is the exponentiation function of SAFER, then $X(a)+X(a+128) = 1($ mod 256) holds with probability 1, and Nakahara has extended this observation to relate not only the msb of the input to the lsb of the output, but also to the 2nd lsb of the output [327, 456]. Moreover, in [327, 456], Nakahara generalises to other exponents, not just 45. Recently Nakahara *et al.* [459] applied techniques that were first used to more generally attack the SAFER family in [458], and [459] showed that three and a half rounds of SAFER++ (64-bit block) can be attacked requiring 2^{33} known plaintexts. The reason for this discrepancy between the two and a half rounds claimed by the designers and the three to four rounds claimed by [459] is largely because the designers restricted themselves to homomorphic attacks, whereas [459] applies strictly non-homomorphic attacks where some key

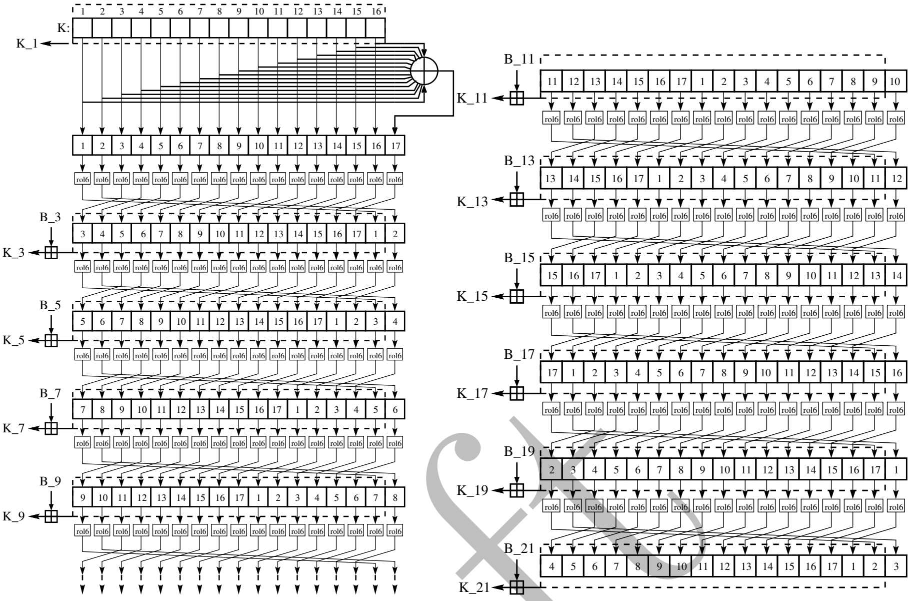


Fig. 2.12. Detailed Generation of Odd-Index Subkeys in the SAFER++ (64-bit block) Key Schedules

bits are assumed fixed. Therefore the results of [459] must be seen in the context of a weak-key class. Table 2.4 gives a summary of the complexity of linear attacks on SAFER++ [459].

Table 2.4. Complexity of Linear Attacks on SAFER++ (64-bit block)

# Rounds Attacked	Linear Relation	# Known Plaintexts	# Subkey Bits Explored	Attack Complexity	Fraction of Keys
2	(1)	2^5	37	2^{42} ♣	
3.5	(3)	2^{33}	88	2^{121} ♣	2^{-6}

♣ The attack applies to all key sizes defined for SAFER++.

One important point to note with regard to the linear attack of [459] is that it identifies a surprisingly small byte and bit branch number for the PHT diffusion layer. Whereas the designers of SAFER++use the fact that the lowest row weight of the matrix M is 10, implying a high diffusion, a more detailed examination of the matrix reveals a byte branch number ≤ 7 and a bit branch number ≤ 5. Moreover the S-box defined by $y = 45^x \mod 257$ contains a linear relationship which holds with probability 1. In other words, we can write $y_0 = f(x_0, x_1, \ldots, x_6) + x_7$, where y_0 is one of the output bits of the S-box, and depends linearly on the input, x_7. This moderated diffusion combined with a high linear characteristic is what enables this linear attack on three to four rounds of SAFER++.

It is also interesting to note that affine relationships exist between the bit outputs of the S-box, both for exp45 and log45 [62] (see Sect. 2.9.1).

2.4.5 Triple-DES

2.4.5.1 The design

One variant of Triple-DES which is in widespread use is called two-key Triple-DES. It takes a 64-bit plaintext and a $2 \times 56 = 112$-bit key. It occurs as a natural extension of the existing standard DES, where security has been increased, in particular the key input size, by repeating the cipher three times and the key twice. Note that double-DES is not an option due to a meet-in-the-middle attack which renders double-DES with no greater security than single DES. To allow for backwards-compatibility, two-key Triple-DES was suggested in the form encrypt-decrypt-encrypt although, in fact, the form encrypt-encrypt-encrypt can also revert to single DES encryption by using the all-zero key in the first two encryptions, and a single-DES key in the third encryption (the all-zero key being self-dual). There is also three-key Triple-DES (3DES), which takes 64-bit plaintexts and a 168-bit key, and this is also in widespread use. It is considered to be a lot more secure than two-key Triple-DES and can also be made backwards compatible with DES by making all three keys the same in encrypt-decrypt-encrypt mode. One should also mention DESX which also takes three keys but is relatively efficient compared to Triple-DES, requiring only a single DES encryption preceded by XOR with another key, and completed by XOR with a third key.

A round of DES is summarised in Fig. 2.13. DES is a Feistel cipher. L and R are the left and right splittings of the 64-bit plaintext, and C and D are the left and right splittings of the 56-bit key. The key is input linearly via XOR and there are 8 6-bit in, 4-bit out S-boxes which are applied in parallel to the 48-bit input to give 32-bit output.

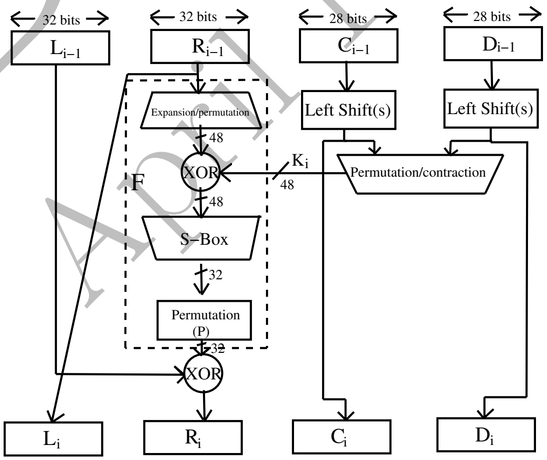


Fig. 2.13. A Round of DES

Essentially the DES standard recommends 16 rounds, and three-key Triple DES is simply a concatenation of three instances of DES, where a different key is input for each instance of DES to give a total key input of 3 × 56 = 168 bits.

2.4.5.2 Security analysis

The security of Triple-DES is significantly less than the 128 bits required for this cipher category and its performance on workstations is rather bad. However Triple-DES will still be considered as a benchmark for other algorithms. It has been shown by Merkle and Hellman in [442] that two-key triple encryption can be broken using 2^{56} chosen plaintexts, and 2^{112} single encryptions. The standard way to attack triple-DES is to use the meet-in-the-middle attack [441], requiring 3 plaintext/ciphertext pairs, and 2^{112} single encryptions. Advanced meet-in-the-middle attacks for two-key triple encryption have been proposed by Van Oorschot and Wiener in [608]. Two-key Triple-DES is generally considered weaker than three-key Triple-DES.

Clearly, any attack on DES is relevant to an attack on Triple-DES and in fact the best attacks on reduced variants of Triple DES are the attacks on DES. A well-known weakness of DES is the *complementation property*. Specifically, let p and k be plaintext and key inputs to DES, respectively. Then the complementation property is summarised as follows:

$$\mathrm{DES}_k(p) = \overline{\mathrm{DES}_{\overline{k}}(\overline{p})}$$

where $\overline{*}$ denotes the complement of the bit-string, $*$. However, this evident deviation from a random cipher does not comprise the cipher to any great extent, only enabling an improvement from 3 known plaintexts and 2^{112} complexity to 6 known plaintexts and 2^{111} complexity. It has been shown that differential cryptanalysis can cover as many as 18 rounds of DES, and it is suspected that this may also be so for linear cryptanalysis. Moreover, due to the no-swap between rounds 16 and 17, it may be possible to gain one more round. It is also interesting to note that affine relationships exist between the bit outputs of the S-box, for the first seven S-boxes of DES [62] (see Sect. 2.9.1).

Kelsey *et al.* used related-key techniques [345] to attack three-key triple-DES, and Biham [60] encrypts the same plaintext 2^{28} times using Triple-DES under 2^{28} different keys, allowing the attacker to recover one of the 2^{28} keys using 2^{84} steps and 2^{84} single encryptions. In [406] a more efficient meet-in-the-middle attack is presented by Lucks which can break three-key triple DES with about 1.3×2^{104} single encryption steps, and 2^{32} known plaintext/ciphertext pairs. The attack works by saving single encryption steps and exploiting known and/or chosen plaintext/ciphertext pairs, the complementation property weakness of DES and a certain number of weak keys.

2.5 128-bit block ciphers considered during Phase II

The 128-bit block ciphers selected for phase II of NESSIE were Camellia, RC6, and SAFER++ (128-bit block). We also continued to study the AES, in order to compare the submitted ciphers with a current standard. None of the ciphers considered have been broken so the following security evaluation identifies weaknesses that occur in reduced-round versions of the ciphers, and weaknesses that may lead to more effective attacks in the future. We first describe each cipher in some detail, along with the most important attacks known on each cipher. Note that the algorithms given here are not complete specifications, but references are given to complete specifications which may be found on the NESSIE website. After discussing each cipher we summarise and compare some of the distinguishing features of the ciphers, identifying potential weaknesses and noting the best-known attacks, as shown in Tables 2.16 and 2.17 of Sect. 2.9.2.

2.5.1 Camellia

2.5.1.1 The design

Camellia [24] is an 18-round 128-bit block cipher which supports 128-, 192-, and 256-bit key lengths, with two layers of two 64-bit FL-blocks after the 6th and 12th rounds. It is a byte-oriented 18-round Feistel cipher with a particular emphasis on low-cost hardware applications, and is designed to be resistant to differential and linear cryptanalysis with linear bias and differential probabilities both $\leq 2^{-128}$. Camellia has recently been selected by CRYPTREC. Camellia uses four 8×8-bit S-boxes with input and output affine transformations and logical operations. However there is no 32-bit integer addition, so as to avoid the possibility of a long critical path (longest inherent sequential computation) due to the carry propagation associated with addition. The diffusion layer uses a linear transformation based on a Maximum-Distance-Separable code with a branch number of 5 (activity on t input bytes to the linear transformation layer will diffuse to activity on at least $5-t$ output bytes from the linear transformation layer). Camellia also uses FL and FL^{-1} functions which are inserted every 6 rounds so as to enhance irregularity of the cipher. The FL functions are similar to those of MISTY1, except that Camellia also uses a 1-bit rotation so as to make bytewise cryptanalysis harder. The entire structure of Camellia is given in Fig. 2.14. The FL and FL^{-1} functions are shown in Fig. 2.15.

The design of Camellia is based on E2 which was a previous block cipher by the same designers and was a submission to the AES. The main difference between E2 and Camellia is the adoption, for Camellia, of the 1-round SPN, not the 2-round SPN of E2, leading to an expected improvement in speed for Camellia. The design of the F-function of Camellia follows that of E2. The F-function transforms a 64-bit input, X_{64}, to a 64-bit output, Y_{64}, using a 64-bit subkey k_{64}, given by $Y_{64} = F(X_{64}, k_{64})$. More specifically, the F function is described by,

$$\begin{aligned} &F : \mathbf{L} \times \mathbf{L} \to \mathbf{L} \\ &(X_{64}, k_{64}) \to Y_{64} = P(S(X_{64} \oplus k_{64})) \end{aligned}$$

where $\mathbf{L}$ denotes a vector-space of 64 bits, P is a byte-linear transformation, and S operates in parallel on 8-bit segments of the 64-bit input, with each 8-bit segment being subject to an 8×8 S-box transformation (one of 4 S-boxes, so that S is given by $s_1, s_2, s_3, s_4, s_2, s_3, s_4, s_1$). Each of the 8×8-bit S-boxes, s_1, s_2, s_3, s_4

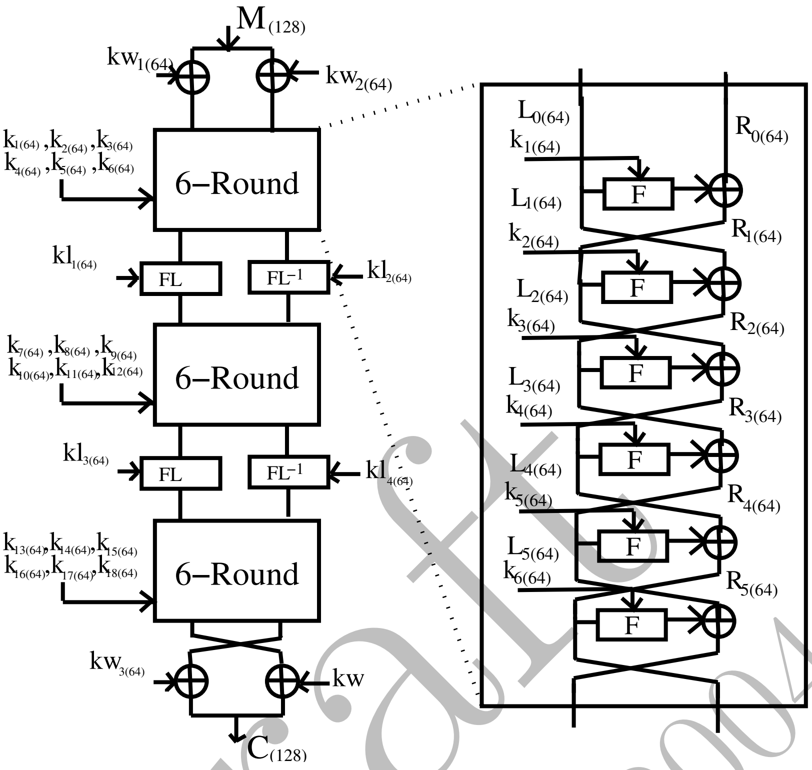


Fig. 2.14. Camellia

is affine equivalent to x^{-1} over GF(2^8) — which is similar to the Rijndael design. The P transformation is designed to have an optimal branch number and can be represented as follows,

$$\begin{pmatrix} z_8 \\ z_7 \\ \dots \\ z_1 \end{pmatrix} \rightarrow \begin{pmatrix} z'_8 \\ z'_7 \\ \dots \\ z'_1 \end{pmatrix} = P \begin{pmatrix} z_8 \\ z_7 \\ \dots \\ z_1 \end{pmatrix}$$

where,

$$P = \begin{pmatrix} 0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 \end{pmatrix}$$

The key schedule of Camellia makes use of the F-function of the encryption module, and is the same for encryption and decryption. The user key is encrypted

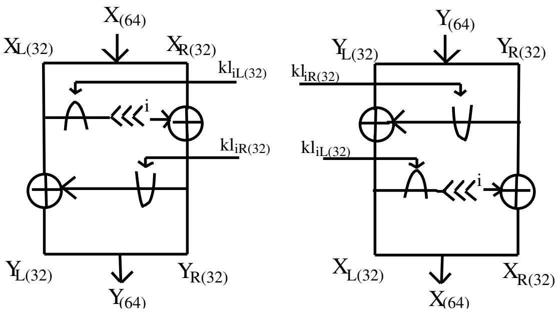


Fig. 2.15. FL and FL^{-1} Functions for Camellia

by means of the F-function using pre-fixed constants, where these constants, $\sum_i$, are defined as continuous values from the hexadecimal representation of the square root of the ith prime. The subkeys are then generated partly from rotated values of the user-input key, K (where $K = K_{L(128)}$, $K = K_{L(128)}||K_{RL(64)}$, or $K = K_{L(128)}||K_{R(128)}$, for a 128-bit, 192-bit, or 256-bit key, K, respectively), and partly from rotated values of the encrypted keys, K_A and K_B. Fig. 2.16 shows how to generate these encrypted keys.

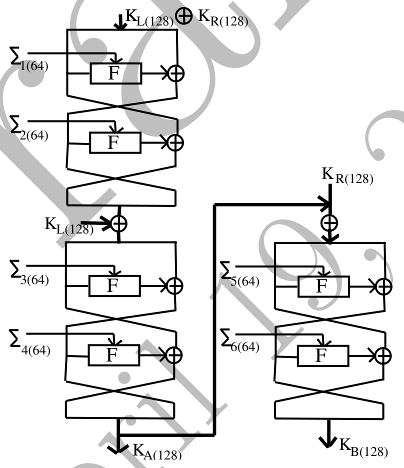


Fig. 2.16. Key Schedule for Camellia

We refer the reader to the documentation of Camellia [24] for a more detailed description.

2.5.1.2 Security analysis

No security flaws have been found for Camellia and it has an interesting design. The designers [25] claim that for Camellia no differential/linear characteristics exist with linear bias and differential probabilities $> 2^{-128}$ over 12 rounds and 2^{-132} over 15 rounds. This claim is due to the use of four S-boxes which are affine transformations of x^{-1} over GF(2^8). This particular mathematical function ensures optimal differential and linear characteristics through the S-box (irrespective of the affine transformation) of 2^{-6}. Another reason for the strong resistance of the cipher to differential and linear cryptanalysis is the use of a linear transformation for the diffusion layer with a high branch number of 5. The affine transformations which modify the input and output to the S-boxes are designed to provide resistance to interpolation attacks by making each S-box less algebraic in character. The designers also claim that 10 rounds is indistinguishable from a random permutation with respect to truncated differential and linear cryptanalysis. As with all Feistel ciphers with bijective F-function, an impossible differential attack exists over 5 rounds of Camellia [353], but the designers have not found any other attacks of this type. The designers also claim security against interpolation, linear sum, and Square attacks. Iterated ciphers with identical rounds are susceptible to the Slide attack, and this is one reason why the FL functions are inserted in the cipher every 6 rounds, so as to introduce irregularity and resistance to Slide attacks.

The security of Camellia against the Square attack is discussed by Yeom *et al.* in [627]. A 4-round distinguisher allows for a Square attack (see Sect. 2.2.3.15), and four-round Camellia can be attacked by guessing a one byte subkey and using 2^{16} chosen plaintexts. This attack may be extended up to 9 rounds including the first FL/FL^{-1} layer by considering the key schedule. In [578] Shirai *et al.* discuss the security of Camellia against differential and linear attack, and the security is evaluated against the upper bounds of maximum differential characteristic probability (MDCP) and maximum linear characteristic probability (MLCP), calculated by determining the least numbers of active S-boxes, found by search. An evaluation method for truncated differential and linear paths is used to discard wrong paths. Using the above techniques, tighter upper bounds on MDCP and MLCP were found for reduced-round Camellia. Consequently, 10-round Camellia without FL/FL^{-1} has no differential and linear characteristic with probability higher than 2^{-128}. Shirai developed these attacks further in [577], proposing differential and linear attacks on Camellia without FL/FL^{-1} layers, and boomerang and rectangle attacks on Camellia with FL/FL^{-1} layers. The search complexity for the attacks of [577] is reduced by distinguishing between dependent and independent variables in the multi-round characteristics. Shirai obtains a differential attack on 11 rounds without FL/FL^{-1} layers using an 8-round characteristic, and a linear attack on 12 rounds without FL/FL^{-1} layers using a 9-round linear approximation. The boomerang and rectangle attacks are on 9 and 10-round Camellia , respectively, with FL/FL^{-1} layers, and use a technique developed by Biham *et al.* [68,69], where a cipher is described as $E_f \circ E_1 \circ E_0 \circ E_b$, such that the FL/FL^{-1} layer occurs between E_1 and E_0. Truncated and impossible differential cryptanalysis of Camellia (without FL/FL^{-1} functions) is described by Sugita *et al.* in [602], improving on the best known truncated and impossible differential cryptanalysis. A 9-round bytewise characteristic is shown that may lead to an attack on reduced-round Camellia without FL/FL^{-1} in a chosen plaintext scenario. A 7-round impossible differential is also shown by [602] on Camellia without FL/FL^{-1} functions. However, the designers of Camellia suspect that the FL and FL^{-1} functions will make attacking Camellia using impossible differentials difficult, since the functions change differential paths depending on key values. A Square attack on Camellia is proposed by He and Qing in [292], requiring 2^{112} encryptions and 13×2^8 plaintexts, over 6 rounds. The designers propose an 18-round cipher, but claim that even a 10-round variant is secure. However, they do not describe attacks on reduced variants of Camellia. A 3-round iterative differential characteristic with probability 2^{-52} has been found by Biham *et al.* in [67], which can be iterated to further rounds. They also found 5 additional 7-round characteristics with probability 2^{-104}. A 9-round variant of Camellia (without FL layers) was attacked using 2^{105} chosen plaintexts. Also, a one-round truncated iterative differential was found which over 7 rounds has probability 2^{-112} (assuming no FL layer). This can be extended to 8 rounds with probability 2^{-112}. This differential has the added advantage that it passes through the FL/FL^{-1} layers with probability 2^{-8}. Linear cryptanalysis of Camellia did not produce any efficient results — the best linear approximation of the S-boxes being $\frac{1}{2} \pm \frac{1}{16}$. A higher-order differential attack on 10 rounds of 256-bit key Camellia without FL rounds is performed by Kawabata and Kaneko in [339]. This also leads to an attack on 9 rounds for a 192-bit key and an attack on 8 rounds for a 128-bit key.

A recent embedding by Murphy and Robshaw of the AES, Rijndael, in a larger block cipher, the Big Encryption System (BES) [452, 454] has led to questions regarding future potential attacks on the AES. Camellia uses the same x^{-1} function for the S-box as Rijndael, and hence is open to the same form of attack. However, Camellia also inserts FL and FL^{-1} layers every six rounds, and an initial estimate of the extra complexity needed to overcome these layers for a BES-style redescription is at most 2^{16}. In short, if an attack using BES and a system of overdefined quadratic equations was ever successful on AES, then it might also be quite successful on Camellia. BES is discussed further in the section on Rijndael-128 of this report. As discussed in the security analyses for KHAZAD and MISTY1, Biryukov and De Cannière [82] compare minimal systems of multivariate polynomials which completely define certain block ciphers, including Camellia. As pointed out in [165], the Camellia (and Rijndael) S-box can be described by a system of 23 quadratic equations in 80 terms. Quadratic representations are utilised in [82] and, along with a set of linear equations to define the linear layers, the whole of the block cipher can be described by 5104 equations in 2816 variables using 14592 linear and quadratic terms. Similarly, the key schedule can be described by 1120 equations in 768 variables using 3328 linear and quadratic terms. In total [82] estimates that 6224 equations in 3584 variables using 17920 linear and quadratic terms are required. This gives the number of free terms as the number of terms minus the number of equations (See Sect. 2.2.3.17). For Camellia this is 11696 free terms. Roughly the same figures occur for Rijndael.

Finally, Fuller and Millan [245] recently observed that the bit-output functions of the S-box which is x^{-1} over GF(2^8) are all affine transformations of the same function. This observation was given in relation to the Rijndael S-box, but Camellia uses essentially the same S-box. This observation of [245] suggests a potential extra hardware saving for the Camellia S-box, although this may later also be seen as a security weakness (see Sect. 2.9.2).

2.5.2 RC6

2.5.2.1 The design

We consider here the version of RC6 [324] which takes 128-bit plaintext and key-lengths from 0 to 256 bytes, although the most useful variants may be versions with 16, 24, or 32-byte keys. 20 rounds are recommended. RC6 is quite a simple cipher to describe and it is a natural progression from the cipher RC5 which has undergone cryptanalysis for around 8 years without serious breaks or major security weaknesses being identified (although it is interesting to note that a key was recently found for 64-bit RC5 using a distributed internet exhaustive key search (see Sect. 2.2.3.1) involving over 300000 people over 5 years — the aim of a distributed internet search is to partition the exhaustive key search problem into many small subproblems which are then solved independently by participants in the search). However, RC5 was broken in a series of three papers which each improved the previous paper's result by a factor of about 1000 [85,336,365]. These attacks are generally based on the fact that the *rotation amounts* in RC5 do not depend on all the bits in a register. These attacks led to a redesign of RC5 into RC6 prior to submission of RC6 to the AES. The designers [324] claim that RC6 is an improvement on RC5, because of the introduction of fixed rotations and an extra quadratic function to enhance diffusion and resistance to differential and linear cryptanalysis. The key schedule is inherited from RC5, is quite involved and is considered strong. The cipher is fully-parameterised so that mini-versions (e.g. 4-bit, 8-bit, 16-bit) can be implemented and analysed, and this helps for the security analysis of the full 128-bit cipher. For a 128-bit block size, the word size is $w = 128/4 = 32$ bits, and $r = 20$ rounds. RC6 uses a 32-bit multiplication, mod 2^w, to enhance security, and therefore benefits greatly from software/hardware implementations with optimised multiplication. A round in RC6 is a bit like a round in DES, where half of the data is updated by the other half, and then the two halves are swapped. In fact, [568] reconfigures RC6 as a Feistel-like cipher by swapping the B and C registers. The full encryption procedure is as follows:

```
Input:   Plaintext in $A, B, C, D$
Output:  Ciphertext in $A, B, C, D$
Key:     $S[0, 1, \ldots, 2r + 3]$, $r$ rounds
Procedure:
  $B = B + S[0]$                                 $+$ is addition mod $2^w$
  $D = D + S[1]$
  for $i = 1$ to $r$ do
  {
    $t = (B \times (2B + 1)) <<< \log_2(w)$      $\times$ is mult. mod $2^w$
    $u = (D \times (2D + 1)) <<< \log_2(w)$      $<<<$ is rotate left
    $A = ((A \oplus t) <<< u) + S[2i]$           $\oplus$ is bitwise addition mod 2
    $C = ((C \oplus u) <<< t) + S[2i + 1]$
    $(A, B, C, D) = (B, C, D, A)$
  }
  $A = A + S[2r + 2]$
  $C = C + S[2r + 3]$
```

Decryption is similar to encryption, but not the same — for instance addition is replaced with subtraction, and rotate right operations are used.

Two constants are added into the key schedule. The constants are P_w and Q_w, which are binary expansions of $e - 2$ and $\phi - 1$, respectively, where e is the natural logarithm, and ϕ is the Golden Ratio. The key schedule is as follows:

```
Input:   User-supplied $b$ byte key preloaded into the $c$-word
         Array $L[0, \dots, c - 1]$
         Number $r$ of rounds
Output:  $w$-bit round keys $S[0, \dots, 2r + 3]$
Procedure:
  $S[0] = P_w$
  for $i = 1$ to $2r + 3$ do
    $S[i] = S[i - 1] + Q_w$
  $A = B = i = j = 0$
  $v = 3 \times \max\{c, 2r + 4\}$
  for $s = 1$ to $v$ do
  {
    $A = S[i] = (S[i] + A + B) <<< 3$
    $B = L[j] = (L[j] + A + B) <<< (A + B)$
    $i = (i + 1) \mod (2r + 4)$
    $j = (j + 1) \mod c$
  }
```

2.5.2.2 Security analysis

No security flaws have been found for RC6 and it has resisted cryptanalysis during and after the AES process. It is specified over 20 rounds although the fact that 15 of the 20 rounds were broken meant that NIST did not consider that 20 rounds gave a sufficient security margin.

The designers performed extensive differential and linear cryptanalysis of RC6 in [143] and conclude that RC6 is highly resistant to both attacks. Table 2.5 shows non-exhaustive estimates for the numbers of plaintexts necessary to attack RC6 as quoted in [143], although some of the figures on the right-hand side of the table exceed 2^{128} which is the total number of possible plaintexts.

Table 2.5. Differential/Linear Cryptanalysis of RC6 [143]

	# Rnds				
attack	8	12	16	20	24
diff. crypt.	2^{56}	2^{117}	2^{190}	2^{238}	2^{299}
lin. crypt.	2^{47}	2^{83}	2^{119}	2^{155}	2^{191}

The two most obvious notions of difference for RC6 are XOR and subtraction, mod 2^w, and [143] claims that difference using subtraction gives a more effective attack. An aim of this type of attack will be to try to use differences that do not provide different rotation amounts, thereby minimising the avalanche effect. The purpose of the introduction of the quadratic function $f(x) = x(2x+1)$ in RC6, a feature not in RC5, is to increase dependency of the data-dependent rotations, and to speed-up diffusion. Both these effects improve the resistance of the cipher to differential and linear cryptanalysis. The most manageable differential and linear attacks use one-bit characteristics as, although multiple-bit characteristics do exist with higher probability, it is difficult to connect them up into a differential or linear trail. The most effective type of linear approximation appears to exploit approximation across the data-dependent rotations. The paper of [143] identifies such approximations of the form $A \cdot \Gamma_a = B \cdot \Gamma_b \oplus C \cdot \Gamma_c$ for the data-dependent rotation, $A = B <<< C$, where single-bit masks work best across the integer addition and the quadratic functions. Moreover, the best approximation across $y = (f(x) <<< 5)$ is $y[5] = x[0]$ with probability 1. In order to further protect against the theoretical risk of multiple linear approximations and linear hull attacks, the designers propose a minimum number of 20 rounds for RC6. Table 2.6 provides figures for the estimated number of plaintexts needed for a potential multiple linear approximation attack, with or without linear hulls [143].

Table 2.6. Multiple Linear Cryptanalysis of RC6 [143]

	# Rnds				
attack	8	12	16	20	24
basic linear attack	2^{62}	2^{102}	2^{142}	2^{182}	2^{222}
+ mult. linear approx.	2^{51}	2^{91}	2^{131}	2^{171}	2^{211}
+ mult. linear approx. + linear hulls	2^{47}	2^{83}	2^{119}	2^{155}	2^{191}

Jonsson and Kaliski construct 6-round characteristics for RC6 [324], so as to attack 8 rounds, and this leads to 2^{76} chosen plaintext pairs for differential cryptanalysis of 8 rounds, and 2^{60} known plaintexts for linear cryptanalysis of 8 rounds. Shimoyama *et al.* [568] develop further the multiple linear attack approach for RC6 with a 256-bit key, where they generalise the Piling-Up Lemma using a certain *Matrix Representation*. They achieve a 14-round key recovery attack using 2^{120} known plaintexts and 2^{186} round computations. They also achieve an 18-round key recovery attack on a fraction of 2^{-90} of the keys with 2^{127} known plaintexts, 2^{64} memory, and 2^{193} round computations.

As mentioned previously, the key schedule for RC6 is the same as that for RC5. No weak keys or related-key attacks have been found for RC5, perhaps owing to the key schedule being quite complicated — the design of the key schedule is somewhat incompatible with the encryption structure of RC5 or RC6.

One of the most effective attacks on RC6 is by Knudsen and Meier [366] showing that, by means of a chosen-plaintext attack, RC6 can be distinguished from a random permutation with up to 15 rounds, and for 1 in 2^{80} keys up to 17 rounds can be distinguished. Moreover, key-recovery attacks can be mounted on RC6 with up to 15 rounds faster than exhaustive search for the key. To do this, [366] considers two round iterations of a form quite different from [143]. Instead of exploiting bitwise linear approximations, input-output dependencies are considered by fixing the least-significant 5 bits in the first and third words of the input block, A and C. The correlations of the corresponding two 5-bit integer values at the output every two rounds later can then be effectively measured by χ^2 tests. This gives a considerable improvement over the basic linear attack. The attack of [366] has similarities to that considered by Baudron *et al.* [42], and by Gilbert *et al.* [257]. The approach of [366] is motivated by the fact that the least significant $\log(w) = 5$ bits in A and C are not changed by the XOR and data-dependent rotation, if both rotation amounts are zero. Small *negative* rotation events (e.g. $<<<$ 30, or $<<<$ 31) are also exploited in [366]. The paper [366] also analyses mini-versions of RC6 to verify the experimental evidence, and the 15-17 round attacks are extrapolated from experimental evidence computed on up to 6 rounds of RC6, where it is estimated that 2^{14} more plaintexts are needed in going from s to $s + 2$ rounds. Furthermore weak key classes are exploited for RC6 in [366], and these exist because RC6 uses addition mod 2^{32}, which introduces carry propagation into the cipher. The results of these χ^2 attacks on RC6 provide further evidence for the strength of RC6, as the results suggest that the χ^2 attacks tend to attain the same level of complexity as previous differential and linear attacks [114], and other attacks [346].

In summary, the attacks currently known on RC6 suggest that 20 rounds is secure, although the security margin may be somewhat narrow.

2.5.3 AES (Rijndael)

2.5.3.1 The design

128-bit plaintext block Rijndael has a 128, 192, or 256-bit key over 10, 12, or 14 rounds, respectively. Rijndael [181] has recently been selected as the Advanced Encryption Standard AES and has therefore been subject to intensive study in the last few years. Rijndael is a variant of the Square block cipher [178]. The cipher is non-Feistel, and emphasises a combination of optimal diffusion [182] with optimal nonlinearity for the S-box. The key is added linearly via XOR. Encryption is similar but not identical to decryption. In software, decryption has exactly the same speed as encryption, except on 8-bit machines when decryption is slightly slower. In hardware the speed of encryption and decryption operations is the same, but decryption requires slightly more hardware. A round of Rijndael can be written as follows,

```
Round(State,RoundKey)
{
    ByteSub(State)
    ShiftRow(State)
    MixColumn(State)
    AddRoundKey(State,RoundKey)
}
```

ByteSub is the optimally nonlinear 8×8-bit S-box operation, which is x^{-1} over GF(2^8), followed by an affine transformation. ShiftRow is a bytewise permutation over GF(2^8)^4, MixColumn is a 4-byte *bytewise* affine transform over GF$(2)^{32}$, in fact an MDS code, and AddRoundKey is the XOR of the key onto the output of the round. The diffusion layer is a linear transformation and comprises ShiftRow and MixColumn, and it can be shown that this diffusion has an optimum branch number of 5, and activates at least 25 bytes over 4 rounds.

The key schedule of 128-bit Rijndael over 10 rounds takes in a 128-bit key and generates $128 \times 11 = 1408$ round key bits in the form of 11 128-bit subkeys, one for each round, and one at the beginning [470]. The initial subkey is set to the key, and the remainder of the subkeys are generated iteratively using the following Key Expansion algorithm for a 128-bit key,

```
for $i = 0$ to 3
  $W[i] = \text{Key}[i]$
for $j = 4$ to 40 (in steps of 4)
{
  $W[j] = W[j-4] \oplus \text{SubWord}(\text{Rotl}(W[j-1])) \oplus \text{Rcon}[j/4]$
  for $i = 1$ to 3
    $W[i+j] = W[i+j-4] \oplus W[i+j-1]$
}
```

where 'Rotl' means rotate left, 'Rcon' means round constant, $W[0, 1, \ldots, 10]$ is an array of 32-bit subkeys, and SubWord() is a function that takes a four-byte input word and applies the AES S-box to each of the four input bytes to produce the output word. Note that the key-schedule uses the S-box of the enciphering process.

2.5.3.2 Security analysis

Rijndael has been selected by the NIST as the new AES. No security flaws have been found. It has been chosen for Phase II of NESSIE as a benchmark against which to evaluate other submissions.

Rijndael attracted much public attention after it became the AES [74, 82, 146, 158, 164, 165, 178, 227, 228, 245, 258, 342, 367, 371, 407, 433, 449, 452, 454, 495, 620]. However, this is also because the cipher is particularly elegant and easy to describe, using highly algebraic components — its simplicity invites analysis, and this was one of the philosophies on which Rijndael was based. The use of optimal nonlinearity followed by optimal diffusion using an MDS-based linear transformation helps to give high resistance to differential and linear attacks for Rijndael.

One of the most successful attacks against Rijndael is the *Square attack* [181], which is a form of integral cryptanalysis [371] originally used by Daemen *et al.* to attack the Square block cipher [178]. The Square attack exploits the relatively slow avalanche of the sparse affine mapping (linear mixing along rows and columns of some matrix followed by subkey addition). It is a chosen plaintext attack of up to 6 rounds on key sizes 128, 192, and 256, which makes use of a distinguisher on 3 Rijndael rounds. For this distinguisher some input bytes are chosen to take all 256 values over 256 input chosen plaintexts so that one can predict the bytes which will take all 256 values (active bytes), the bytes which are constant (passive bytes), and the bytes which are balanced ($\oplus$ sum is zero) in future rounds. The Square attack requires 2^{32} plaintexts, 2^{72} cipher encryptions, and 2^{32} memory. Primarily, the Square attack follows the balance of certain data bytes as they progress through the cipher.

No attack is known on more than 7-8 rounds of Rijndael [227, 258, 407], the best attack being the collision attack by Gilbert and Minier which breaks up to 7 rounds [258] for key sizes 128, 192, and 256. This attack requires 2^{32} chosen plaintexts and, for key sizes 192 and 256, requires a time complexity of about 2^{140}. For key size 128 the complexity required is marginally less than exhaustive search. The attack makes use of a 4-round distinguisher which exploits, by means of the birthday paradox, the existence of collisions between some partial functions introduced by the cipher. The attack by Lucks [407] extends the Square attack to Rijndael variants with 192 and 256-bit keys, and achieves an attack on seven rounds by simply guessing the 16 bytes of the last round key and exploiting minor weaknesses in the key schedule. The attack requires 2^{32} chosen plaintexts and 2^{176} or 2^{192} encryptions for 192 and 256-bit keys, respectively. The attacks of Ferguson *et al.* [227] include a 6-round improved Square attack with time complexity about 2^{44} which is an improvement on the original 2^{72}, at the price of about $6 \cdot 2^{32}$ plaintexts. The improvement makes use of a *partial-sum* technique to reduce the workfactor over the original Square attack. There is also an improvement on the 7-round attack of [407] in [227] which requires 2^{155} or 2^{172} encryptions for 192 or 256-bit keys, respectively. There is also an alternative extension to 7 rounds by Ferguson *et al.* [227] that can break all key sizes with encryption complexity 2^{120} but which requires virtually the entire codebook of plaintexts ($2^{128} - 2^{119}$ plaintexts). Moreover, one may even break 8 rounds of Rijndael with $2^{128} - 2^{119}$ plaintexts, requiring 2^{188} or 2^{204} encryptions for 192 or 256-bit keys, respectively [227].

After two rounds, Rijndael provides full diffusion, i.e. every state bit depends on all state bits two rounds ago. This is due to the uniform structure of Rijndael, and the high diffusion. The designers [181] claim that no 4-round differential characteristic exists with probability greater than 2^{-150}, and no 4-round linear characteristic exists with a correlation greater than 2^{-75}. An analysis of the propagation of activity patterns in [181] leads to the conclusion that any linear or differential characteristic over 4 rounds must activate at least 25 bytes. Improved upper bounds are given by Keliher *et al.* on the maximum average linear hull probability for Rijndael by noting that the linear hull effect for Rijndael is significant [342]. An upper bound on the probability is given, namely 2^{-75} for 7 rounds. In a subsequent paper the same authors improve on this result by taking into account more details of the linear characteristics of the Rijndael S-box. They improve their upper bound on maximum average linear hull probability to 2^{-92} for 9 rounds. Related work by Ohkuma *et al.* [495] upper bounds the maximum average differential and linear hull probabilities by 2^{-96} for 4 rounds of Rijndael. In [495] it is also shown how to represent Rijndael as a two-level nested Substitution Permutation Network (SPN) where each level uses an MDS layer for diffusion. There is also an impossible differential attack on 5 rounds by Biham and Keller, requiring $2^{29.5}$ plaintext-ciphertext pairs and 2^{31} time [74], and an extension to 6 rounds has been presented by Cheon *et al.* [140].

In [620] Wernsdorf shows that the round functions of Rijndael generate the alternating group over the set $\{0,1\}^{128}$, eliminating some potential weaknesses of Rijndael, e.g. non-trivial factor groups (the alternating group is a large, simple, primitive and $(2^{128}-2)$-transitive group).

With respect to the key schedule of Rijndael, Ferguson *et al.* [227] exploit weaknesses in the expanded key of Rijndael by proposing a related-key attack on 9 rounds which is a variant of the Square attack and use 256 related keys that differ in a single byte in the fourth round key. Plaintext differences are used to cancel out earlier round key differences, resulting in three bytes at the end of round 6 that sum to zero when taken over the 256 encryptions. Key bytes of the last three rounds are then guessed and used to compute backwards from the ciphertext to detect this property. This 9-round attack on Rijndael with 256-bit keys requires 2^{77} plaintexts under 256 related keys, and 2^{224} encryptions.

May *et al.* [433] provide a modified key-schedule for Rijndael with improved diffusion and nonlinearity, whilst keeping a reasonably fast speed for the key-schedule. More generally, the key schedule has a much slower diffusion than the cipher and contains relatively few nonlinear elements.

It has recently been observed by Fuller and Millan [245] that the output functions of the Rijndael S-box are all affine transformations of the same function. This observation suggests a potential extra hardware saving for the Rijndael S-box, although, perhaps more importantly, this may later also be seen as a security weakness (see Sect. 2.9.2). In other words, let b_i and b_j be two distinct output bit functions of the Rijndael S-box. Then we can always find a Boolean matrix, $\mathbf{A}$, and a Boolean vector, $\mathbf{B}$, such that,

$$b_i(x) = b_j(\mathbf{A}x + \mathbf{B})$$

This surprising result means that the Rijndael block cipher only uses *one* Boolean function from eight bits to one bit (used 128 times in each round). Since [245] was posted, Youssef and Tavares [629] have proved this result by making use of dual bases over GF(2^n) and trace functions, and showed that the result holds for any S-box based on a bijective monomial. They also extended the result to show that all *coordinate* (bit) functions of the Rijndael round function are equivalent under affine transformation of the input to the round function. Following on from [245], Biham [62] has shown that affine relationships exist between the output bits of many S-boxes, not just the Rijndael S-box (see Sect. 2.9.2). Another simplified algebraic formulation for Rijndael was presented by Ferguson *et al.* in [228]. This paper argues that the security of Rijndael is based on a new hardness assumption for the solution of an algebraic formulation of the type derived in [228].

A recent redefinition of Rijndael has raised many questions regarding the security of the cipher. The *Big Encryption System* (BES) has been defined by

Murphy and Robshaw in [452], where the AES can be regarded as being identical to the BES with a restricted message space and key space. Whereas the AES uses operations over GF(2^8) and GF(2), the BES only uses operations over GF(2^8) and this raises the question of algebraic attacks on AES. The key idea for BES is to map every GF(2^8) byte, a of AES to $(a^{2^0}, a^{2^1}, \ldots, a^{2^7})$ for BES, where this vector comprises a and its conjugates over GF(2^8). The central application of this mapping is to convert the GF(2) linear operation in the S-box of AES (an 8×8 binary matrix) to an 8×8 matrix mapping over GF(2^8) which linearly operates on a and its conjugates. This ensures that all operations of BES occur in GF(2^8), and one can write the ith round function of BES as:

$$\text{Round}_B(b\, k_i) = M_B(b^{(-1)}) + k_i$$

where b is the input to the round, M_B is a matrix with elements from GF(2^8) and k_i is the ith round key in BES. It should be noted that the key schedule can also be written completely over GF(2^8). M_B can always be converted to its Jordan Form, R_B, where $R_B = P_B^{-1} \cdot M_B \cdot P_B$ is a particularly well-structured representation for an AES round. The paper [452] identifies Related-Key and Differential attacks on BES which exhibit characteristics with probability one through a round, in spite of the high diffusion. But these attacks are not directly applicable to AES as they do not preserve the conjugacy relation which is necessary for the inverse mapping back to AES. One of the most interesting future lines of inquiry for BES is to combine it with the techniques of the type suggested by Courtois and Pieprzyk [158,164,165] for solving the type of multivariate quadratic systems that arise from block ciphers. A preliminary analysis [452,454] of the complexity of an attack based on the estimates of Courtois and Pieprzyk [165] suggests an attack considerably faster than exhaustive key search. However, there are inaccuracies in these estimates [145,146,454] It has been noted by Knudsen and Raddum [367] that its mathematical elegance makes Rijndael more vulnerable to a devastating attack as there are no *random-looking* elements in the cipher.

As discussed in the security analyses for KHAZAD, MISTY1, and Camellia, Biryukov and De Cannière [82] compare minimal systems of multivariate polynomials which completely define certain block ciphers, including Rijndael-128. As pointed out in [165], the Rijndael S-box can be described by a system of 23 quadratic equations in 80 terms. It is estimated in [82] that the block cipher and key schedule can be described by 6296 equations in 3296 variables using 19296 linear and quadratic terms. For Rijndael-128 this means that there are 13000 free terms (see Sect. 2.2.3.17). Roughly the same figures occur for Camellia-128. Note that there are 37 quadratic equations in total for the AES S-box, whereas a randomly-chosen 8-bit S-box would expect to have zero quadratic equations associated with it. However, it should be noted that this system of equations is far larger than the multivariate quadratic system provided with BES.

Recent work by Barkan and Biham [36] has developed the concept of the *Dual Cipher*. Specifically, let the ciphertext, C, be the result of encrypting the plaintext, P, using the cipher, E, where E is dependent on the secret key, k. We write this as $C = E_k(P)$. Then [36] has stated that, given any invertible functions, f, g, and h, E' is a dual cipher to E if

$$\forall P, k \quad f(E_k(P)) = E'_{g(k)}(h(P)).$$

The dual cipher, E', is equivalent to the original cipher, E, in all aspects. One can therefore analyse and attack the dual cipher instead of the original cipher. Moreover, one can actually implement the dual cipher instead of the original cipher. For Rijndael, [36] demonstrates the existence of *Square-Dual* ciphers in the sense that every component of the original Rijndael cipher has been squared (or conjugated). Thus, Rijndael has 7 dual ciphers of this form. Moreover, one can replace the irreducible polynomial used by Rijndael with another one, and this too leads to another set of dual ciphers — 240 in total. Also, the components of Rijndael can be replaced with their logs to give a set of *Log-Dual* ciphers, 128 in total. Finally, [36] shows that by straightforward modifications of Rijndael, one can create Self-Dual ciphers which exhibit cryptographic weakness although this does not imply weakness in Rijndael itself. It should be noted that this dual cipher analysis also holds for KHAZAD, and ANUBIS, and can also be applied to Camellia and SAFER++.

2.5.4 SAFER++ (128-bit block)

2.5.4.1 The design

SAFER++ [420] is a development from the existing SAFER family of ciphers and uses a combination of substitution and linear transformation to achieve confusion and diffusion, respectively. We here consider the normal and high versions which take in 128-bit plaintexts and require 128-bit or 256-bit keys, respectively. The designers recommend 7 rounds for the 128-bit key version, and 10 rounds for the 256-bit key version.

Figure. 2.17 shows an encryption round for SAFER++ (128-bit block). This figure should be compared with Fig. 2.9 for SAFER++ (64-bit block). It is evident that, whereas SAFER++ (64-bit block) requires zero-padding and merging of the round input, prior to and following the linear transformation, SAFER++ (128-bit block) does not require this, as the round input is already of size 128 bits. For a more detailed discussion, please refer to the section commenting on SAFER++ (64-bit block). Its design has many interesting properties.

2.5.4.2 Security Analysis

No security flaws have been found with SAFER++ (128-bit block) [420] and it has many similarities to SAFER++ (64-bit block). One weakness found in previous versions of the SAFER family by Knudsen was in the key-schedules [351,356], but these weaknesses have been dealt with in SAFER++ (128-bit block). Other pre-NESSIE attacks on the SAFER family include truncated differentials by Knudsen and Berson [362], and Murphy [451] identifies a potential algebraic weakness regarding the existence of invariant $\mathbb{Z}$-modules within the PHT layer. These modules and their cosets are not diffused by the PHT layer, and so provide a way to cope with diffusion in SAFER, regardless of the key schedule. One attack in [451] which used this property enabled a projection of the message/ciphertext space onto a 4-byte $\mathbb{Z}$-submodule so that the probability of any message projection

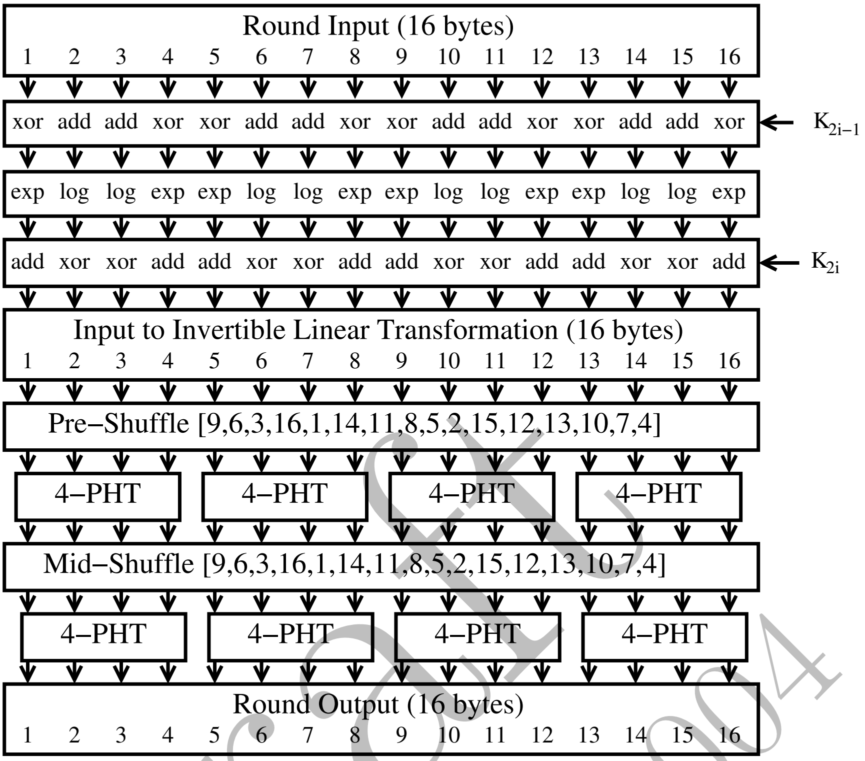

Round Input (16 bytes)
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
xor add add xor xor add add xor xor add add xor xor add add xor
K2i−1
exp log log exp exp log log exp exp log log exp exp log log exp
add xor xor add add xor xor add add xor xor add add xor xor add
K2i
Input to Invertible Linear Transformation (16 bytes)
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
Pre−Shuffle [9,6,3,16,1,14,11,8,5,2,15,12,13,10,7,4]
4−PHT
4−PHT
4−PHT
4−PHT
Mid−Shuffle [9,6,3,16,1,14,11,8,5,2,15,12,13,10,7,4]
4−PHT
4−PHT
4−PHT
4−PHT
Round Output (16 bytes)
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

Fig. 2.17. Encryption Round for SAFER++ (128-bit block).

giving any ciphertext projection is independent of $\frac{1}{4}$ of the key bytes. This result, along with the results of [351] led to a change in the SAFER key schedule. In [619] Wernsdorf shows that the round functions of SAFER++ (128-bit block) generate the alternating group over the set $\{0, 1\}^{128}$, eliminating some potential weaknesses of SAFER++ (128-bit block), e.g. non-trivial factor groups (the alternating group is a large, simple, primitive and $(2^{128} - 2)$-transitive group). The designers claim that one of the main reasons for the security of the cipher against differential and linear cryptanalysis is the high diffusion PHT layer. The designers conclude that SAFER++with six or more rounds is secure against differential cryptanalysis, and with two and a half or more rounds is secure against linear cryptanalysis. Note also that, in [356], Knudsen identified a differential such that, if X is the exponentiation function of SAFER, then $X(a)+X(a+128) = 1($ mod 256) holds with probability 1, and Nakahara has extended this observation to relate not only the msb of the input to the lsb of the output, but also to the 2nd lsb of the output [327, 456]. Moreover, in [327, 456], Nakahara generalises to other exponents, not just 45. Recently Nakahara *et al.* [459] applied techniques that were first used to more generally attack the SAFER family in [458], and [459] showed that, for the 256-bit key version, up to 3.75 rounds can be attacked by linear cryptanalysis, with less effort than exhaustive search, requiring

2^{81} known plaintexts. The reason for this discrepancy between the two and a half rounds claimed by the designers and the three to four rounds claimed by [459] is largely because the designers restricted themselves to homomorphic attacks, whereas [459] applies strictly non-homomorphic attacks where some key bits are assumed fixed. Therefore the results of [459] must be seen in the context of a weak-key class. Table 2.7 gives a summary of the complexity of linear attacks on SAFER++ [459]. One important point to note with regard to the linear attack of [459] is that it identifies a surprisingly small byte and bit branch number for the PHT diffusion layer. Whereas the designers of SAFER++use the fact that the lowest row weight of the matrix M is 10, implying a high diffusion, a more detailed examination of the matrix reveals a byte branch number ≤ 7 and a bit branch number ≤ 5. Moreover the S-box defined by $y = 45^x \mod 257$ contains a linear relationship which holds with probability 1. In other words, we can write $y_0 = f(x_0, x_1, \ldots, x_6) + x_7$, where y_0 is one of the output bits of the S-box, and depends linearly on the input, x_7. This moderated diffusion combined with a high linear characteristic is what enables this linear attack on three to four rounds of SAFER++. Table 2.7 supplements Table 2.4 by stating the complexity of linear attacks on SAFER++that apply to 256-bit keys.

Table 2.7. Complexity of Linear Attacks on SAFER++

# Rounds Attacked	# Known Plaintexts	# Subkey Bits Explored	Attack Complexity	Fraction of Keys
2	2^5	37	2^{42} ♣	
3.5	2^{33}	88	2^{121} ♣	2^{-6}
3.75	2^{81}	97	2^{176} ♠	2^{-13}
	2^{91}	76	2^{167} ♠	2^{-11}

♣ The attack applies to all key sizes defined for SAFER++.
♠ This attack applies to 256-bit keys only.

It is also interesting to note that affine relationships exist between the bit outputs, both for exp45 and log45 [62] (see Sect. 2.9.1).

Impossible Differential and Square attacks have also been demonstrated on SAFER++ (128-bit block) by Nakahara *et al.* [460], and Nakahara [456], respectively, over 2.75 to 3.25 rounds. Table 2.8 details these attacks along with further attacks on similar ciphers from the SAFER family.

SAFER++ (128-bit block) is a Substitution-Permutation Network (SPN), and five-layer SPN's are susceptible to structural analysis leading to integral or multiset attacks. Piret [520] describes a (classical) integral distinguisher over 2 rounds of SAFER++(in the encryption direction). This allows a practical attack against 3 rounds of SAFER++ (128-bit block), as well as attacks on 4 rounds of SAFER++ (128-bit block) and SAFER++ (256-bit block) (always without the last key addition layer), under the chosen-plaintext hypothesis. As a side result, Piret proves that the byte-branch number of the linear transform of SAFER++is precisely 5. Concrete figures for these attacks are given in Table 2.9.

Table 2.8. Attacks by Nakahara *et al.* for SAFER++ (128-bit block) and Similar Designs

Cipher	Attack	#Rounds	Data	Memory	Time	Ref
SAFER++	Imp. Diff.	2.75	2^{64} CP	2^{97}	2^{60}	[460]
	Square	3.25	$2^{9.6}$ CP	$2^{9.6}$	2^{70}	[456]
	Linear	3.25	2^{81} KP	2^{81}	2^{101}	[459]
SAFER SK$_{128}$	Imp. Diff.	2.75	2^{39} CP	2^{58}	2^{64}	[460]
	Square	3.25	$2^{10.3}$ CP	$2^{10.3}$	2^{38}	[456]
	Linear	4.75	2^{63} KP	2^{63}	2^{90}	[458]
SAFER+$_{128}$	Imp. Diff.	2.75	2^{64} CP	2^{97}	2^{60}	[460]
	Square	3.25	$2^{9.6}$ CP	$2^{9.6}$	2^{70}	[456]
SAFER+$_{192}$	Linear	3.25	2^{100} KP	2^{100}	2^{137}	[459]
SAFER+$_{256}$	Linear	3.75	2^{81} KP	2^{81}	2^{176}	[459]

KP: known plaintext, CP: chosen plaintext

Table 2.9. Piret's Integral Attacks on SAFER++ (128-bit block).

Key Size	#Rounds	#Plaintexts	Time Compl.	Space Compl.
128	3	2^{16}	2^{16}	2^{16}
128	4	2^{64}	2^{112}	2^{64}
128	4	2^{64}	2^{120}	2^{16} (different tradeoff)
256	4	2^{64}	2^{144}	2^{64}

Even stronger multiset attacks have recently been presented by Biryukov *et al.* [84]. The general method can be applied to any SPN network with incomplete diffusion, and the method of [84] is also a collision attack, inspired by the attacks of Gilbert and Minier on Rijndael [258]. The multiset attacks of [84] can break up to 4.5 rounds of SAFER++ (128-bit block) in 2^{48} chosen plaintexts and 2^{94} steps. Biryukov *et al.* [84] also present a Boomerang attack on SAFER++ (128-bit block) which exploits the incomplete diffusion of SAFER++[128] and also certain special properties of the SAFER S-boxes. In this way they have constructed a 5 round attack on SAFER++[128] using 2^{75} chosen plaintexts/ adaptive chosen plaintexts and 2^{75} time complexity. The attack completely recovers the 128-bit secret key of the cipher and has a 86% probability of success. The attack can be extended to 5.5 rounds by guessing 46 bits of the secret key. The attacks of [84] are summarised in Table 2.10.

2.6 Large block ciphers considered during Phase II

The large block ciphers selected for phase II of NESSIE were RC6, Rijndael, SHACAL-1, and SHACAL-2. None of these ciphers has been broken so the following security evaluation identifies weaknesses that occur in reduced-round versions of the ciphers, and identifies weaknesses that may lead to more effective attacks in the future. We first describe each cipher in some detail, along with the most important attacks known on each cipher. Note that the algorithms given here

Table 2.10. Multiset and Boomerang Attacks on SAFER++ (128-bit block) by Biryukov *et al.*

Attack	Key Size	#Rounds	Data	Workload	Memory
Multiset	128	4	2^{48}	2^{70}	2^{48}
Multiset	128	4.5	2^{48}	2^{94}	2^{48}
Boomerang	128	4	2^{41}	2^{41}	2^{41}
Boomerang	128	5	2^{75}	2^{75}	2^{48}
Boomerang	128	5.5	2^{121}	2^{121}	2^{48}

Workload expressed in equivalent number of encryptions.

are not complete specifications, but references are given to complete specifications which may be found on the NESSIE website. After discussing each cipher we summarise and compare some of the distinguishing features of the ciphers, identifying potential weaknesses and noting the best-known attacks, as shown in Table 2.18 of Sect. 2.9.3.

2.6.1 RC6

2.6.1.1 The design

We consider here the version of RC6 [324] which takes 256-bit plaintexts and key-lengths from 128 to 256 bits. For the 128-bit plaintext version the designers recommended 20 rounds, but for the 256-bit plaintext version, the recommended number of rounds is not specified. A detailed discussion of RC6 is given in the section on 128-bit block ciphers.

2.6.1.2 Security analysis

Until very recently, no security flaws have been found in RC6, and the 128-bit block variant of RC6 on which the 256-bit variant is based has been well studied.

Recently, Knudsen has detected correlations in 256-bit block RC6 using the χ^2-attack method [359]. From tests on RC6 with 256-bit blocks and three rounds together with other test results, [359] is able to extrapolate an estimated requirement for the number of plaintexts needed for this attack up to 25 rounds. It is estimated that for $3 + 2s$ rounds a χ^2 test would distinguish 256-bit block RC6 from random using 2^{16+20s} plaintexts. This constitutes a successful attack up to 25 rounds where it is expected that the χ^2 attack will require only 2^{236} plaintexts. The attack exploits the least significant five bits in the words A and C of the input of one round, and investigates the statistics of the 10-bit integer obtained by concatenating the least significant five bits in the words A'' and C'' every two rounds later. This is motivated by the fact that the least significant five bits in A and C are not changed by the XOR and data dependent rotation if both rotation amounts are zero. More generally, one can expect a bias for amounts smaller than five, and these strong biases can be iterated over many rounds in the same way as linear approximations.

2.6.2 AES Variant (Rijndael-256)

2.6.2.1 The design

We here consider a variant of the NIST AES standard, 256-bit plaintext block Rijndael, with a 256-bit key over 14 rounds. Whereas 128-bit Rijndael uses 16 8-bit S-boxes per round, 256-bit Rijndael uses 32 8-bit S-boxes per round. A detailed discussion of Rijndael is given in the section on 128-bit block ciphers.

2.6.2.2 Security analysis

No security flaws have been found, and the 128-bit block variant on which it is based was selected as the AES and has been well-studied. Although similar in structure to the 128-bit block Rijndael, the 256-bit block variant still warrants a separate analysis as the byte alignments of this variant are different from those of the 128-bit block variant.

2.6.3 SHACAL-1

2.6.3.1 The design

SHACAL [281] is a 160-bit (20-byte) block cipher using a 0 to 512-bit (64-byte) key that is based on the well known Hash Function FIPS standard, SHA. SHACAL-1 is based on SHA-1, a more recent FIPS standard, which is a minor modification of SHA-0 in the message expansion. It is considered to have very fast implementations. In hash function mode, SHA takes a 512-bit message with a 128-bit initial value. In encryption mode (block cipher), the message becomes the key, and the initial value is replaced by the plaintext. SHA mixes group operations, $+ \bmod 2^{32}$ and XOR, with nonlinear logical functions. SHACAL-1 places the 160-bit plaintext in 5 concatenated 32-bit variables, A, B, C, D, E, and updates these five variables on each of 80 consecutive steps, so that the final ciphertext is contained in A, B, C, D, E after 80 steps. In the process, the 512-bit key is expanded to 2560 bits. SHACAL-1 is shown in Fig. 2.18, where ROL means *rotate left*, + means addition mod 2^{32}, and f is a different linear or nonlinear function for steps $\lfloor i/20 \rfloor$.

The encryption algorithm for SHACAL-1 is as follows.

```
Put the 160-bit plaintext in 32-bit variables $A\,B\,C\,D\,E$.
For 80 steps do
    $A^{i+1} = W^i + \mathrm{ROL}_5(A^i) + f^i(B^i\,C^i\,D^i) + E^i + K^i$.
    $B^{i+1} = A^i$.
    $C^{i+1} = \mathrm{ROL}_{30}(B^i)$.
    $D^{i+1} = C^i$.
    $E^{i+1} = D^i$.
```

where the W^i are 32-bit step keys, the K^i are round-dependent constants, and $f_i(X\,Y\,Z)$ is one of three functions defined below, where the function chosen is dependent on the round.

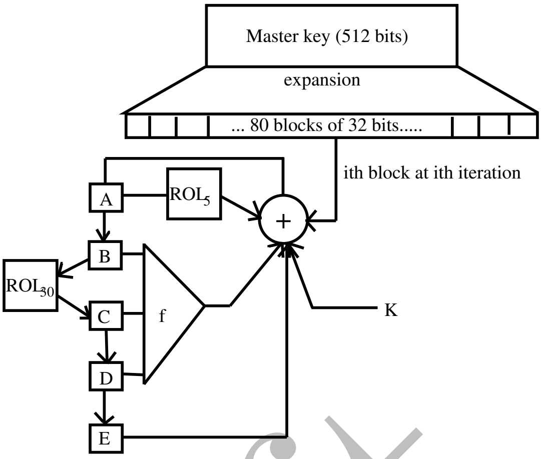


Fig. 2.18. Encryption for SHACAL-1

$$\begin{array}{ll}
f_{if} = (X \text{ AND } Y) \text{ OR } (\bar{X} \text{ AND } Z) & 0 \leq i < 20 \\
f_{xor} = (X \oplus Y \oplus Z) & 20 \leq i < 40, 60 \leq i < 80 \\
f_{maj} = ((X \text{ AND } Y) \text{ OR } (X \text{ AND } Z) \text{ OR } (Y \text{ AND } Z)) & 40 \leq i < 60
\end{array}$$

where $\oplus$ is bitwise XOR, AND and OR are both bitwise logical operations, and $\bar{*}$ is complement. Each set of 20 steps, $r \leq i < r + 20$ constitutes a round of the cipher. In order for SHACAL-1 to be invertible the final addition of the initial value, which occurs in the hash mode of SHA-1, is omitted.

The message expansion is different for SHA-0 and SHA-1. For SHACAL-1 the key schedule (message expansion) is linear, it expands the 512-bit key (Master Key) to 2560 bits, and can be described as follows,

– The Master key is a concatenation of 16 32-bit words: $[W^0|W^1|\ldots|W^{15}]$.
– $W^i = \mathrm{ROL}_1(W^{i-3} \oplus W^{i-8} \oplus W^{i-14} \oplus W^{i-16})$, $16 \leq i < 80$.

Keys shorter than 512 bits may be accomodated by padding the key input up to 512 bits.

2.6.3.2 Security analysis

NESSIE considers that the security margins of SHACAL-1 are very large. It also has the interesting property of being able to share most of the code of the SHA-1 hash function. The best attack known on SHA-0 is that of Chabaud and Joux [138] who obtain collisions using 2^{61} encryptions by tracking perturbations through the hash function in combination with differential masks. However, it was found by the authors of [138] that they could not extend the attack to SHA-1 because SHA-1 interleaves bits in the message expansion so that it is not possible to split the expansion into 32 little expansions. The idea of the attack was to study the propagation of local perturbations in a linear variation of SHA-0 in order to discriminate between the role of the bare architecture and that of the elementary building blocks. The attack then looks for differential characteristic masks that can be added to the input word with non-trivial probability of keeping the output of the compression function unchanged. One first proposes a variant of SHA-0 called SHI1 which keeps all the rotations on blocks but replaces ADD with XOR, and makes the f_i functions XOR. Single bit errors or perturbations are then introduced to the input of SHI1 and the perturbation is traced through the cipher. These perturbations are made to disappear by introducing five other perturbations. This allows an attack on SHI1 via differential masking. A second variant on SHA-0, SHI2 is then analysed, where SHI2 replaces ADD with XOR but this time keeps the nonlinear f_i. However we can still view f_i as acting like $\oplus$ with some probability, and the probability of a successful perturbation attack can be computed. A third variant of SHA-0, SHI3, is then analysed, where SHI3 uses ADD as in SHA-0, but uses XOR instead of the nonlinear f_i. In this case the addition mod 2^{32} causes the perturbations to spread out due to carry propagation. However one is still able to devise a perturbation attack on SHI3 with probability 2^{-44}. Finally SHA-0 itself is analysed by taking into account the analyses of SHI2 and SHI3, and this leads to a perturbation based attack on SHA-0 requiring 2^{61} plaintexts. It should be emphasised that, although SHA-1 and SHA-0 are so similar, this attack does not carry over to SHA-1 or, consequently, SHACAL-1.

Ad-hoc linear and differential cryptanalysis of SHA-1 by Handscuh *et al.* has suggested that such attacks will not be effective against SHACAL-1 [280, 281]. Both attacks are complicated by the integer addition and the f_i functions of SHACAL-1. It is noted [280, 281] that $Z = A + S$ can be written bitwise as $z_j = a_j + s_j + \sigma_{j-1}$ and $\sigma_j = a_j s_j + a_j \sigma_{j-1} + s_j \sigma_{j-1}$, where σ_{j-1} is the carry bit, and $\sigma_{-1} = 0$. This bitwise way of expressing addition is used extensively in the analysis of [280, 281]. Linear cryptanalysis mostly uses single-bit approximations as heavier linear approximations are difficult to connect together. The cyclic structure of SHACAL-1 means that in all four rounds we can readily identify a family of linear approximations that always hold over four steps. A *perfect* (probability 1) linear approximation exists over both 4 and 7 steps. There is a 10-step linear approximation for rounds 2 and 4 which is valid over 40 steps with an estimated bias of 2^{-21}, and from these characteristics it is estimated that at least 2^{80} known plaintexts are required, although this cannot be considered a *break* as it is a very loose lower bound. For differential cryptanalysis there exists a 5-step characteristic over any 5 steps with probability 1, and it is conjectured that over 80 steps, the full cipher, the best differential characteristic has probability around 2^{-116}. It is emphasised in [280, 281] that their estimations are over-favourable to the cryptanalyst as it would be impossible to connect up all the constituent characteristics so as to achieve these biases. Neither is it expected that the more refined techniques involving linear hulls, multiple linear approximations, differentials, ... will make much difference.

Van Den Bogaert and Rijmen [606] search for optimal differential characteristics for reduced round SHACAL-1. The search is performed under the requirement that the Hamming Weight of every 32-bit word of the input is upper bounded by 2. It is found that there are two 10-step characteristics for f_{if} with probability 2^{-12} (this is a factor of 2 better than [281]), a 10-step characteristics for f_{xor} with best case probability 2^{-12}, and a 20-step characteristic for f_{if} and f_{maj} with probabilities 2^{-42} and 2^{-41} respectively (these figures agree with [281]).

Recently Saarinen [553] has noted that a slide attack can be mounted on SHA-1 with about 2^{32} effort. This attack is described in detail in Chapter 4 on hash functions in this report, with respect to a security analysis of SHA-1. The analysis demonstrates an unexpected property of the compression function of SHA-1, namely that the procedure for message expansion can be slid. However, it is not clear that this weakness can be exploited in the context of SHACAL-1. Also, in [553], Saarinen shows that the slide attack on SHA-1 points to a weakness in the key schedule of SHACAL-1, and this can be exploited in a related-key attack. Given access to two SHACAL-1 encryption oracles whose keys are "slid" (in the same way that the message expansion can be slid for the hash function) the cipher can be distinguished from a randomly chosen 160-bit permutation. This requires about 2^{128} chosen plaintexts. When certain properties hold for the (related) keys, the complexity can be further reduced to about 2^{96} chosen plaintexts. Differential cryptanalysis including boomerang attacks [347] and rectangle attacks [76] have also been applied to SHACAL-1. The best known attack works for 49 steps of the compression function with a data complexity of $2^{151.9}$ chosen plaintexts and a time complexity of $2^{508.5}$ [76].

2.6.4 SHACAL-2

2.6.4.1 The design

SHACAL-2
is based on SHA-256, which was introduced by NIST in 2000 [468,472]. In spite of similarity in name to SHACAL-1, SHACAL-2 is a completely different function. It is a 256-bit block cipher with a 512-bit key, although it can also be configured to take 512-bit blocks when based on SHA-512. SHA-256 operates in a similar way to SHA-1 but with some notable differences.

The encryption algorithm for SHACAL-2 is as follows.

- Put the 256-bit plaintext in eight 32-bit variables $A\,B\,C\,D\,E\,F\,G\,H$.
- For 64 steps do
- $\quad T_1 = H + \sum_1(E) + Ch(E\,F\,G) + K^i + W^i$.
- $\quad T_2 = \sum_0(A) + Maj(A\,B\,C)$.
- $\quad H^{i+1} = G^i$.
- $\quad G^{i+1} = F^i$.
- $\quad F^{i+1} = E^i$.
- $\quad E^{i+1} = D^i + T_1$.
- $\quad D^{i+1} = C^i$.
- $\quad C^{i+1} = B^i$.
- $\quad B^{i+1} = A^i$.
- $\quad A^{i+1} = T_1 + T_2$.

where the W^i are 32-bit step keys, the K^i are constants, different in each step, and

$$\begin{aligned} Ch(X\,Y\,Z) &= (X \text{ AND } Y) \oplus (\bar{X} \text{ AND } Z) \\ Maj(X\,Y\,Z) &= (X \text{ AND } Y) \oplus (X \text{ AND } Z) \oplus (Y \text{ AND } Z) \\ \textstyle\sum_0(X) &= S_2(X) \oplus S_{13}(X) \oplus S_{22}(X) \\ \textstyle\sum_1(X) &= S_6(X) \oplus S_{11}(X) \oplus S_{25}(X) \end{aligned}$$

where S_i means right rotation by i bits.

In order for SHACAL-2 to be invertible the final addition of the initial value, which occurs in hash mode for SHA-256, is omitted.

For SHACAL-2 the key schedule (message expansion) expands the 512-bit key (Master Key) to 2048 bits, and is as follows,

– The Master Key is a concatenation of 16 32-bit words: $[W^0|W^1|\ldots|W^{15}]$.
– $W^i = \sigma_1(W^{i-2}) + W^{i-7} + \sigma_0(W^{i-15}) + W^{i-16} \quad 16 \le i < 64$.

where σ_0 and σ_1 are defined as:

$$\begin{aligned} \sigma_0(x) &= S_7(x) \oplus S_{18}(x) \oplus R_3(x) \\ \sigma_1(x) &= S_{17}(x) \oplus S_{19}(x) \oplus R_{10}(x) \end{aligned}$$

where R_i means right shift by i bits. Keys shorter than 512 bits may be accomodated by padding the key input up to 512 bits.

2.6.4.2 Security analysis

No security flaws have been found in SHACAL-2. It also has the interesting property of being able to share most of the code of the SHA-256 hash function. Saarinen [553] has noted that the Slide attack on SHA-1 does not carry over to SHA-256, and hence does not consitute a threat for SHACAL-2. SHA-256 is a recently designed primitive, so more time is needed to perform a careful and thorough security evaluation of both SHA-256 and, consequently, SHACAL-2.

2.7 64-bit block ciphers not selected for Phase II

The 64-bit block ciphers not selected for phase II of NESSIE were CS-Cipher, Hierocrypt-L1, Nimbus, and NUSH. After discussing each cipher briefly we summarise and compare some of the distinguishing features of the ciphers, identifying potential weaknesses and noting the best-known attacks, as shown in Table 2.19 of Sect. 2.9.4. Note that the algorithms given here are not complete specifications, but references are given to complete specifications which may be found on the NESSIE website.

2.7.1 CS-cipher

2.7.1.1 The design

CS-Cipher is a 64-bit block, 128-bit key SPN cipher over 8 rounds [234]. Each round starts with a subkey XOR, followed by a layer of four 16-bit non-linear mixing transformations, M, and a byte permutation which is based on the Fast Fourier Transform (FFT) graph. This is repeated twice more in each round, but with constants used instead of the subkey in the initial XOR. There is a final key XOR after the last round. Whereas many ciphers use separate nonlinear (confusion) layers and linear (diffusion) layers, CS-Cipher also uses a nonlinear diffusion primitive within the nonlinear layer. The encryption process is summarised as,

$$k^8 \oplus E(k^7 \oplus \ldots E(k^1 \oplus E(k^0 \oplus m)) \ldots)$$

where k^i are the subkeys, m is the plaintext, and E is the round encryption function. The key schedule generates 9 subkeys of 64 bits each, is Feistel, and is summarised by,

$$k^i = k^{i-2} \oplus F_{c_i}(k^{i-1}) \qquad F_{c_i}(x) = T(P(x \oplus c^i))$$

where T is transposition, P is permutation, and the c_i are constants. Further design details may be found in [234].

2.7.1.2 Security analysis

Two successive applications of an FFT graph have been shown to give very good diffusion properties. The M transformation implements a multi-permutation [561] which here means that fixing either of the two 8-bit inputs arbitrarily makes both 8-bit outputs permutations of each other. This makes E a mixing function so that, if we arbitrarily fix seven of the eight 8-bit inputs, all outputs become permutations of the remaining free input. This gives good diffusion. Also each byte in the output of one round depends on all eight input bytes to that round. The non-linear transformation of the encryption takes two bytes of input, and has the following property: if one takes 256 inputs that are constant in one byte and take on all values once in the other byte, then each byte value occurs once in each of the two output bytes. This nonlinear transformation includes reasonably nonlinear involutions, P, and the designers show that at least five P boxes must be active per round, and this implies a satisfactory resistance to differential or linear cryptanalysis.

In [613] the designers prove, by counting the number of active S-boxes, that a modified version of CS-cipher with all constants and round keys replaced by independent random values is secure against linear and differential cryptanalysis. The designers [613] claim that these results carry over to the real CS-cipher and that $5\frac{1}{3}$ rounds of CS-cipher is therefore secure against linear and differential cryptanalysis.

No weaknesses or attacks have been reported on CS-cipher.

2.7.2 Hierocrypt-L1

Attacks on Hierocrypt-L1 significantly reducing the security margin have been found that the submitters were not aware of [41].

2.7.2.1 The design

Hierocrypt-LI (HC-L1) is a 64-bit block, 128-bit key SPN block cipher over 6 rounds [493]. It is hierarchical in structure with an SPN structure itself built of smaller SPN structures. Each round consists of a layer with two parallel XS-boxes each operating on a 32-bit input, followed (except in the last round) by a linear diffusion layer with a 64-bit input based on a bytewise MDS matrix. An XS-box consists of an upper subkey mixing layer, an upper S-box layer, a linear MDS-based diffusion layer (with a 32-bit input), a lower subkey mixing layer, and a lower S-box layer. The subkeys are XORed in, and the S-box used has 8-bit inputs and outputs. After the last round an output transformation introduces another subkey.

The key-schedule consists of two processes, namely the intermediate-key-generation and the round-key generation. The former generates intermediate keys out of the secret key, while the latter generates a round key out of each intermediate key. There are two ways to generate round keys; one is for a certain number of rounds close to the plaintext, and the other is for the remaining rounds nearer to the ciphertext. Most intermediate keys are used to generate one plaintext-side and one ciphertext-side round key. The schedule algorithms adopt a Feistel structure with a linear key scheduling in order to update intermediate keys, so that intermediate keys are supposed to be partially randomised in a non-linear manner.

For further details of the design refer to [493].

2.7.2.2 Security analysis

In the submission [493] the designers give a plausible argument that 6 rounds of Hierocrypt-L1 is secure against differential and linear cryptanalysis. They claim integral attacks for consistency work only up to 5 S-box layers (2.5 rounds) and that truncated differential attacks work only up to 5 rounds. During the 2nd NESSIE workshop the designers gave bounds not just on the best differential/linear characteristic, but also on the best differential and linear hull [495]. For 4-round Hierocrypt-L1, the upper bound on the probability of both is 2^{-48}.

Improved integral attacks for consistency on Hierocrypt-L1, for 6 or 7 S-box layers (3 or 3.5 rounds) have been found. This is not a real threat to the security of the cipher, but it is a better attack than the designers claimed exists. During the NESSIE assessment phase, an integral attack for consistency on 3.5 rounds was found by Barreto *et al.* [41]. Another such attack was also found by the designers. However, although a Gilbert-Minier distinguisher-type attack was successfully applied to 7 rounds of Rijndael by Gilbert and Minier [258], the alternation of upper and lower MDS diffusion layers effectively prohibits the construction of a 5-round Gilbert-Minier-type distinguisher on Hierocrypt.

It is also interesting to note that, as Hierocrypt uses essentially the same S-box as Rijndael, affine relationships exist between the bit outputs of the S-box [62,245] (see Sect. 2.9.5).

Further to this, Furuya and Rijmen [247] discovered linear relationships between the master key and several of the round subkeys. These flaws are common to key scheduling of all members of the Hierocrypt family. The attack of [247] exploits the fact that all intermediate keys of the key schedule are used to generate round keys, whereas the randomising effect of a Feistel structure requires that only half the intermediate keys are used. This flaw results in simple linear relations between intermediate keys. Also it turns out that the right halves of certain intermediate keys can be calculated from the right half of the padded secret key. Finally, because the Hierocrypt key schedule is similar to that of DES, it exhibits significant deterministic iterative differentials.

A summary of the designers' known attack requirements compared to the attack requirements found by NESSIE is given in Table 2.11.

Table 2.11. Attack Requirements for Hierocrypt-3 and Hierocrypt-L1

Attack	#S-box Layers	#Chosen-Plaintexts	#Subkey-Guesses
HC-3			
Designers	5	2^{13}	2^{168}
NESSIE	6	6×2^{32}	2^{40}
NESSIE	7	22×2^{32}	2^{168}
HC-L1			
Designers	5	2^{32}	2^{72}
NESSIE	6	6×2^{32}	2^{40}
NESSIE	7	14×2^{32}	2^{104}

2.7.3 Nimbus

There is a very practical attack on Nimbus by Biham and Furman [72].

2.7.3.1 The design

Nimbus is a 64-bit block cipher with a key of at least 128 bits over 5 rounds [410]. It is not an SPN. Each round consists of a subkey XOR, multiplication by another subkey, mod 2^{64}, and then bit-reversal of the data word. An encryption round is given by,

$$Y_i = K_i^{\text{odd}} \cdot g(Y_{i-1} \oplus K_i)$$

where K_i and K_i^{odd} are subkeys (K_i^{odd} is always odd), $\oplus$ is XOR, g is the bit-reversal function, and $\cdot$ is multiplication mod 2^{64}. Y_0 is the plaintext, and Y_5 is the ciphertext.

The key schedule generates ten 64-bit subkeys, with two new subkeys used in each round. These ten subkeys are generated from the user input key, which is at least 128 bits, by means of nested Nimbus encryption operations on successive 64-bit blocks of the user input key, with the encryption key being a constant derived from π.

For further details of the design please refer to [410].

2.7.3.2 Security analysis

The designer claims that Nimbus is secure against differential and linear cryptanalysis, interpolation attacks, impossible differential attacks, saturation attacks and related key attacks, and also claims that there is no effective attack on more than 4 rounds. Two new iterative differentials for multiplication operations with probability about $\frac{1}{2}$ have been found. By applying one of these differentials to

Nimbus, a 1-round iterative differential characteristic with probability $\frac{1}{2}$ can be obtained. Iterating this to the full 5-round cipher, a differential characteristic with probability 2^{-5} is obtained. This characteristic was used by Furman [246] to devise an attack on full Nimbus using 256 chosen plaintexts and 2^{10} complexity. Recently, Borisov *et al.* [111] summarised the attack of [246] using the language of multiplicative differentials, redefining the differential pairs of [246] as (x, x^*) where $x^* = -x \mod 2^{63}$ but $x^* \neq -x \mod 2^{64}$, a property that survives multiplication by the relevant key bits.

2.7.4 NUSH

2.7.4.1 The design

We here consider the version of NUSH which is a 64-bit block cipher, with a 128, 192, or 256-bit key over 9 rounds [391]. 128-bit and 256-bit block sizes were also submitted to NESSIE. Each round consists of four iterations. In each iteration two of four variables are updated using a subkey, while the other two are changed in a non-linear manner. Then the four registers are cycled round (byte-wise) in a Feistel way. NUSH does not use S-boxes and, to introduce nonlinearity, there are four different kinds of operations, using XOR, OR, AND, and bit rotations. Like IDEA, NUSH depends on the mixing of non-commutative operations for confusion and diffusion. The cipher also has a pre-whitening step and a post-whitening step where a subkey is added via XOR.

The key schedule of NUSH simply takes the user input key and partitions this key into different subkeys for use in the encryption algorithm. No nonlinearity is used in the key schedule. For further details of the design please refer to [391].

2.7.4.2 Security analysis

In the submission [391], the designers mention resistance against differential and linear cryptanalysis, weak key and related key attacks as well as other attacks, but do not include any details of their analysis.

A linear attack with complexity less than that of exhaustive key search for the different variants of NUSH has been reported by Wenling and Dengguo [618]. NESSIE has confirmed that there is such a linear approximation. This approximation is effective over the full cipher and holds with probability $\frac{1}{4}$ or $\frac{3}{4}$ depending on whether AND or OR is chosen in the iteration. Specifically, the linear approximation is of the form,

$$A_i[0] \oplus B_i[0] \oplus D_i[0] \simeq A_{i-1}[0] \oplus B_{i-1}[0] \oplus D_{i-1}[0]$$

where A, B, C, D are 16-bit partitions of the input/output block. But NESSIE has found some flaws in the further analysis. NESSIE has devised attacks on NUSH with 64-bit or 128-bit block size based on this linear approximation. These attacks are slightly faster than exhaustive search (by a factor of 2) for all key sizes. Furthermore, removing the first two rounds leaves all nine variants of NUSH vulnerable to a linear attack, suggesting a very limited security margin for the cipher.

2.8 128-bit block ciphers not selected for Phase II

The 128-bit block ciphers not selected for phase II of NESSIE were ANUBIS, Grand Cru, Hierocrypt-3, Noekeon, NUSH, Q, and SC2000. After discussing each cipher briefly we summarise and compare some of the distinguishing features of the ciphers, identifying potential weaknesses and noting the best-known attacks, as shown in Tables 2.20 and 2.21 of Sect. 2.9.5. Note that the algorithms given here are not complete specifications, but references are given to complete specifications which may be found on the NESSIE website.

2.8.1 Anubis

No security flaws have been found in the tweaked version of ANUBIS, and it is very similar to Rijndael.

2.8.1.1 The design

ANUBIS is a 128-bit SPN block cipher which accepts keys of length $32N$ bits (a minimum of 128 bits) and uses $8 + N$ rounds depending on the key size — a minimum of 12 rounds [37]. Each round consists of a subkey addition, 16 S-boxes (8-bit to 8-bit) and a linear transformation (presented as a matrix transpose operation). As in KHAZAD, all round components were chosen to be involutions in order to guarantee that encryption and decryption are identical but with the order of the subkeys reversed. Confusion and diffusion layers are kept separate, with the diffusion layer realised as a matrix transposition followed by a linear transformation designed to be an MDS code. In the original submission the S-boxes were randomly generated to avoid any internal structure. This tended to have a high cost in hardware, hence the tweaked submission used an S-box which could be decomposed into 3 layers of 4×4 mini-S-boxes, the same S-box as for KHAZAD (see Fig. 2.4 for KHAZAD), which has a complexity estimated to be one fifth of that for Rijndael [40].

The key schedule of ANUBIS is complicated and appears to be quite strong. It expands the cipher key into a series of round subkeys and uses a two-stage key-evolution and key-selection function. The schedule makes use of the encryption S-box combined with permutation, linear transformations based on MDS codes, and the addition of constants. For further details of the design refer to [37].

2.8.1.2 Security analysis

The designers claim [37] that no 4-round differential characteristic with probability higher than 2^{-125} exists and that no 4-round linear approximation with bias of more than $2^{-57.5}$ exists. They claim that related key attacks, interpolation attacks and boomerang attacks are infeasible, that truncated differential attacks only work up to 6 rounds and that saturation attacks work only up to 6 rounds. Such a saturation attack requires 6×2^{32} chosen-plaintexts, 2^{24} bits of storage (in addition to the memory required for storing the plaintext-ciphertext pairs) and time equivalent to 6×2^{48} S-box lookups. Extending the attack to 7 rounds requires almost the entire code book, 2^{64} bits of additional storage and analysis time equivalent to about 2^{120} encryptions. For 8 rounds, 2^{104} storage bits and

analysis time of 2^{204} encryptions is required (with the same data complexity). The designers claim [37] that the Gilbert-Minier attack can break 7 rounds using 2^{32} chosen -plaintexts and about 2^{140} S-box lookups. The designers claim [37] that the best impossible differential attack is on 5 rounds and uses $2^{29.5}$ chosen-plaintexts and 2^{31} analysis time.
Other than a small error in the calculation of linear approximation biases (the maximal bias is 17/256 instead of 13/256) that seems to have no effect on the overall security, no weaknesses or attacks have been reported. Apart from this mistake,it seems that the rest of the claims are correct, and that even the linear weakness cannot be exploited.

A summary of the known attacks according to the designers is given in Table 2.12.

Table 2.12. Designers Claims of Security — Known Attacks

Attack	Rounds	Key Size	Complexity		
			Data	Memory	Time
Saturation	6	all	6.2^{32} CP	2^{24} bits	6.2^{48}
	7	all	2^{119} CP	2^{64} bits	2^{104}
	8	> 204	2^{119} CP	2^{104} bits	2^{204}
Gilbert-Minier	7	> 140	2^{32}		2^{140}
Imp. Diff.	5	all	$2^{29.5}$		2^{31}

2.8.2 Grand Cru

2.8.2.1 The design

Grand Cru is a 128-bit block SPN block cipher over 10 rounds, requiring a key of at least 128 bits [112]. Grand Cru can be viewed as an enhanced version of Rijndael [181]. Rijndael encryption for a 128-bit key, K^0, can be described by,

$$\sigma_{K^0_{10}} \circ \pi \circ \gamma \circ \sigma_{K^0_9} \circ \prod_{i=8}^{0} (\theta \circ \pi \circ \gamma \circ \sigma_{K^0_i})$$

where σ is round key addition, γ is nonlinear substitution, π is a byte permutation, and θ is a linear transformation on a subset of the bytes. Grand Cru adds three keyed operations to give a four layered cipher (comprising four *subciphers*):

$$\psi_{K^3_1} \circ \nu^{-1} \circ \sigma_{K^0_{10}} \circ \beta_{K^2_9} \circ \pi_{K^1_9} \circ \gamma \circ \sigma_{K^0_9} \circ \prod_{i=8}^{0} (\beta_{K^2_i} \circ \theta \circ \pi_{K^1_i} \circ \gamma \circ \sigma_{K^0_i}) \circ \nu \circ \psi_{K^3_0}$$

where π_{K^1} is now a keyed permutation, where the round subkey K^1_i can take on $(4!)^5$ possible values, β_{K^2} is a keyed byte-wise rotation, where K^2_i can take on 2^{48} values, and two outer round key additions, ψ_{K^3}, are appended, using addition mod 2^8. ν is an extra diffusion layer. Note that the S-box is identical to that used in [181], as is the 4×4 matrix over GF(2^8) described by θ.

The cipher requires four 128-bit keys to derive the subkeys for the different keyed operations. When the user-selected key is shorter than 512 bits, these four keys are derived using a function claimed to be one-way. All round subkeys are derived using the Rijndael key schedule. Further details of the design can be found in [112].

2.8.2.2 Security analysis

Grand Cru is a cipher that implements *multiple layered security.* The idea behind this is to mix several ciphers in such a way that if all but one of them are broken, one is still left with a secure cipher. The designer shows that introducing the keyed operations not found in Rijndael does not reduce the security compared to Rijndael [112]. There is also a short analysis of the different ciphers that emerge when all but one set of the subkeys are known or chosen. It should be noted that Rijndael is very close to being one of these ciphers. Hence the designer claims that any attack that breaks Grand Cru also breaks Rijndael. Only in the case of weak keys for the permutation subcipher, π, and the rotation subcipher, β, may Grand Cru be weaker than Rijndael. The effectiveness of the different subciphers can be examined by assuming that the other subcipher keys are known or chosen. The designer shows that for the permutation subcipher there is a meet-in-the-middle attack requiring 2^{110} operations and storage (faster than exhaustive search).

No attacks or weaknesses have been reported by NESSIE on Grand Cru.

2.8.3 Hierocrypt-3

Attacks on Hierocrypt-3 significantly reducing the security margin have been found by Barreto *et al.* that the submitters were not aware of [41].

2.8.3.1 The design

Hierocrypt-3 (HC-3) is a 128-bit block SPN cipher taking 128-bit, 192-bit, or 256-bit keys, and operating over 6, 7, or 8 rounds, depending on the key size [493]. Like HC-L1, HC-3 has a hierarchical structure. At the highest level, an HC-3 round consists of, in order:

- A layer of four simultaneous applications of 32×32-bit keyed substitution boxes (XS-boxes).
- A diffusion layer consisting of a bytewise linear transform defined by the MDS_H matrix.

Within each round a similar structure exists. A 32-bit XS-box consists of, in order:

- An upper subkey mixing layer which XORs 32-bit input data with four subkey bytes.
- An upper (key-independent and nonlinear) S-box layer composed of the parallel application of four 8×8-bit S-boxes.
- A diffusion layer consisting of a bytewise linear transform defined by the MDS_L matrix.

– A lower subkey mixing layer.
– A lower S-box layer.

The output transformation is composed of an XS-box layer followed by an XOR layer with the last 128-bit subkey. The key schedule for Hierocrypt-3 follows the same algorithm as for Hierocrypt-L1, which is discussed under 64-bit block ciphers. More details of the design can be found in [493].

2.8.3.2 Security analysis

In the submission [493], the designers show that Hierocrypt-3 is secure against differential and linear cryptanalysis using conservative estimates. They also studied integral (Square) attacks for consistency and claim they work only up to 4 S-box layers (2 rounds) for a 128-bit key and up to 5 S-box layers (2.5 rounds) for a 192-bit or 256-bit key. They claim that 5 rounds is secure against truncated differentials. During the 2nd NESSIE workshop the designers gave bounds not just on the best differential and linear characteristics, but also on the best differential and linear hull [495]. For 4-round Hierocrypt-3, the upper bound on the probability of either is 2^{-96}.

During the NESSIE assessment phase, an integral attack for consistency was found on 7 S-box layers (3.5 rounds) by Barreto *et al.* [41]. Another such attack was also found by the designers. The attack requirements are summarised in Table 2.11 which is located in the subsection related to Hierocrypt-L1, Sect. 2.7.2, along with further comments regarding the security of Hierocrypt-3.

2.8.4 Noekeon

Both key schedules of Noekeon were found, by Knudsen and Raddum [368], to be susceptible to related key attacks.

2.8.4.1 The design

Noekeon is a 128-bit block, 128-bit key SPN cipher over 16 rounds [180]. Each round operates on four 32-bit words, a_0, a_1, a_2, a_3 and starts with the addition of a round constant to a_0. Then a_0 and a_2 are XORed together to make the word w, and two copies of w are made. One of the copies is rotated 8 bits to the left, and the other is rotated 8 bits to the right. These rotated copies are then XORed back onto w, and w is XORed onto a_1 and a_3. After this the 128-bit working key (see below) is added to the four words. Then a_1 and a_3 are used to create a temporary w in the same way as described above, and this word is XORed onto a_0 and a_2. The words a_1, a_2 and a_3 are then rotated 1, 5, and 2 bits, respectively, to the left. Then, for all 32 positions in a word, the bits that are in the same position in the different words are passed through a 4-bit S-box (32 parallel S-boxes in total). Finally, the round ends with the words a_1, a_2, a_3 being rotated 1, 5 and 2 bits, respectively, to the right. After the last round, the linear operations described before the rotations of a_1, a_2, a_3 are repeated one more time. The encryption and decryption routines are very similar.

Noekeon has two key schedules, one for applications where related-key attacks are not considered dangerous and one for applications where related-key attacks can be mounted. The stronger key schedule consists of taking the user selected key and encrypting it once with the all zero key. The ciphertext is then used as the so-called working key. The simpler key schedule simply uses the user selected key as the working key.

More details of the design can be found in [180].

2.8.4.2 Security analysis

The designers [180] claim resistance against linear and differential cryptanalysis. For linear attacks they claim that no 4-round linear characteristic with correlation coefficient higher than 2^{-24} exists. For differential attacks, the designers give a plausible argument that no 4-round differential with probability higher than 2^{-48} exists. The bounds are sufficient to conclude resistance against both attacks. The designers' reasoning that Noekeon is not vulnerable to attacks based on truncated differentials and to the interpolation attack also seems to be correct. Note, however, that the constituent functions of the S-box include the identity function on four of the sixteen different inputs.

In [368] Knudsen and Raddum show that there exist many related keys for which plaintexts of certain differences result in ciphertexts of certain differences with high probabilities independent of the key schedule used. It is also shown in [368] that for six of seven S-boxes which satisfy the design criteria of the Noekeon designers, the resulting block ciphers are vulnerable to either a differential attack, a linear attack or both. It is concluded that Noekeon is not designed according to an optimal diffusion strategy [368].

2.8.5 NUSH

2.8.5.1 The design

This version of NUSH is a 128-bit block cipher, with a 128, 192, or 256-bit key over 9 rounds [391]. 64-bit and 256-bit block sizes were also submitted to NESSIE.

More details of the design can be found in [391].

2.8.5.2 Security analysis

NUSH has no security margin [478] as it was broken. Details of the algorithm and security analysis can be found in Sect. 2.7.4 which describes the 64-bit version of NUSH.

2.8.6 Q

There are attacks on Q by Biham *et al.* and Keliher *et al.* both faster than exhaustive search [73, 341].

2.8.6.1 The design

Q is a 128-bit block cipher with a key size of 128, 192, or 256 bits over 8 rounds for 'low security' and over 9 rounds for 'high security' [434]. It was designed to be faster than Serpent and also to be immune to differential and linear cryptanalysis. The data is divided into sixteen 8-bit words (a 4 × 4 matrix of bytes). At the beginning of each round, a subkey word is XORed in, then an 8 × 8 S-box is applied 16 times (for each of the 16 data bytes) and an additional subkey XOR is performed. A 4 × 4 S-box is then applied 32 times, the round key is XORed in, a byte permutation is performed, and the 4 × 4 S-box is again used 32 times. After the last round, there is additional subkey XOR, an 8 × 8 S-box layer, a subkey XOR and an additional subkey XOR (post-whitening). Two of the three subkeys used in each round are the same, and are equal for all rounds and also for the last half round.

The key schedule applies an operation similar to encryption to the low order half of a 256-bit key (which may be a zero-padded 128-bit key. It includes an XOR with a round counter and an XOR with a constant derived from the Golden Ratio.

More details of the design can be found in [434].

2.8.6.2 Security analysis

The designers have claimed [434] that there are no differentials with probability larger than 2^{-120} after 7 rounds, and that there is no differential attack on the full Q. An equivalent claim is made regarding linear cryptanalysis. The designers also claim security against key-related attacks, slide attacks, Davies-Murphy attacks, boomerang attacks, approximation attacks and impossible differential attacks.

During the NESSIE assessment phase, a differential with higher probability than the upper limit claimed by the designers was found by Biham *et al.* and used to break the full Q. These results were presented at FSE 2001 [73]. A paper performing linear cryptanalysis of the full Q, with a significantly better attack than this differential attack, was presented by Keliher *et al.* at the 2nd NESSIE workshop [341]. The differential attack on full Q with 128-bit keys requires 2^{105} chosen plaintexts and has a time complexity of 2^{77} encryptions. The best attack on the full Q with larger key sizes requires 2^{125} chosen ciphertexts, and has a time complexity of 2^{96} for 192-bit keys, and 2^{128} for 256-bit keys. Table 2.13 summarizes these results.

Table 2.13. Data/Time Complexity of Attacks on Q for Different Key Sizes

Key Size (bits)	Number of Rounds	Chosen Plaintexts	Complexity (encryptions)
128	8	2^{105}	2^{77}
192	9	2^{125}	2^{96}
256	9	2^{125}	2^{128}

2.8.7 SC2000

2.8.7.1 The design

SC2000 is a 128-bit block cipher taking a 128, 192, or 256-bit key over 6.5 or 7.5 rounds [569]. It is a mixture of a Feistel cipher and an SPN. The round function of SC2000 consists of a layer of 32 parallel 4-bit S-boxes followed by two rounds of a Feistel network. The round keys are XORed with the cipher block before and after the application of the S-boxes. The last half round consists of key additions and S-box look-ups. The F-function in the Feistel network consists of a layer of four 6-bit S-boxes and eight 5-bit S-boxes, multiplication by a fixed 32×32 bit matrix and a final mixing of the words with each other using the AND operation with a constant and XOR. This produces two words of output from the function in the Feistel network. This two-round Feistel structure is the last operation on one round of SC2000. The key schedule is a complex transformation of the key selected by the user, in such a way that every 32-bit word of the round keys depends on the whole key.

More details of the design can be found in [569].

2.8.7.2 Security analysis

The designers [569] have studied the efficiency of the key avalanche and conclude that the complexity is that of exhaustive search if one tries to bypass one round in the beginning and end by guessing some of the key bits. A differential attack on 4.5 rounds has been reported by the designers. This attack finds 28 bits in the first and last round key. The key schedule in SC2000 appears to be very strong. The knowledge of one round key doesn't seem to leak any information about any of the other round keys or the key selected by the user, so the key schedule prevents the attacker from searching exhaustively for the remaining key bits. However, the success of this attack raises some questions about the design of SC2000. In [570] the designers present differential characteristics with higher probabilities than those found in their submission to NESSIE.

Raddum and Knudsen [539] and Dunkelman and Keller [211] both report attacks on SC2000 when the number of rounds is reduced to 3.5 or 4.5 from the original 6.5. In [539] two different 3.5-round differential characteristics with probabilities 2^{-106} and 2^{-107} are given. These characteristics have higher probabilities than those reported in [457]. The characteristics can be used to extract up to 32 bits of the first and last round keys in a 4.5-round variant of SC2000. In [211] distinguishers for 2.5 and 3 rounds are found and used to attack a 3.5 round variant of the cipher. These results on SC2000 were presented at the 2nd NESSIE workshop.

It is also interesting to note that affine relationships exist between the bit outputs of the S-box, for all three S-boxes, S4, S5, and S6 [62] (see Sect. 2.9.5).

2.9 Comparison of studied block ciphers

2.9.1 64-bit block ciphers considered during Phase II

Table 2.14 highlights the best attacks known on the block ciphers considered in this section. It identifies some unusual features of each cipher and some of its potential weaknesses, and gives a list of the best attacks, including the number of rounds broken, data and time complexities, and references for the attacks. MISTY2 and KASUMI are not part of NESSIE but are included for comparison with MISTY1.

Table 2.15 shows affine bit-relations between output bits of the S-boxes, i.e. the number of equations of the form,

$$b_i(x) = b_j(\mathbf{A}x + \mathbf{B}) + c$$

for the vector of S-box input bits, x, a Boolean matrix $\mathbf{A}$, a Boolean vector $\mathbf{B}$, and a constant, $c \in \{0, 1\}$. The submatrices of $\mathbf{A}$ which are permutation or rotation matrices are also enumerated [62].

2.9.2 128-bit block ciphers considered during Phase II

Table 2.16 highlights the best attacks known on the block ciphers considered in this section. It identifies some unusual features of each cipher and some of its potential weaknesses, and gives a list of the best attacks, including the number of rounds broken, data and time complexities, and references for the attacks.

Table 2.17 shows affine bit-relations between output bits of the S-boxes, i.e. the number of equations of the form,

$$b_i(x) = b_j(\mathbf{A}x + \mathbf{B}) + c$$

for the vector of S-box input bits, x, a Boolean matrix $\mathbf{A}$, a Boolean vector $\mathbf{B}$, and a constant, $c \in \{0, 1\}$. The submatrices of $\mathbf{A}$ which are permutation or rotation matrices are also enumerated [62].

2.9.3 Large block ciphers considered during Phase II

Table 2.18 highlights the best attacks known on the block ciphers considered in this section. It identifies some unusual features of each cipher and some of its potential weaknesses, and gives a list of the best attacks, including the number of rounds broken, data and time complexities, and references for the attacks.

2.9.4 64-bit block ciphers not selected for Phase II

Table 2.19 highlights the best attacks known on the block ciphers considered in this section, identifying the number of rounds over which they operate, some unusual features of the cipher, some potential weaknesses of the cipher, and a list of the best attacks, including the number of rounds broken, data and time complexities, and references for the attacks.

2.9.5 128-bit block ciphers not selected for Phase II

Table 2.20 highlights the best attacks known on the block ciphers considered in this section, identifying the number of rounds over which they operate, some unusual features of the cipher, some potential weaknesses of the cipher, and a list of the best attacks, including the number of rounds broken, data and time complexities, and references for the attacks.

Table 2.14. A summary of each 64-bit block cipher selected for Phase II

Cipher	Rnds	Unusual Features	Potential Weaknesses	Best Attacks				
				Rnds	Technique	Time	Data	Ref.
IDEA	8.5	Juxtaposition of 3 dissimilar algebraic groups, + mod 2^{16}, × mod $2^{16}+1$ and ⊕ mod 2. No S-boxes Encryption ≃ Decryption Widely studied for over a decade — few security flaws	Homomorphisms from × mod $2^{16}+1$ to + mod 2^{16} or ⊕ mod 2 (partial) Linear Key Schedule (weak) Relatively large weak-key classes	2	Diff	2^{42}	2^{10}	[437]
				2	Square	23	2^{64}	[191]
				2.5	Square	2^{58}	3.2^{16}	[457]
				2.5	Square	2^{79}	2^{48}	[457]
				3	Diff-Lin	$3 \cdot 2^{42}$	2^{29}	[86]
				3.5	Miss-in-the-middle	2^{53}	238.5	[66]
				3.5	Square	2^{82}	2^{34}	[191]
				4	Miss-in-the-middle	2^{70}	2^{38}	[66]
				4	Diff-Lin (related-key)	38	38.3	[288]
				4	Square	2^{34}	2^{114}	[191]
				4.5	Miss-in-the-middle	2^{112}	2^{64}	[66]
				4.5	Boomerang 2^{101} weak keys	2^{18}	2^{18}	[86]
				5	Boomerang 2^{95} weak keys	4	4	[86]
				8.5	Boomerang 2^{53} weak keys	4	4	[86]
				8.5	Boomerang 2^{64} weak keys	2^{16}	2^{16}	[86]
KHAZAD	8	All components involutions Efficient Diffusion Very high diffusion branch by use of linear transformation (MDS code) S-box does not depend on simple mathematical function. Encryption = Decryption	Slightly weak S-box nonlinearity	3	Square	2^{16}	2^{8}	[39]
				3	Imp. Diff.	2^{64}	2^{13}	[74]
				4	Square	2^{80}	2^{9}	[39]

Table 2.14. (continued) A summary of each 64-bit block cipher selected for Phase II

Cipher	Rnds	Unusual Features	Potential Weaknesses	Best Attacks				
				Rnds	Technique	Time	Data	Ref.
MISTY1	8	Entire algorithm built from recursive components. Hard to analyse. Additional FL layers. S-boxes have irregular 7/9 splitting. Simple ANF for S-boxes S-boxes have optimal nonlinearity. Key schedule uses S-boxes Provable security against diff/lin. crypt. Encryption $\simeq$ Decryption	Low algebraic degree of S-boxes. Simple key schedule. Hardware/software penalty through 7/9 S-box splitting. Generalised linear approx. Characteristics not optimal. Complicated.	4	Slicing/Diff (FL)	$2^{81.6}$	2^{27}	[381]
				4	Collision (FL)	2^{89}	2^{20}	[380]
				4	(FL)	$2^{90.4}$	2^{23}	-
				4	(FL)	2^{62}	2^{38}	-
				4	(FL)	2^{89}	2^{20}	-
				4	(FL)	2^{76}	2^{28}	-
				4	(FL)	2^{27}	25	[371]
				5	High Diff (No FL)	2^{17}	$11 \cdot 2^{7}$	[604]
				5	High Diff (No FL)	2^{38}	2^{6}	[601]
				6	Diff (No FL)	2^{106}	2^{39}	[380]
				6	(No FL)	2^{61}	2^{54}	-
								[371]
MISTY2 (not part of NESSIE)	-			5	High Diff (No FL)	2^{39}	2^{7}	[601]
				5	Diff (FL)	2^{62}	2^{38}	[380]
				5	Collision (FL)	2^{76}	2^{28}	[380]
				6	Integral (FL)	2^{71}	2^{34}	[371]
KASUMI (not part of NESSIE)	-			4	Diff (No FL)	2^{22}	2^{10}	[605]
				5	Square (FL)	2^{80}	2^{38}	3GPP
				5	Lin. (FL)	2^{95}	2^{58}	3GPP
				6	Rel Key (FL)	2^{112}	$3 \cdot 2^{17}$	[102]
				6	Imp Diff (Rnds 2-7,FL)	2^{100}	2^{55}	[380]

Table 2.14. (continued) A summary of each 64-bit block cipher selected for Phase II

Cipher	Rnds	Unusual Features	Potential Weaknesses	Best Attacks				
				Rnds	Technique	Time	Data	Ref.
SAFER++ (64-bit block)	8	4-point PHT for diffusion Two incompatible group operations, ⊕ and +, mod 256. S-boxes use exp and log. Key addition uses both ⊕ and + mod 256. Mini versions possible for analysis. Use of bias words for key schedule	Non-homomorphic linear approx. Rather small bit and byte branch numbers for diffusion layer. S-box using 45^x exhibits linear relationship with prob. 1	2	Lin	2^{42}	2^5	[459]
				3	Lin weak key frac: 2^{-6}	2^{121}	2^{33}	[459]
Triple-DES (three key)	48	DES analysed for many years with no significant security flaws. Extension of DES. Backwards compatible with DES.	S-boxes are not optimally nonlinear. Lin. Crypt. Inefficient.		2^{28} related-keys	2^{84}	2^{28}	[60]
					Improved m.i.t.m.	$1.3 \cdot 2^{104}$	2^{32}	[406]
				8	Diff-Lin			[389]
				8	Diff-Lin	$2^{14.8}$	$2^{14.8}$	[70]
				9	Diff-Lin	$2^{29.17}$	$2^{15.75}$	[70]
				10	Diff-Lin	2^{50}	2^{20}	[70]
					Related-key			[345]
					Meet-in-the-middle	2^{112}	3	[441]
(two key)					Chosen plaintext	2^{56}	2^{56}	[442]

Table 2.21 shows affine bit-relations between output bits of the S-boxes, i.e. the number of equations of the form, $b_i(x) = b_j(\mathbf{A}x + \mathbf{B}) + c$ for the vector of S-box input bits, x, a Boolean matrix $\mathbf{A}$, a Boolean vector $\mathbf{B}$, and a constant, $c \in \{0, 1\}$. The submatrices of $\mathbf{A}$ which are permutation or rotation matrices are also enumerated [62].

Table 2.15. Affine relations for the S-boxes of some Phase II 64-bit block ciphers

			Sbox			Inverse Sbox		
Cipher	Sbo	Size	# Aff. Eqns	Perm	Rot	Total	Perm	Rot
KHAZAD-tweak		8×8	-	-	-	-	-	-
KHAZAD-orig		8×8	-	-	-	-	-	-
MISTY1	S7	7×7	> 2000	-	-	> 2000	-	-
	S9	9×9	> 100000	20	32	720	-	-
SAFER++ (64-bit block)	exp45	8×8	256	-	-	(log45) 7	-	-
DES	S1	6×4	1	-	-	not invertible		
	S2	6×4	3	-	-	not invertible		
	S3	6×4	4	-	-	not invertible		
	S4	6×4	28	-	-	not invertible		
	S5	6×4	3	-	-	not invertible		
	S6	6×4	3	-	-	not invertible		
	S7	6×4	6	1	-	not invertible		
	S8	6×4	-	-	-	not invertible		

Table 2.16. A summary of each 128-bit block cipher selected for Phase II

Cipher	Rnds	Unusual Features	Potential Weaknesses	Best Attacks				
				Rnds	Technique	Time	Data	Ref.
Camellia	18	Byte-Oriented, Feistel	x^{-1} S-box function	5	Imp Diff			[65]
		Use of x^{-1} for S-box	potentially open to	6	Square (FL)		2^{16}	[627]
		FL/FL^{-1} layers	algebraic attacks e.g.	6	Square (No FL)	2^{112}	$2^{11.7}$	[292]
		with 1-bit rot.	quadratic expressions,	7	Imp Diff (No FL)			[602]
			and affine relationship	8	Trunc Diff (No FL)	$2^{55.6}$	$2^{83.6}$	[392]
			between S-box o/ps	9	Square+key (FL)	2^{202}	2^{60}	[627]
			Identical rounds lead	9	Diff Crypt (No FL)		2^{105}	[67]
			to Slide attacks	9	Boomerang (FL)	2^{170}	2^{124}	[577]
				10	Rectangle (FL)	2^{241}	2^{127}	[577]
				11	Diff (No FL)	2^{232}	2^{104}	[577]
				11	High Diff (No FL)	2^{255}	2^{21}	[287]
				11	High Diff (FL)	2^{256}	2^{93}	[287]
				12	Lin (No FL)	2^{247}	2^{119}	[577]
RC6	20	Very simple description	Vulnerable via approx.	16	Diff. Crypt.		2^{128}	[143]
		Progression from RC5	across data-dep. rots.	16	Lin. Crypt.		2^{119}	[143]
		Data-dependent rots.	Small safety margin	8	Lin. Crypt.		2^{47}	[324]
		Quadratic function for		14	mult. lin. crypt.	2^{186}	2^{120}	[568]
		diffusion.		18	mult. lin. crypt.	2^{193}	2^{127}	[568]
		Strong, complex key-			with 2^{-90} weak key fraction			
		schedule.		15	χ^2 (stat.) attack			[366]
		Mini-versions possible.			by fixing 5 lsbs in A,C			
		Uses 32-bit mult. mod 2^{32}		17	χ^2 (stat.) attack			[366]
		Feistel-like			for 2^{-80} keys			
		Encryption ≠ Decryption			Extrapolated			
					experimentally			

Table 2.16. (continued) A summary of each 128-bit block cipher selected for Phase II

Cipher	Rnds	Unusual Features	Potential Weaknesses	Best Attacks				
				Rnds	Technique	Time	Data	Ref.
Rijndael	10,12,14	Very high	Small safety margin	5	Imp Diff	2^{31}	$2^{29.5}$	[74]
		diffusion	Potentially open to	6	Square	2^{72}	2^{32}	[181]
		Use of x^{-1} for S-box	algebraic attacks	7	Collision (192,256)	2^{140}	2^{32}	[258]
		Encryption $\neq$ Decryption	e.g. quadratic	7	Collision (128-bit key)	2^{128}	2^{32}	[258]
		Diffusion via MDS trans.	expressions, and	7	Square (192-bit key)	2^{176}	2^{32}	[407]
		Highly elegant	affine relationship	7	Square (256-bit key)	2^{192}	2^{32}	[407]
		Provable resistance to	between S-box o/ps	6	Square	2^{44}	2^{32}	[227]
		Diff/Lin. Crypt.	Unnecessarily weak	7	Square (192-bit key)	2^{155}	2^{32}	[227]
		Implicit Nested SPN	key-schedule.	7	Square (256-bit key)	2^{172}	2^{32}	[227]
		structure	Description using BES	7	Square	2^{120}	2^{128}	[227]
			Too elegant to be	8	Square (192-bit key)	2^{188}	$2^{128} - 2^{119}$	[227]
			random?	8	Square (256-bit key)	2^{204}	$2^{128} - 2^{119}$	[227]
				9	Related-key/Square	2^{224}	2^{77}	[227]
					256 related keys			
					(256-bit keys)			
SAFER++ (128-bit block)	8	4-point PHT for diffusion	Non-homomorphic	2.75	Imp. Diff.	2^{60}	2^{64}	[460]
		Two incompatible group	linear approx.	3.25	Square	2^{70}	$2^{9.6}$	[456]
		operations, $\oplus$ and	Rather small bit	3.25	Lin. Crypt.	2^{101}	2^{81}	[327, 459]
		$+$, mod 256.	and byte branch	3.75	(256-bit key)	2^{176}	2^{81}	[327, 459]
		S-boxes use exp and log.	numbers for diffusion		weak key frac: 2^{-13}			
		Key addition uses both $\oplus$	layer.		Lin (256-bit key)	2^{167}	2^{91}	[459]
		and $+$ mod 256.	S-box using 45^x		weak key frac: 2^{-11}			
		Mini versions possible	exhibits linear	4	Integral (128-bit key)	2^{112}	2^{64}	[520]
		for analysis.	relationship with	4	Integral (128-bit key)	2^{120}	2^{64}	[520]
		Use of bias words for	prob. 1		(less memory)			
		key schedule	Decryption diffusion	4	Integral (256-bit key)	2^{144}	2^{64}	[520]
			weak	4	Boomerang (128-bit key)	2^{41}	2^{41}	[84]
				5	Boomerang (128-bit key)	2^{75}	2^{75}	[84]
				5.5	Boomerang (128-bit key)	2^{121}	2^{121}	[84]

Table 2.17. Affine relations for the S-boxes of some Phase II 128-bit block ciphers

			Sbox			Inverse Sbox		
Cipher	Sbox	Size	# Aff. Eqns	Perm	Rot	Total	Perm	Rot
Camellia		8×8	504	-	-	504	-	-
AES-Rijndael		8×8	504	-	-	504	-	-
SAFER++ (128-bit block)	exp45	8×8	256	-	-	(log45) 7	-	-

Table 2.18. A summary of each large block cipher selected for Phase II

Cipher	Rnds	Unusual Features	Potential Weaknesses	Best Attacks				
				Rnds	Technique	Time	Data	Ref.
RC6	20	Very simple description Progression from RC5 Data-dependent rots. Quadratic function for diffusion. Strong, complex key-schedule. Mini-versions possible. Uses 32-bit mult. mod 2^{32} Feistel-like Encryption $\neq$ Decryption	Vulnerable via approx. across data-dep. rots. Small safety margin	$3+2s$	χ^2 (stat.) attack Extrapolated experimentally		2^{16+20s}	[359]
Rijndael	14	Wide-Trail Strategy Use of x^{-1} for S-box Encryption $\neq$ Decryption Diffusion via MDS trans. Highly elegant Provable resistance to Diff/Lin. Crypt. Implicit Nested SPN structure	Small safety margin Potentially open to algebraic attacks e.g. quadratic expressions, and affine relationship between S-box o/ps Unnecessarily weak key-schedule. Description using BES Too elegant to be random?					
SHACAL-1	80 (steps)	SHA well studied. Derived from hash fn. uses + mod 2^{32}, AND, OR, and data rots.	Potentially weak key-schedule (slide property)	41 47 49	Diff. Amp. Boomerang Rectangle	2^{491} $2^{508.4}$ $2^{508.5}$	2^{141} $2^{158.5}$ $2^{151.9}$	[347] [347] [76]
SHACAL-2	64 (steps)	Derived from hash fn. uses + mod 2^{32}, AND, OR, and data rots.						

Table 2.19. A summary of each 64-bit block cipher not selected for Phase II

Cipher	Rnds	Unusual Features	Potential Weaknesses	Best Attacks				
				Rnds	Technique	Time	Data	Ref.
CS-Cipher	8	SPN, nonlinear diff	-	5.5	Lin/Diff	-	-	[234]
		FFT-based Diffusion						
Hierocrypt-L1	6	Hierarchical SPN	Weak key-schedule	2.5	Integral	2^{72}	2^{32}	[493]
		MDS Diffusion		3	Integral	2^{40}	6×2^{32}	[493]
		Feistel key schedule		3.5	Integral	2^{104}	14×2^{32}	[41]
				5	Trunc Diff			[493]
					Key-schedule			[247]
Nimbus	5	Not SPN,	High prob. Diff.	5	Diff	2^{10}	2^{8}	[72]
		Mult, mod 2^{64}						
NUSH	9	Four iterations/round	High lin. bias	-	Lin	2^{K-1}	-	[618]
		Feistel-like, No S-box						

Table 2.20. A summary of 128-bit block ciphers not selected for Phase II

Cipher	Rnds	Unusual Features	Potential Weaknesses	Best Attacks				
				Rnds	Technique	Time	Data	Ref.
ANUBIS	$8 + N$	Like KHAZAD/Rijndael		5	Imp Diff	2^{31}	$2^{29.5}$	[37]
		Uses involutions		6	Saturation (all keys)	6×2^{48}	6×2^{32}	[37]
		Diffusion uses		7	Saturation (all keys)	2^{120}	2^{128}	[37]
		matrix transposition		7	Gilbert-Minier (key-size > 140)	2^{140}	2^{32}	[37]
				8	Saturation (key size > 204)	2^{204}	2^{128}	[37]
Grand Cru	10	Enhanced Rijndael						
		Multi-layered security						
Hierocrypt-3	6-8	Hierarchical SPN	Weak key-schedule	2.5	Integral	2^{168}	2^{13}	[493]
		MDS Diffusion		3	Integral	2^{40}	6×2^{32}	[493]
		Feistel key schedule		3.5	Integral	2^{168}	22×2^{32}	[41]
Noekeon	16	four 32-bit words	S-box includes		Diff/Lin			[368]
		Two key schedules	identity funcs					
			many related keys					
NUSH		Four iterations/round	High lin. bias	-	Lin	2^{K-1}	-	[618]
		Feistel-like, No S-box						
Q	8	4×4-bit data blocks	High Diff/Lin prob.	[73]				
8	Diff (128-bit key)	2^{77}	2^{105}	[73]				
				9	Diff (192-bit key)	2^{96}	2^{125}	[73]
				9	Diff (256-bit key)	2^{128}	2^{125}	[73]
				8	Lin	2^{32}	2^{97}	[341]
SC2000	6.5 - 7.5	Feistel/SPN	High Diff Prob.	3.5	Diff			[211]
		4×4, 5×5, 6×6 S-boxes		4.5	Diff			[569]
		Strong key-schedule		4.5	Diff			[539]

Table 2.21. Affine relations for the S-boxes of some 128-bit block ciphers not selected for Phase II

			Sbox			Inverse Sbox		
Cipher	Sbox	Size	# Aff. Eqns	Perm	Rot	Total	Perm	Rot
Hierocrypt-3		8×8	504	-	-	504	-	-
SC2000	S4	4×4	> 1000	7	-	> 1000	7	-
	S5	5×5	> 1000		-	> 3000	8	-
	S6	6×6	210		-	210	8	-

3. Stream ciphers

3.1 Introduction

A stream cipher is an algorithm for encrypting a sequence of elements or characters from a plaintext alphabet, usually the binary alphabet $\{0, 1\}$. Stream ciphers are commonly classified as being *synchronous* or *self-synchronising*. In a *synchronous* stream cipher the keystream is generated independently of the plaintext and ciphertext, so the keystream depends only on the key. In contrast, the keystream of a *self-synchronising* stream cipher depends on the key and a fixed amount of the previously generated ciphertext. Most stream ciphers can be classified as *additive* stream ciphers. An additive stream cipher is a synchronous cipher in which the ciphertext is the XOR of the plaintext and the keystream. None of the submissions is a self-synchronising stream cipher, therefore only synchronous stream ciphers are considered.

In specific applications, stream ciphers are more appropriate than block ciphers:

- Stream ciphers are generally faster than block ciphers, especially in hardware.
- Stream ciphers have less hardware complexity.
- Stream ciphers process the plaintext character by character, so no buffering is required to accumulate a full plaintext block (unlike block ciphers).
- Synchronous stream ciphers have no error propagation.

Most stream ciphers are based on simple devices that are easy to implement and run efficiently. A common example of such a device is the *linear feedback shift register* (LFSR) [551]. Such simple devices produce predictable output given some previous output. Thus, the output of such devices is typically used as the input to a function that produces the keystream. Keystreams can also be produced by using certain modes of operation of a block cipher.

Many of the common uses of stream ciphers require frequent key reinitialization or rekeying. A full definition of a stream cipher intended for such uses should give details of how the cipher should be rekeyed. The original NESSIE call for primitives did not require stream ciphers to be accompanied by a rekeying schedule. Only the two SOBER stream ciphers provided a rekeying schedule with the original schedule, though rekeying schedules have subsequently been provided for the other submissions.

[0] Coordinator for this chapter: SAG — Marcus Schafheutle, Stefan Pyka

3.2 Security requirements

The techniques used to analyse stream ciphers use mathematical and statistical properties of the generator or approximations to it. In particular, the keystream generator should produce a memoryless balanced sequence of bits, which are modelled as a sequence of independent identically distributed Bernoulli random variables with parameter 0.5 (fair coin tosses). Stream cipher analysis is essentially concerned with analysing the keystream generator to find deviations from this statistical model. It is customary when analysing stream ciphers to consider known plaintext attacks. This essentially means assuming that a large amount of keystream is known. The statistical deviations are exploited to give methods for attacking the stream cipher based on the known keystream. Such methods are usually classified in one of the following three ways:

1. *Distinguishing Attack.*
   A method for distinguishing output from the keystream generator from a 'random' sequence of the same length.
2. *Prediction.*
   A method for predicting output from the keystream generator more accurately than guessing
3. *Key Recovery.*
   A method for recovering the key from the output of the keystream generator.

Key recovery is clearly the most powerful of these three methods as it enables both prediction and a distinguishing attack. Prediction also clearly enables a distinguishing attack. However, a distinguishing attack can be thought of as a type of prediction. This is because a distinguishing attack makes statements that certain sequences from a keystream generator are more or less likely to occur than they would if produced 'at random', thus making predictions about the keystream sequence that are more accurate than guessing.

3.2.1 Classification of attacks

Stream ciphers tend not to use iterated functions in the same way as block ciphers, so the classification of attack techniques is more difficult. However, the most common techniques are discussed below.

Exhaustive key search

This attack is the most general type of attack that can be applied to any stream cipher. Given a keystream sequence generated by an unknown key, an attacker simply tries all possible keys and checks whether the generated keystream matches the given keystream sequence. For exhaustive key search for stream ciphers there exist very efficient time-memory tradeoff techniques as described in [87]. In a time-memory tradeoff attack, some key-output relations are precomputed and stored in memory. In the real-time phase of the attack, the given output data is searched until a stored output pattern is found. The attacker has then found the corresponding key. The time complexity for exhaustive key search is split into a time and a memory complexity. The name 'time-memory tradeoff' results from this idea.

Periodic and Statistical Attacks

If the period of a keystream generator is too small, then the keystream will repeat itself, so enabling easy prediction. The period must be large enough to ensure that the keystream is not repeated. More generally, if the keystream deviates in an obvious way from the memoryless Bernoulli distribution discussed above, then there is an obvious prediction technique available.

Linear Complexity

The linear complexity of a sequence is the length of the shortest LFSR that can produce that sequence. The linear complexity of a sequence is easily calculated using the Berlekamp-Massey algorithm [419]. If this linear complexity is too small, then an attacker can reproduce the sequence on an LFSR.

Maximum Order Complexity

The Maximum Order Complexity (MOC) Test determines the length of the shortest possibly non-linear feedback shift register which can produce the given bit sequence For the MOC profile, this is done for the first 1,2,3.. bits of the sequence.

Correlation Attacks

Correlation attacks are the most important general attacks on LFSR-based stream ciphers. In a correlation attack, the output of a keystream generator is correlated in some manner with the output of a much simpler device, such as a component LFSR of the generator. This correlation can sometimes be exploited to determine the key. The first ideas were described by Thomas Siegenthaler [585]. Meier and Staffelbach [438] and others subsequently improved these ideas by developing fast correlation attacks.

Higher Order Correlation attacks

Many stream ciphers are built of a linear sequence generator and a non-linear output function f. Correlation attacks try to find a linear approximation of f. Equivalenty higher order approximations are possible. With the aid of the approximations an overdefined system of multivariate equations can be defined. The XL method [164] can be adapted to solve these equations. This kind of attack is not well examined, but a more detailed description can be found in [154].

Divide-and-Conquer Attacks

In such attacks a portion of the key (or of the internal state) is guessed. The constraints now placed on the keystream may allow the determination of the remainder of the key faster than searching this remainder exhaustively.

Rekeying Attacks

There are many applications in which a stream cipher is frequently rekeyed. It is sometimes possible to exploit this rekeying in order to find the key.

Sidechannel attacks

Using sidechannel attacks one exploits the behaviour of primitives in use, for example different performances or different power consumption for different keys or internal states. For a complete description of sidechannel attacks see the annex.

3.2.2 Assessment process

The stream cipher submissions were assessed with reference to the above generic common stream cipher attack techniques. Furthermore, some stream cipher submissions were analysed using techniques specific to that primitive.

Statistical testing is a vital part of stream cipher analysis in order to provide assurance that the generator possesses the required statistical properties. The NESSIE toolbox provides extensive tools for stream cipher and block cipher testing. A list of these tools used in stream cipher assessment is given below, and further details can be found in [479]. It is emphasized that properties like large period, large linear complexity and a good statistical behaviour are necessary but not sufficient conditions for a stream cipher to be considered cryptographically secure.

3.2.2.1 The NESSIE statistical toolbox for stream ciphers

The NESSIE toolbox provides several tests in order to assess the statistical properties of output sequences from the keystream generator of a stream cipher:

- Collision Test
  The collision test splits up the bit sequence into blocks of a fixed size. A collision occurs if the same block appears more than once. The test statistically evaluates the number of collisions.
- Constant Runs Test
  For the constant runs test, the sequence of bits is subdivided into runs, that is maximal disjoint subsequences of consecutive 0s and 1s. The frequencies of these runs of the various lengths are evaluated statistically.
- Correlation Test
  The correlation test statistically evaluates the correlation between a sequence and shifts of the sequence.
- Coupon Collector's Test
  The coupon collector's test splits up the bit sequence into blocks of a fixed size. The test statistically evaluates the number of blocks required until all possible blocks have appeared. The coupon test is also applied to cyclic shifts of the original sequence.
- Dyadic Complexity Test
  The dyadic complexity test is an implementation of the complexity measure suggested by Goretzky and Klapper [348] for sequences of bits. This measure is cryptologically relevant because feedback shift registers with carry, also described in [348], have low dyadic complexity.
- The Fast Spectral Test
  The fast spectral test applies the fast Walsh transform to the bit sequence. It

uses two values derived from the transform to assess the randomness of the sequence.

- Frequency Test
  The frequency test splits up the bit sequence into blocks of a fixed size. The frequencies of these blocks are evaluated statistically.
- Gap Test
  The gap test splits up the bit sequence into blocks of a fixed size. The blocks are interpreted as binary representations of numbers, to give a sequence of numbers. A gap is a maximal subsequence containing no members in a certain numerical range, and the lengths of gaps are evaluated statistically. This test is also applied to cyclic shifts of the original sequence.
- Linear Complexity Test
  The linear complexity test uses the Berlekamp–Massey algorithm to determine the length of the shortest linear feedback shift register which can produce the given bit sequence. The linear complexity profile is also evaluated.
- Maximum Order Complexity Test
  The maximum order complexity test determines the length of the shortest possibly nonlinear feedback shift register which can produce the given bit sequence. For the maximum order complexity profile, this is done for the first 1,2,3... bits of the sequence.
- Overlapping m-tuple Test
  The overlapping m-tuple test splits up the bit sequence into overlapping subsequences of length m. The frequency of these (dependent) subsequences is evaluated statistically. This test is also applied to cyclic shifts of the original sequence.
- Percolation Test
  The percolation test is the simulation of a forest fire. The bit sequence to be tested determines where trees are standing in the simulated forest. The test evaluates statistically how fast a fire propagates in the simulated forest.
- Poker Test
  The poker test splits up the bit sequence into groups of k successive blocks of a fixed size, known as (poker) hands. The poker test statistically evaluates the frequencies of these hands. This test is also applied to cyclic shifts of the original sequence.
- Rank Test
  In the rank test, the bits of the sequence to test are used to fill square matrices. The bits are treated as elements of the field GF(2), and the ranks of the matrices are evaluated statistically.
- Run Test
  The run test splits up the bit sequence into blocks of a fixed size. These blocks are interpreted as binary representations of numbers, to give a sequence of numbers. A run is a maximal subsequence of strictly increasing numbers, and the lengths of runs are evaluated statistically.
- Universal Maurer Test
  The universal Maurer test splits up the bit sequence into disjoint subsequences of bits. The test statistically evaluates the distances between identical subse-

quences. The test result of the Maurer test is closely related to the entropy of the bit sequence.
- Ziv-Lempel Complexity Test
  The Ziv-Lempel complexity test is based on a measure of the rate at which new patterns occur in the sequence.

These tests were applied to all of the stream cipher submissions, but no submission exhibited anomalous behaviour. Detailed results of the statistical tests are available as NESSIE public reports.

Some block cipher tests were also used for stream cipher analysis. As an example, the following tests were applied to analyse the key loading (initialization vector loading) process used in the stream ciphers SNOW, SOBER-t16 and SOBER-t32:

- Dependence Test
  The dependence test evaluates the dependence matrix and the distance matrix of a function. Furthermore, the degree of completeness, the degree of avalanche effect and the degree of strict avalanche criterion of the function are computed.
- Linear Factors Test
  The linear factors test is used to find out whether there are any linear combinations of output bits which, for all keys and plaintexts, are independent of one or more key or plaintext bits. Such a linear combination is called a linear factor.

These tests detected linearity properties in the key loading of SOBER-t32 [199].

3.3 Overview of the common designs

In this section we give an overview of stream cipher design techniques which are most commonly used today in practice. Because of the broad field of stream cipher design, only a brief overview is presented in order to give a rough classification of the stream cipher submissions.

3.3.1 Stream ciphers based on feedback shift registers

Feedback shift registers, in particular LFSRs, are widely used as building blocks for stream ciphers. LFSRs produce sequences having large periods and good statistical properties, they are well-suited for hardware implementations and there are mathematical techniques to analyse them. Unfortunately, the output sequence of an LFSR is linear and so is easily predictable. When LFSRs are used as components for keystream generators, it is very important that the output sequence does not inherit linearity properties from the output sequences of the component LFSRs. Some methodologies with this objective are briefly discussed in the next sections.

The submitted ciphers SNOW, LILI-128, SOBER-t16 and SOBER-t32 are all based on LFSRs.

Nonlinear combination generators

A nonlinear combination generator uses a number of LFSRs. The keystream is generated as a nonlinear function f of the outputs of these LFSRs.

Nonlinear filter generators

In this construction, the keystream is generated as a nonlinear function f of the stages of a single LFSR.

Clock-controlled generators

An irregularly clocked LFSR does not exhibit the same linearity properties as one that is regularly clocked. Thus a common technique is to use one LFSR sequence to control the clocking of another LFSR. A more general form of clock-control is the irregular decimation of the output sequence of one device by another device. For each generated sequence bit from the first device, the second device decides whether the generated bit should be used as a keystream bit or discarded.

The stream cipher submissions LILI-128, SOBER-t16 and SOBER-t32 use LFSRs, nonlinear filters and irregular clocking as main components.

3.3.2 Stream ciphers based on block ciphers

Some modes of operation of block ciphers can be used to generate a keystream sequence, such as the Output Feedback (OFB) Mode, the Cipher Feedback (CFB) Mode and the Counter Mode (CTR). The BMGL stream cipher submission to NESSIE is in reality a block cipher mode of operation. Note that stream ciphers based on block cipher modes of operation can potentially be attacked by cryptanalysis of the underlying block cipher. There are also generic distinguishing attacks on block ciphers in OFB and Counter Mode. For a block cipher with block size b, $2^{b/2}$ blocks of keystream are sufficient to distinguish the keystream from a truly random sequence. This is achieved by looking for repeated occurrences of blocks, which are not possible when the stream is generated by a block cipher in OFB or Counter Mode (unless the sequence has started to repeat itself).

3.3.3 Pseudorandom number generators based on modular arithmetic

The security of these generators is based on the presumed intractability of an underlying number-theoretic problem. Popular examples of this class are the RSA generator and the Blum-Blum-Shub generator [100]. The required modular arithmetic makes these generators extremely slow compared to other keystream generators, so such generators are primarily used as pseudorandom number generators.

3.3.4 Other stream ciphers

Stream ciphers based on LFSRs are well-suited for hardware. Some recent stream ciphers have been designed particularly for efficient software implementation, and are not based on LFSRs. Examples include the stream ciphers RC4 [542], SEAL [544] and SCREAM [279], and the NESSIE stream cipher submission LEVIATHAN.

3.3.5 Current standards

At present a standard for dedicated stream ciphers, such as the AES standard for block ciphers, does not exist. One probable reason is that most stream ciphers in use are either secret or proprietary designs.

The standard ISO 10116 (2nd edition) specifies modes of operation for 64-bit block ciphers that give keystream generators, in particular the Cipher Feedback Mode and the Output Feedback Mode. Several other standards (ANSI X9.52, FIPS 81) specify modes of operation for the DES and triple-DES cipher. Recently, the National Institute of Standards and Technology (NIST) held two public workshops on block cipher modes of operation. Some of the proposed modes are suitable for stream cipher design, for example the Counter Mode and the Key Feedback Mode (BMGL is based on this mode). In the NIST publication SP800-30A, modes of operation that could be used as keystream generators, such as Cipher Feedback Mode, Output Feedback Mode and Counter Mode are recommended. However, as described in section 3.3.2, block ciphers in OFB and Counter Mode might be vulnerable to distinguishing attacks. When speaking of OFB mode, one usually considers the OFB mode with full feedback in contrast to the OFB mode with r-bit ($r < n$) feedback. With full feedback the complete output word from the last encryption step is used for feedback, while with r-bit feedback only r bits are used for feedback. In the first case the feedback function can be treated as a random permutation with expected cycle length of about 2^{n-1}, while in the second case the feedback function is a random function with expected cycle length $2^{n/2}$. This is the reason why OFB is used in full feedback mode, and why it is vulnerable to distinguishing attacks. On the other hand there is a simple way to remove this drawback. If the block size is twice as big as the key size, then one needs as many output words as the key space size to mount a distinguishing attack, which is not better than brute force. So if one uses a 256-bit block cipher with a 128-bit key, then this configuration provides adequate security in the normal category.

3.4 Stream cipher primitives considered during Phase II

3.4.1 BMGL

BMGL is a stream cipher designed by Johan Håstad and Mats Näslund [285] and submitted to the NESSIE project. BMGL provides key sizes as for Rijndael (128-bit, 192-bit and 256-bit). Furthermore, the tweaked version of BMGL allows rekeying with an 128-bit initialization vector.

3.4.1.1 Design

The construction of BMGL is based on so called *hardcore functions* for one-way permutations and on the possibility to construct pseudo random number generators with the aid of them. Given a one-way function f, a set of binary functions $\{b_r\}$ is called a family of *hardcore functions*, if for any r the output $b_r(x)$ cannot be distinguished computationally from a random bit.

Given the seeds x_0 and r and the one-way permutation f, one can construct a random number generator g by $x_{i+1} = f(x_i)$ and $g(x_0, r) = b_r(x_1), b_r(x_2), \ldots$. One can show, that if there is an efficient algorithm D that distinguishes with non-negligible advantage $g(x, r)$ from a random string (with given r), then there is an efficient algorithm P that given r, $f(x)$ predicts $b_r(x)$ with non-negligible advantage [101]. Furthermore Goldreich and Levin [269] have shown, that in this case, there is an efficient algorithm B, that inverts $f(x)$ on random x with non-negligible probability. They also could prove, that for every one-way function f, *hardcore functions* exist. The set $\{b_r\}$ can be defined by the inner product $b_r(x) := x \cdot r \bmod 2$ of n-bit strings x and r. This also holds for extending the inner product on matrix products, in order to generate several bits at once. Let M_m^n be the set of all $m \times n$-matrices and let $R \in M_m^n$. Then one can define hardcore functions

$$B_R^m(x) = \begin{pmatrix} r_{11} & r_{12} & \ldots & r_{1n} \\ r_{21} & r_{22} & \ldots & r_{2n} \\ \ldots & \ldots & & \ldots \\ r_{m1} & r_{m2} & \ldots & r_{mn} \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ \ldots \\ x_n \end{pmatrix} .$$

This results in the basic construction of BMGL:

Let $f : \{0,1\}^n \to \{0,1\}^n$ be a one-way function and let n, m and λ be integers. Then the generator $BMGL_{n,m,\lambda m}(f)$ is defined as follows: The input of the generator is $x_0 \in \{0,1\}^n$ and $R \in M_m^n$. Let $x_i = f(x_{i-1})$. Then the output of the generator is defined by $\{B_R^m(x_i)\}_{i=1}^{\lambda}$.

In the BMGL submission for the NESSIE project the authors have defined the function f as the block cipher Rijndael. The construction of BMGL is based on theoretical results for one-way functions. Therefore the security of BMGL with Rijndael as one-way function has to be related to these theoretical results. The authors show, that if Rijndael is secure, then their construction of BMGL is secure.

In the BMGL submission Rijndael is used in a new mode of operation called *key feedback mode* (KFB). The idea is not to feedback the ciphertext as new plaintext input in the next iteration step, but as a new key. In some sense it is a "dual" to OFB mode. The plaintext and the random matrix R may be publicly known and need not be a part of the secret key.

3.4.1.2 Security analysis

The security of the construction results from the assumption that the one-way function used in the BMGL generator does not lose its one-wayness property even if the function is iterated many times. Based on previous papers about the reduction of the security of generators of pseudorandom bits to the existence of one-way functions [269, 284], the designers show formally in the model of exact security that a non-trival attack on BMGL gives a black-box reduction to an attack on the underlying iterated one-way function, i.e. Rijndael. The analysis allows us to quantify the loss of security. The security and correctness of the overall construction has been verified several times before in the case of one-way permutations [265, 266, 269, 284, 402]. All proofs given in the submission were carefully checked.

In [286] the submitters generalize the generator in order to allow keystream synchronization with random access properties. Furthermore, they present a sketch for a security proof based on the assumption that the iterated Rijndael mapping is hard to invert even if an attacker has a number of extra plaintext-ciphertext pairs.

We note that BMGL is in reality a mode of use of a block cipher, in this case Rijndael. This means that certain "generic attacks" on modes of use are applicable as noted by Babbage [29]. The type of generic attacks described there is a time-memory tradeoff for stream ciphers, where the internal state size is not much bigger than the key space size. Babbage recommends that the internal state size of the key stream generator should be at least two times bigger than the key space size to prevent time-memory tradeoffs. For the stream cipher BMGL this is not the case.

3.4.2 SNOW

SNOW is a synchronous stream cipher designed by Patrick Ekdahl and Thomas Johansson [214] and submitted to the NESSIE project. It uses a 128-bit or 256-bit key and has an internal memory of 576 bits. The tweaked version of SNOW allows rekeying with a 64-bit initialization vector.

3.4.2.1 The design

SNOW consists of an LFSR and a Finite State Machine, the states of which are words in $GF(2^{32})$. We denote addition in $GF(2^{32})$ by the symbol $\oplus$, addition modulo 2^{32} by the symbol $+$, and the ith bit of the element x in the field by $x[i]$. SNOW uses an LFSR of length 16 defined by the recurrence relation

$$s_{t+16} = \alpha(s_t \oplus s_{t+3} \oplus s_{t+9})$$

where $\alpha \in GF(2^{32})$ is given in the NESSIE submission. The Finite State Machine consists of two registers whose values at time t we will denote by a_t and b_t respectively. Let

$$\begin{aligned} a_{t+1} &= a_t \oplus R(f_t + b_t) \\ b_{t+1} &= S(a_t) \\ f_t &= (s_{t+15} + a_t) \oplus b_t \\ z_t &= f_t \oplus s_t, \end{aligned}$$

where R denotes a 7-bit left (towards the most significant bit) rotation and S is a 32-bit to 32-bit S-box given in the NESSIE submission. The sequence (z_t) is used as the keystream. There is a key initialisation process described in the submission.

3.4.2.2 Security analysis

A distinguishing attack. Coppersmith, Halevi and Jutla [147] have observed that if $\sigma_t = s_{t+15}[15] \oplus s_{t+15}[16] \oplus s_{t+16}[22] \oplus s_{t+16}[23] \oplus f_t[15] \oplus f_{t+1}[23]$ then σ_t has bias $\epsilon = 2^{-8.3}$.

For a sequence (v_t) and polynomial $q(X) = \sum_{i=0}^{N} c_i X^i$, let $T_q^j(v_t)$ denote $\sum_{i=0}^{N} c_i v_{j+i}$. They also show that there are about 2^{100} weight six polynomials which are divisible by the linear feedback polynomial given above. If p is such a polynomial, then $T_p^j(s_t) = 0$ for each j. For each such p, $T_p^j(\sigma_t) = T_p^j(f_t[15]) \oplus T_p^j(f_t[23])$, so we can obtain $T_p^j(\sigma_t)$ since we know the f_t. However, $T_p^j(\sigma_t)$ has bias $\epsilon^6 = 2^{-49.8}$ since p has weight 6. As there are about 2^{100} such polynomials p, we can find 2^{100} such $T_p^j(\sigma_t)$, each with a bias of $2^{-49.8}$.

The mean of the sum of these 2^{100} values is $2^{99} + 2^{75}$ and the standard deviation approximately $2^{49} + 2^{24}$. The mean of a sum of 2^{100} 'random' values would be 2^{99} and the standard deviation 2^{49}. We can assume that the distributions of these sums are normal, so most (about 95%) realisations of these sums lie within two standard deviations of the mean. Thus it is clear we can distinguish the two distributions. This leads to a distinguishing attack on SNOW requiring 2^{95} observed bits of keystream and workload about 2^{100}.

Guess and determine attacks. Hawkes and Rose [290] present two attacks on SNOW, the first requiring 2^{64} observed bits of keystream with workload 2^{256} (so no faster than exhaustive key search) and the second 2^{224} observed bits of keystream with workload 2^{95}.

For the basic attack we fix t and assume that $b_t = S(a_t \oplus \mathbf{1})$ and $b_{t+14} = S(a_{t+14} \oplus \mathbf{1})$ where $\mathbf{1} = 2^{32} - 1$. We guess s_t, s_{t+1}, s_{t+2}, s_{t+3}, a_t and a_{t+14} (192 bits in total) and can then determine the shift register state based on these assumptions and test if this shift register state is correct by comparing its output with the keystream. If no guesses give the correct keystream for this value of t, we repeat this with another value of t. As the probability that these assumptions hold is 2^{-64}, we expect to have to try about 2^{64} values of t. The details of how to determine the shift register state from each guess are given in [290]. This gives an attack requiring 2^{64} observed bits of keystream with workload 2^{256}.

We can improve the workload of this attack by also assuming that a_{t+1} is either 0 or $\mathbf{1}$ as well as $b_t = S(a_t \oplus \mathbf{1})$ and $b_{t+14} = S(a_{t+14} \oplus \mathbf{1})$ as before. We guess s_t, s_{t+1}, a_t, a_{t+14} and whether a_{t+3} is 0 or $\mathbf{1}$ (129 bits in total) and can then determine the shift register state and test whether this state is correct. The probability that these assumptions are correct is 2^{-95}, so we expect to have to repeat this for about 2^{95} values of t. As before, the details of how to determine the shift register state from each guess are given in [290]. This gives us an attack requiring 2^{224} observed bits of keystream with workload 2^{95}.

3.4.3 SOBER-t16

SOBER-t16 is a synchronous stream cipher designed by Philip Hawkes and Greg Rose [289] and submitted to the NESSIE project. SOBER-t16 uses a 128-bit key and has an internal memory of 272 bits. SOBER-t16 allows rekeying with an initialization vector. Hawkes and Rose also submitted to NESSIE SOBER-t32, a similar stream cipher but with a 256-bit key (see Section 3.4.4).

3.4.3.1 Description of SOBER-t16

SOBER-t16 is based on an LSFR of length 17 over the field $GF(2^{16})$. This LFSR is 'stuttered' as described below. We represent elements of this field with 2^{16} elements by 16-bit binary vectors corresponding to polynomials modulo the irreducible polynomial

$$x^{16} + x^{14} + x^7 + x^6 + x^4 + x^2 + x + 1.$$

We denote addition in $GF(2^{16})$ by the symbol $\oplus$, addition modulo 2^{16} by the symbol $+$ and the ith bit of the element x in the field by $x[i]$.

The linear feedback shift register. SOBER-t16 uses an LFSR of length 17 over $GF(2^{16})$ given by the recurrence relation

$$s_{t+17} = \alpha s_{t+15} \oplus s_{t+4} \oplus \beta s_t$$

where $\alpha =$ `0XE382` and $\beta =$ `0X67C3`.

The nonlinear filter. If the shift register output is given by the sequence (s_t), then the nonlinear filter (NLF) used by SOBER-t16 is given by

$$v_t = ((f(s_t + s_{t+16}) + s_{t+1} + s_{t+6}) \oplus K) + s_{t+13}$$

where

$$f(a) = S(\bar{a}) \oplus (a - \bar{a})$$

and $\bar{a}$ denotes the 8 most significant bits of a, so $a - \bar{a}$ is the 8 least significant bits of a. The value K is a 16-bit key-dependent constant initialised by the key loading and S is an 8-bit to 16-bit substitution box specified in the submission.

The stuttering. This pair of shift register and nonlinear filter are stuttered as follows. The first output word v_1 of the nonlinear filter is the first 'stutter control word' and is partitioned into 8 pairs of bits. Starting with the least significant pair, these pairs determine the stuttering according to the table below, where $C =$ `0X6996`, and $\sim C$ is the bitwise complement of C. When all the pairs have been used, the shift register is clocked and the output of the nonlinear filter becomes the next stutter control word. This procedure is repeated until sufficient keystream has been produced.

00	Clock LFSR.
01	Clock LFSR. Output XOR of C with NLF output to the keystream. Clock LFSR.
10	Clock LFSR twice. Output the NLF output to the keystream.
11	Clock LFSR. Output XOR of $\sim C$ with NLF output to the keystream.

Key loading and rekeying. The 17 states $r_1, \ldots, r_{17}$ of the shift register are set to the first 17 Fibonacci numbers. The key loading uses two operations: the '`Include`' operation and the '`Diffuse`' operation. The `Include` operation consists of adding a given word to r_{15} modulo 2^{16}. The `Diffuse` operation consists of clocking the shift register and replacing r_4 with the XOR of r_4 with the nonlinear filter output. To load the key, the `Include` operation is applied with each key word in turn, with each `Include` being followed by the `Diffuse` operation. When this has been completed, the `Include` operation is then performed using the key length as input and the `Diffuse` operation is applied 17 times. The shift register is then clocked and K set to the nonlinear filter output.

It is also possible to rekey the cipher by using a public 'frame key'. This is done by loading the frame key after the secret key in the same manner.

3.4.3.2 Observations on SOBER-t16

We first consider a simplified version of SOBER-t16 in which the stuttering is not used. Ekdahl and Johansson [215] have found a distinguishing attack on this unstuttered SOBER-t16, which we now briefly describe. Let $w_t = v_t \oplus s_t \oplus s_{t+1} \oplus s_{t+6} \oplus s_{t+16} \oplus s_{t+13}$ and $W_t = w_{t+17} \oplus \alpha w_{t+15} \oplus w_{t+4} \oplus \beta w_t$. Then $W_t = z_{t+17} \oplus \alpha z_{t+15} \oplus z_{t+4} \oplus \beta z_t$ so we can obtain the sequence (W_t) from the keystream. The distribution of w_t (for fixed K) can be estimated by simulation and is non-uniform for all the values of K for which this has been done. The error in the distribution obtained this way can also be estimated. We can then use this distribution to calculate the distribution of W_t (for fixed K). The Neyman-Pearson lemma then tells us that (in the worst case scenario) we need 2^{92} keystream words to distinguish between keystream from the cipher and a random keystream with probability of error 2^{-32}. The computational complexity of this distinguishing attack is 2^{92}.

Ekdahl and Johansson [215] have also found a distinguishing attack on SOBER-t16 (including stuttering). Let $\mathcal{E}$ be the event that if $z_n = C_o \oplus v_t$ then $z_{n+2} = C_1 \oplus v_{t+4}$, $z_{n+7} = C_2 \oplus v_{t+15}$ and $z_{n+8} = C_3 \oplus v_{t+17}$, where $C_1, C_2, C_3 \in \{C, \sim C, 0\}$. We can calculate the most probable position in the keystream for each of $(v_t, v_{t+4}, v_{t+15}, v_{t+17})$ so we can show that $\mathcal{E}$ happens with probability $2^{-5.5}$. Let $W'_t = W_t \oplus C_3 \oplus \alpha C_2 \oplus C_1 \oplus \beta C_0$. If the event $\mathcal{E}$ happens, then $W'_t = z_{n+8} \oplus \alpha z_{n+7} \oplus z_{n+2} \oplus \beta z_n$ so we can calculate the sequence (W'_t) from the keystream. If we assume that the distribution is uniform when $\mathcal{E}$ does not occur, then we can calculate the distribution of W'_t. The Neyman-Pearson lemma then tells us that we need 2^{111} keystream words to distinguish between keystream from the cipher and a random keystream with probability of error 2^{-32}. The computational complexity of this distinguishing attack is 2^{111}.

Pyka [534] has independently found a bias in each bit of w_t (other than the 0th, 1st, 2nd and 4th bits) when $K = 0$. For $K \neq 0$ the correlations change their values depending on the bits of K. He also pointed out that for unstuttered SOBER-t16, the distribution of w_t for a given K can be calculated precisely, since the bias of $f(s_t + s_{t+16}) \oplus s_t \oplus s_{t+16}$ can be determined by looking at all the possibilities for s_t and s_{t+16}. The bias of each bit can then be determined by looking at the probabilities of the carry values.

Besides those attacks, SOBER-t16 is vulnerable to timing attacks and power attacks due to its irregular decimation [289].

3.4.3.3 Comments from the submitters

The submitters have made some comments on the relevance of these results about SOBER-t16 and about the results about SOBER-t32, discussed below. These comments are briefly summarised in Section 3.4.4.3, in the discussion of SOBER-t32.

3.4.3.4 Conclusions

There is a distinguishing attack on SOBER-t16, as well as a much faster one on the unstuttered version of SOBER-t16. The nonlinear filter also exhibits significant biases. Furthermore, SOBER-t16 is vulnerable to timing and power attacks.

3.4.4 SOBER-t32

SOBER-t32 is a synchronous stream cipher designed by Philip Hawkes and Greg Rose [289] and submitted to the NESSIE project. SOBER-t32 uses a 256-bit key and has an internal memory of 544 bits. SOBER-t32 allows rekeying with an initialization vector. Hawkes and Rose also submitted SOBER-t16 to NESSIE, a similar stream cipher but with a 128-bit key (see Section 3.4.3).

3.4.4.1 Description of SOBER-t32

SOBER-t32 is based on an LSFR of length 17 over the field $GF(2^{32})$. This LFSR is ‘stuttered’ as described below. We represent elements of this field with 2^{32} elements by 32-bit binary vectors corresponding to polynomials modulo the irreducible polynomial

$$x^{32} + (x^{24} + x^{16} + x^{8} + 1)(x^{6} + x^{5} + x^{2} + 1).$$

We denote addition in $GF(2^{32})$ by the symbol $\oplus$, addition modulo 2^{32} by the symbol $+$ and the ith bit of the element x in the field by $x[i]$.

The linear feedback shift register. SOBER-t32 uses an LFSR of length 17 over $GF(2^{32})$ given by the recurrence relation

$$s_{t+17} = s_{t+15} \oplus s_{t+4} \oplus \alpha s_t$$

where $\alpha = \texttt{0XC2DB2AA3}$.

The nonlinear filter. If the shift register output is given by the sequence (s_t), then the nonlinear filter (NLF) used by SOBER-t32 is given by

$$v_t = ((f(s_t + s_{t+16}) + s_{t+1} + s_{t+6}) \oplus K) + s_{t+13}$$

where

$$f(a) = S(\bar{a}) \oplus (a - \bar{a})$$

and $\bar{a}$ denotes the 8 most significant bits of a, so $a - \bar{a}$ is the 24 least significant bits of a. The value K is a 32-bit key-dependent constant initialised by the key loading and S is an 8-bit to 32-bit substitution box specified in the submission.

The stuttering. This pair of shift register and nonlinear filter are stuttered as follows. The first output word v_1 of the nonlinear filter is the first 'stutter control word' and is partitioned into 16 pairs of bits. Starting with the least significant pair, these pairs determine the stuttering according to the table below, where $C =$ `0X6996C53A`, and $\sim C$ is the bitwise complement of C. When all the pairs have been used, the shift register is clocked and the output of the nonlinear filter becomes the next stutter control word. This procedure is repeated until sufficient keystream has been produced.

00	Clock LFSR.
01	Clock LFSR. Output XOR of C with NLF output to the keystream. Clock LFSR.
10	Clock LFSR twice. Output the NLF output to the keystream.
11	Clock LFSR. Output XOR of $\sim C$ with NLF output to the keystream.

Key loading and rekeying. The 17 states $r_1, \ldots, r_{17}$ of the shift register are set to the first 17 Fibonacci numbers. The key loading uses two operations: the '`Include`' operation and the '`Diffuse`' operation. The `Include` operation consists of adding a given word to r_{15} modulo 2^{32}. The `Diffuse` operation consists of clocking the shift register and replacing r_4 with the XOR of r_4 with the nonlinear filter output. To load the key, the `Include` operation is applied with each key word in turn, with each `Include` being followed by the `Diffuse` operation. When this has been completed, the `Include` operation is then performed using the key length as input and the `Diffuse` operation is applied 17 times. The shift register is then clocked and K set to the nonlinear filter output. It is also possible to rekey the cipher by using a public 'frame key'. This is done by loading the frame key after the secret key in the same manner.

3.4.4.2 Observations on SOBER-t32

We first consider a simplified version of SOBER-t32 in which the stuttering is not used. Ekdahl and Johansson [215] have found a distinguishing attack on this unstuttered version of SOBER-t32, which we now briefly describe. Let $w_t = v_t \oplus s_t \oplus s_{t+1} \oplus s_{t+6} \oplus s_{t+16} \oplus s_{t+13}$ (as for SOBER-t16). Although we cannot estimate the distribution of w_t, we can estimate the distribution of $w_t[i] \oplus w_t[i-1]$ for each i by simulation. It can be shown that $s_{t+\tau_5} \oplus s_{t+\tau_4} \oplus s_{t+\tau_3} \oplus s_{t+\tau_2} \oplus s_{t+\tau_1} \oplus s_t = 0$ where $\tau_1 = 11$, $\tau_2 = 13$, $\tau_3 = 4 \cdot 2^{32} - 4$, $\tau_4 = 15 \cdot 2^{32} - 4$, $\tau_5 = 7 \cdot 2^{32} - 4$. Thus if we let $Z_t = w_{t+\tau_5} \oplus w_{t+\tau_4} \oplus w_{t+\tau_3} \oplus w_{t+\tau_2} \oplus w_{t+\tau_1} \oplus w_t$ then $Z_t = z_{t+\tau_5} \oplus z_{t+\tau_4} \oplus z_{t+\tau_3} \oplus z_{t+\tau_2} \oplus z_{t+\tau_1} \oplus z_t$ so the sequence (Z_t) can be calculated from the keystream. Since we know the distribution of $w_t[i] \oplus w_t[i-1]$ for each value of i, we can calculate the distribution of $Z_t[i] \oplus Z_t[i-1]$. For certain values of i this has bias $2^{-40.5}$. The Neyman-Pearson lemma then tells us that (in the worst case scenario) we need $2^{86.5}$ keystream words to distinguish between keystream from the cipher and a random keystream with probability of error 2^{-32}. The computational complexity of this distinguishing attack is $2^{86.5}$.

De Cannière, Lano, Preneel and Vandewalle [189] enhanced the attack described in [215] by adapting the attack on SOBER-t16 so that it works for SOBER-t32. This attack results in a distinguishing attack on full SOBER-t32 of complexity 2^{153}.

The designers of SOBER-t32 describe a guess-and-determine attack on unstuttered SOBER-t32 with complexity 2^{320} [289]. This attack is extended by de

Cannière to improve the complexity to 2^{304} [188]. The unstuttered version of SOBER-t32 may also be analysed by noting that the 8 most significant bits of $\bar{a}$ determine the 8 most significant bits of $f(a)$. Babbage and Lano [30] also describe a guess-and-determine attack based on this observation on unstuttered SOBER-t32 with complexity 2^{244} and also indicate that their approach might be adapted to SOBER-t32 by using timing information to find an unstuttered part of the keystream that their approach requires.

The observation that the 8 most significant bits of $f(a)$ are determined solely by the 8 most significant bits of $\bar{a}$ is also the basis of some comments about the nonlinear filter function. This results in poor diffusion properties for the nonlinear filter, as the only means of diffusion of the 24 least significant bits of the input is by carry propagation. However, long carry chains only occur with low probability. In particular Dichtl and Schafheutle [199] specify a sum of bits from the initial state of the shift register that remains constant with very high probability when the last key bit is inverted. They have also identified other such sums that remain constant for inversion of each of the 11 least significant key bits of the last key word and for each of 8 of the 9 least significant bits of the second to last key word. For more significant positions, carry propagation seems to be sufficiently high to destroy such linearity properties. This property of the key loading could be destroyed by increasing the number of `Diffuse` steps. Similarly, they show that there is a correlation between the initial states derived from different frame keys but the same key. In particular, they give a sum of bits of the initial state that does not change with very high probability if the least significant bit of the last key frame word is inverted. As the frame key is a binary counter this is particularly relevant.

Besides those attacks, SOBER-t32 is vulnerable to timing attacks and power attacks owing to its irregular decimation [289].

3.4.4.3 Comments from the submitters

The submitters have made some comments on the relevance of these results, which also apply to SOBER-t16. These comments are briefly summarised here.

- There is a distinguishing attack on the unstuttered version of SOBER-t32. The submitters in their comments state that irregular stuttering is an essential part of the security of SOBER. However, the NESSIE project believes that the original SOBER submissions clearly state the SOBER stream ciphers are secure without the stuttering.
- The submitters question the rejection of a cipher based solely on distinguishing attacks based on large known plaintext, which is faster than exhaustive key search [291]. The submitters argue that there are many stream ciphers in use for which distinguishing attacks exist (such as a block cipher used in a standard mode that gives a stream cipher) and that the distinguishing attacks presented for the SOBER ciphers do not yield any information about the state, so do not translate into key recovery or prediction attacks.

3.4.4.4 NESSIE response to the comments from the submitters

- Distinguishing attacks on full SOBER-t16 and SOBER-t32 have been found since the comments from the submitters were received.
- Distinguishing attacks do not currently give a method for determining the keystream for SOBER-t16 and SOBER-t32. However, the NESSIE consortium believes that the SOBER distinguishing attacks mean that a recommendation is inadvisable.
- The SOBER design raises other security issues not discussed by the submitters.

3.4.4.5 SOBER-t32 Conclusions

There are distinguishing and guess-and-determine attacks on the unstuttered version of SOBER-t32, and recently distinguishing attacks for full SOBER-t32 have been found. For the guess-and-determine attack there are reasons for believing that it could be extended to SOBER-t32. The nonlinear filter exhibits poor diffusion and has significant biases. Furthermore, SOBER-t32 is vulnerable to timing and power attacks.

3.5 Stream cipher primitives not selected for Phase II

3.5.1 LEVIATHAN

LEVIATHAN is a synchronous additive stream cipher submitted by Cisco and designed so that it can efficiently find arbitrary locations in the keystream. Though LEVIATHAN may work with arbitrary output size and an arbitrary number of key bytes, owing to standardization the key should be either 128 bits or 256 bits and the output size should be 32 bits. It was submitted to NESSIE, but not considered during Phase II of the project.

3.5.1.1 Design

The key scheduling. Let $K[i]$ be the ith byte of the key of length m bytes, and let $j \in \{0, 1, 2, 3\}$. We define a sequence (k_r) of integers modulo 256 and a sequence (π_r) of byte permutations (i.e. permutations of the set $\{0, \dots, 255\}$) as follows.

Let π_0 be the identity permutation, and $k_0 = j$. For $r \neq 257$ let

$$k_{r+1} = k_r + K[i \bmod m] + \pi_r(r \bmod 256) \bmod 256.$$

If $r = 257$, define $k_{257} = j + k_1 + K[i \bmod m] + \pi_{256}(0 \bmod 256)$. The permutations (π_r) are updated as follows:

$$\pi_{r+1}(i) = \begin{cases} \pi_r(k_{r+1}) & \text{if} \quad i = r \bmod 256 \\ \pi_r(r) & \text{if} \quad i = k_{r+1} \\ \pi_r(i) & \text{else} \end{cases}$$

We then let $\sigma_j = \pi_{512}$. So to define σ_j, we are repeating a loop twice in which we take a permutation and then swap certain pairs of images of elements under the permutation to obtain a new permutation.

Finally we define byte permutations $S0 = \sigma_3$, $S1 = \sigma_2\sigma_3$, $S2 = \sigma_1\sigma_2\sigma_3$ and $S3 = \sigma_0\sigma_1\sigma_2\sigma_3$.

The key stream generation. LEVIATHAN is defined by a set of binary tree structures of height 16. Each node of each tree is associated with a triple of words (each of four bytes) $z|y|x$. The triple at the root of the jth tree is $1|0|j/2^{16}$. Key-dependent functions a and b map the triple s at a node to a triple at each of its two descendants, so that its lefthand descendant is $a(s)$ while its righthand descendant is $b(s)$, where $a(s)$ in the kth level is chosen, if the kth bit of j is zero. Otherwise $b(s)$ is chosen.

We apply a function c to the triple at each leaf of each tree to give a word and use the words thus obtained as the keystream.

The functions a, b and c are defined as follows:

$$\begin{aligned} f(z|y|x) &= 2z|SRSRy|LSLSx \\ g(z|y|x) &= 2z+1|LSLSy|SRSR(\bar{x}) \\ d(z|y|x) &= z|x+y+z|2x+y+z \\ c(z|y|x) &= x \oplus y \\ a &= f \circ d \\ b &= g \circ d \end{aligned}$$

where L and R denote rotation left and right respectively by one byte, $\bar{x}$ denotes the complement of x and S denotes a key-dependent S-box given below.

The permutation S is defined as follows. Let the word y consist of the bytes y_0, y_1, y_2 and y_3. Then the image of y under S is defined by

$$\begin{aligned} y_0 &\mapsto S0(y_0) \\ y_1 &\mapsto y_1 \oplus S1(y_0) \\ y_2 &\mapsto y_2 \oplus S2(y_0) \\ y_3 &\mapsto y_3 \oplus S3(y_0) \end{aligned}$$

where the byte permutations $S0$, $S1$, $S2$ and $S3$ are defined in the key scheduling.

3.5.1.2 Security analysis

There is a distinguishing attack on LEVIATHAN faster than exhaustive key search [172]. It shows, that the probability of a collision in 64 bit output words is doubled compared to a random function. Hence LEVIATHAN was not selected for further study in phase II.

3.5.2 LILI-128

LILI-128
is a synchronous stream cipher designed by Ed Dawson and submitted to the NESSIE project. It uses a 128-bit key and an internal memory of 128 bits. LILI-128 was not considered during Phase II of the project.

3.5.2.1 Design

LILI-128
consists of two components, one used for clock control and the other for data generation. Each component consists of an LFSR ($LFSR_c$ and $LFSR_d$) and a function f (f_c and f_d) tapping the LFSRs. LILI-128 can be viewed as a clock-controlled nonlinear filter generator.

The key scheduling. During key scheduling the 128 key bits are loaded directly into the LFSRs. The first 39 key bits are loaded into $LFSR_c$, and the last 89 key bits into $LFSR_d$. Neither $LFSR_c$ nor $LFSR_d$ may be zero.

The clock control component. The clock control component consists of the regularly clocked $LFSR_c$ of length 39 and the function f_c. The feedback poynomial of $LFSR_c$ is

$$x^{39} + x^{35} + x^{33} + x^{31} + x^{17} + x^{15} + x^{14} + x^2 + 1$$

and it produces a maximum length sequence.

$LFSR_c$ is clocked once, then the function f_c takes the contents of the stages 12 and 20 as input and produces an output integer by

$$c = f_c(x_{12}, x_{20}) = 2x_{12} + x_{20} + 1.$$

The data generation component. $LFSR_d$ of length 89 produces a maximum length sequence. Its feedback polynomial is

$$x^{89} + x^{83} + x^{80} + x^{55} + x^{53} + x^{42} + x^{39} + x + 1.$$

After the clock control component has produced the integer c, $LFSR_d$ is clocked c times. The nonlinear function f_d defined by a truth table takes the content of 10 stages of $LFSR_d$ as input and calculates an output bit. This output bit is taken as the new keystream bit.

3.5.2.2 Security analysis

The key of LILI-128 can be recovered faster than exhaustive key by several kinds of time-memory tradoff attacks [28,552] or a fast correlation attack [321]. The last attack has a complexity around 2^{71} bit operations assuming a received sequence of length around 2^{30} bits and a precomputation phase of complexity 2^{79} table lookups. Hence LILI-128 was not selected for further study in phase II.

3.5.3 RC4

RC4 is the de facto standard for stream ciphers (see [542] for a detailed description). The second output byte of RC4 can be easily distinguished from a random one [415]. Although this can be easily overcome, the keystream still can be distinguished from a random output [233] and the key schedule has severe weaknesses [232]. There is also no form of rekeying defined for RC4. Therefore RC4 was not considered by NESSIE.

4. Hash functions

4.1 Introduction

Hash functions are used as a building block in various cryptographic applications. The most important uses are in the protection of information authentication and as a tool for digital signature schemes (see Chapter 7). A hash function is a function that maps an input of arbitrary length into a fixed number of output bits, the *hash value*. In order to be useful for cryptographic applications, a hash function needs to satisfy some additional requirements. Hash functions can be further divided into one-way hash functions and collision-resistant hash functions. We informally give the conditions we require of these different types:

- **A one-way hash function** should be *preimage* and *second preimage* resistant, that is, it should be 'hard' to find a message with a given hash value (preimage) and it should be 'hard' to find a message that hashes to the same value as a given message (second preimage).
- **A collision-resistant hash function** is a one-way hash function for which it is 'hard' to find two distinct messages that hash to the same value.

In some applications additional properties may be required for a hash function, for example pseudo-randomness of the generated output. Note that, in contrast to other cryptographic primitives, the computation of a hash function does *not* depend on any secret information.

Before presenting a detailed analysis of the hash functions studied by the NESSIE project, we first discuss the security requirements and give an overview of common designs and current standards. For a more comprehensive overview of cryptographic primitives for information authentication (including hash functions) we refer to the treatment by Preneel in [530].

4.2 Security requirements

In this section we give practical and formal definitions for one-way and collision-resistant hash functions and describe the general model of an iterated hash function. We then discuss different types of attacks on hash functions and describe the assessment process followed by the project.

[0] Coordinator for this chapter: KUL — Bart Van Rompay

4.2.1 Security model

The following informal definitions for one-way and collision-resistant hash functions were given by Preneel in [530].

A **one-way hash function** is a function h satisfying the following conditions:

1. The argument X can be of arbitrary length and the result $h(X)$ has a fixed length of n bits.
2. The hash function must be one-way in the sense that given a Y in the image of h, it is 'hard' to find a message X such that $h(X) = Y$ (preimage-resistance) and given X and $h(X)$ it is 'hard' to find a message $X' \neq X$ such that $h(X') = h(X)$ (second preimage-resistance).

A **collision-resistant hash function** is a function h satisfying the following conditions:

1. The argument X can be of arbitrary length and the result $h(X)$ has a fixed length of n bits.
2. The hash function must be one-way, i.e., preimage-resistant and second preimage-resistant.
3. The hash function must be collision-resistant: this means that it is 'hard' to find two distinct messages that hash to the same result (i.e., find X and X' ($X' \neq X$) such that $h(X) = h(X')$).

Note that if an attacker can find a second preimage, he can also find a collision. Therefore the second preimage condition in this definition is redundant. However, preimage-resistance is not always implied by collision resistance.

Most hash functions are iterated constructions, in the sense that they are based on a compression function with fixed size inputs; they process every message block in a similar way. The input X is padded by an unambiguous padding rule to a multiple of the block size. Typically this also includes adding the total length in bits of the input. The padded input is then divided into t blocks denoted X_1 through X_t. The hash function involves a *compression function* f and a *chaining variable* H_i between stage $i-1$ and stage i:

$$\begin{aligned} H_0 &= IV \ , \\ H_i &= f(H_{i-1}, X_i) \ , \ 1 \leq i \leq t \ , \\ h(X) &= g(H_t) \ . \end{aligned}$$

Here IV denotes the Initial Value (a constant which should be defined as part of the description of the hash function) and g denotes the (optional) *output transformation*. A collision-resistant compression function can be extended to a collision-resistant hash function taking arbitrary length inputs. For a detailed discussion on the relation between a compression function and a hash function that is built from it, we refer to [530].

4.2.1.1 Formal definitions

Before discussing security aspects we first give more precise definitions of a hash function and its cryptographic properties. The following formal definitions for a hash function, a compression function and an iterated hash construction are similar to those given by Black *et al.* in [93].

Definition 4.1. *A* **hash function** *is a function* $h : \mathcal{D} \rightarrow \mathcal{R}$ *where the domain* $\mathcal{D} = \{0,1\}^*$ *and the range* $\mathcal{R} = \{0,1\}^n$ *for some* $n \geq 1$*.*

Definition 4.2. *A* **compression function** *is a function* $f : \mathcal{D} \rightarrow \mathcal{R}$ *where* $\mathcal{D} = \{0,1\}^a \times \{0,1\}^b$ *and* $\mathcal{R} = \{0,1\}^n$ *for some* $a, b, n \geq 1$ *with* $a + b \geq n$*.*

Definition 4.3. *The* **iterated hash** *of the compression function* $f : (\{0,1\}^n \times \{0,1\}^b) \rightarrow \{0,1\}^n$ *is the hash function* $h : (\{0,1\}^b)^* \rightarrow \{0,1\}^n$ *defined by* $h(X_1 \dots X_t) = H_t$ *where* $H_i = f(H_{i-1}, X_i)$ *for* $1 \leq i \leq t$ *(set* $H_0 = IV$*).*

The following definitions for (second) preimage-resistance and for collision-resistance were given by Brown in [126]. These definitions are somewhat simplified because we consider a fixed hash function rather than a family of hash functions. Therefore our assumptions, particularly the assumption about collision-resistance, are stronger than the usual assumptions for families of hash functions.

Definition 4.4 (Preimage-resistance). *A hash function* $h : \{0,1\}^* \rightarrow \mathcal{R}$ *is preimage-resistant of strength* (t, ϵ) *if there exists no probabilistic algorithm* I_h *that accepts input* $Y \in_R \mathcal{R}$ *and outputs a value* $X \in \{0,1\}^*$ *in running time at most* t*, where* $h(X) = Y$ *with probability at least* ϵ*, assessed over the random choices of both* Y *and* I_h*.*

Definition 4.5 (Second preimage-resistance). *Let* $\mathcal{S}$ *be a finite subset of* $\{0,1\}^*$*. A hash function* $h : \{0,1\}^* \rightarrow \mathcal{R}$ *is second preimage-resistant of strength* $(t, \epsilon, \mathcal{S})$ *if there exists no probabilistic algorithm* S_h *that accepts input* $X \in_R \mathcal{S}$ *and outputs a value* $X' \in \{0,1\}^*$ *in running time at most* t*, where* $X' \neq X$ *and* $h(X') = h(X)$ *with probability at least* ϵ*, assessed over the random choices of both* X *and* S_h*.*

Note: One can define a stronger notion of (second) preimage resistance, where the value $Y \in \mathcal{R}$ (or the value $X \in \mathcal{S}$) is a fixed point rather than a random point, and where one maximises over all such points.

Definition 4.6 (Collision-resistance). *A hash function* $h : \{0,1\}^* \rightarrow \mathcal{R}$ *is collision-resistant of strength* (t, ϵ) *if no probabilistic algorithm* C_h *is known that outputs values* $X, X' \in \{0,1\}^*$ *in running time at most* t*, where* $X' \neq X$ *and* $h(X) = h(X')$ *with probability at least* ϵ*, assessed over the random choices of* C_h*.*

Note: Since $\{0,1\}^*$ is infinite and $\mathcal{R}$ is finite, collisions for h do exist. And since C_h has no input, there exists a very efficient algorithm, namely one that immediately outputs (X, X') for some fixed collision. Nevertheless, for a good hash function with $\mathcal{R} = \{0,1\}^n$, the best collision-finding algorithm *known* with high success probability should have a running time of about $2^{n/2}$ (see Sect. 4.2.2).

4.2.2 Classification of attacks

For one-way and collision-resistant hash functions the computation of the function does not involve any secret information. Hence, it is not meaningful to distinguish attacks depending on the information available to an attacker. The goal of the attacker is to find a *preimage* or *second preimage* for the hash function, or to generate a *collision*. Collision-resistance is not required in all applications (the most important case where it is required is in conjunction with digital signatures), which is the reason for the separate category of one-way hash functions. Also it can be noted that some attacks find a preimage or collision only for random messages, while others leave the attacker enough freedom to choose (part of) the messages.

Note that the security of a hash function can be examined through analysis of its compression function. A collision-resistant compression function can be extended to a collision-resistant hash function taking arbitrary length inputs. On the other hand, attacks on the compression function do not necessarily mean attacks on the hash function: an attack that finds preimages or collisions for f with chosen H_{i-1} leads to an attack on h, but an attack that finds preimages or collisions for f with a random H_{i-1} does not yield an attack on h (unless when the IV can be changed). Another example is an attack that finds collisions for the compression function where the initial chaining value is not the same for both messages ($H_{i-1} \neq H'_{i-1}$). Attacks on the compression function which cannot be extended are seen as certificational weaknesses of the hash function involved.

We now briefly describe the best known attacks on hash functions as in [530].

Random (Second) Preimage Attack

This attack can be applied to any hash function and depends only on the size n in bits of the hash result. An attacker simply selects a random message and hopes that a given hash result occurs. If the hash function has the required “random” behaviour, his probability of success equals $1/2^n$. This attack can be carried out off-line and in parallel. It is expected that a (second) preimage can be found in approximately 2^n operations.

Birthday Attack

This attack can be applied to any hash function and depends only on the size n in bits of the hash result. The birthday paradox states that for a group of 23 people, the probability that at least two people have a common birthday exceeds $1/2$. Intuitively one expects that the probability is much lower. This can be exploited to find collisions for a hash function in the following way: an adversary generates r_1 variations on a bogus message and r_2 variations on a genuine message. The expected number of collisions equals $r_1 \cdot r_2/2^n$. The probability of finding a bogus message and a genuine message that hash to the same result is given by $1 - \exp(-r_1 \cdot r_2/2^n)$, which is about 63% when $r = r_1 = r_2 = 2^{\frac{n}{2}}$. Finding the collision does not require r^2 operations: after sorting the data, which requires $O(r \log r)$ operations, comparison is easy. One can reduce the memory requirements for collision search by translating the problem to the detection of a cycle in an iterated mapping.

Meet-in-the-Middle Attack

This attack — as well as the attacks described below — depends on some properties of the compression function. It is a variation on the birthday attack, but instead of comparing the hash result, one compares intermediate chaining variables. If the attack can be applied to a hash function, it enables an adversary to construct a (second) preimage, which is not possible for a simple birthday attack. The opponent generates r_1 variations on the first part of a bogus message and r_2 variations on the last part. Starting from the initial value and going backwards from the hash result, the probability for a matching intermediate variable is again given by $1 - \exp(-r_1 \cdot r_2/2^n)$. The cycle finding algorithm can be used to perform a meet-in-the-middle attack with negligible storage.

Correcting Block Attack

This attack consists of substituting all blocks of the message except for one or more blocks. For a (second) preimage attack, one chooses an arbitrary message X and finds one or more correcting blocks Y such that $h(X\|Y)$ takes a certain value. For a collision attack, one starts with two arbitrary messages X and X' and appends one or more correcting blocks denoted with Y and Y', such that the extended messages $X\|Y$ and $X'\|Y'$ have the same hash result.

Fixed Point Attack

The idea of this attack is to look for a H_{i-1} and X_i such that $f(H_{i-1}, X_i) = H_{i-1}$. If the chaining variable is equal to H_{i-1}, it is possible to insert an arbitrary number of blocks equal to X_i without modifying the hash result. Producing collisions or a second preimage with this attack is only possible if the chaining variable can be made equal to H_{i-1}: this is the case if IV can be chosen equal to a specific value, or if a large number of fixed points can be constructed (i.e., if one can find an X_i for a significant fraction of H_{i-1}'s). Of course this attack can be extended to fixed points that occur after more than one iteration.

Differential Attacks

Differential cryptanalysis [78] is a powerful tool for the analysis of not only block ciphers (see Chapter 2) but also of hash functions. This attack method examines input differences to a compression function and the corresponding output differences. A collision is obtained if the output difference is zero.

Analytical Weaknesses

A large number of attacks have been based on blocking the diffusion of the data input to the hash function: this means that changes have no effect or can be cancelled out easily in a next stage. Among the most remarkable attacks in this class are the attacks developed by Dobbertin on the MDx-family [204, 205]. They combine conventional optimisation techniques (simulated annealing, genetic algorithms) with conventional cryptanalytic techniques. Another property of these attacks is that differences are only controlled in certain points of the algorithms; this in contrast to differential cryptanalysis, where typically all intermediate differences are controlled to a certain extent.

Side-Channel Attacks

Side-channel attacks are a major threat for all implementations of cryptographic algorithms. Hash functions are only vulnerable to this type of attack when part of the input to the function is secret, for example when a hash function is used in a construction for a message authentication code (MAC) as described in Chapter 5, or when it is used in a key derivation function (KDF) as described in Chapter 6. In such a case side-channel attacks on hash functions are similar as for other symmetric primitives. For a detailed discussion on side-channel attacks and countermeasures that can be applied to protect an implementation we refer to Annex A.

4.2.3 Assessment process

The hash function submissions were assessed with reference to the above generic common hash function attacks and by specific attacks when appropriate. Statistical analysis was carried out for various input lengths. Hash functions were submitted to the dependence test and linear factors test described in Sect. 2.2.4. If the NESSIE submissions were based on compression functions, these were also submitted to the two tests. Furthermore the stream cipher tests described in Sect. 2.2.4 were also applied to the hashes both in output feedback and counter mode. None of the hash functions tested exhibited any anomalous behaviour.

4.3 Overview of the common designs

In this section we give an overview of those hash functions which are most commonly used in practice today. Most known hash functions are iterated constructions that are based on a compression function with fixed size input. Sometimes, this compression function is based on a block cipher or on modular arithmetic; in other cases it is built from scratch specifically for the purpose of hashing. We conclude the section with a summary of the procedures which have been undertaken by several organisations for the standardisation of hash functions.

4.3.1 Hash functions based on block ciphers

In systems where a block cipher implementation is already available, a hash function can be obtained at little extra cost (in design, evaluation and implementation) by constructing it from the underlying block cipher. A disadvantage is that these hash functions are usually slower than the dedicated proposals (especially when the block cipher used has a slow key schedule). Furthermore, the use of a block cipher in a different context may reveal weaknesses which are not relevant to encryption (it is still an open problem which requirements for a block cipher are sufficient in order to produce a secure hash function).

Most block cipher based hash functions belong to one of two classes: those producing a hash value of single block length and those producing a hash value of double block length. In case the AES (Advanced Encryption Standard, see Chapter 2) is used as underlying block cipher, the resulting output lengths of these hash function classes are 128 and 256 bits respectively. Another relevant characteristic is the rate of the hash function, which is equal to the number of blocks that are hashed for each computation of the encryption function.

There are several *single block length hash functions* of rate 1 with provable black-box security (assuming the underlying block cipher satisfies the required randomness properties). Two dual constructions are those attributed to Matyas-Meyer-Oseas and Davies-Meyer. In each step of the iteration the previous value of the chaining variable (H_{i-1}) and the message block to be processed (X_i) serve as key and plaintext of the encryption function (or vice versa), and there is an extra feed-forward with the purpose of making the compression function uninvertible. For a block cipher $E_K(X)$ (with key K and plaintext X) the compression function of the two constructions is as follows:

- Matyas-Meyer-Oseas: $H_i = E_{g(H_{i-1})}(X_i) \oplus X_i \ ,$
- Davies-Meyer: $H_i = E_{X_i}(H_{i-1}) \oplus H_{i-1} \ .$

Here g is a function that maps H_{i-1} to a key suitable for E. The Miyaguchi-Preneel hashing scheme is a variation on Matyas-Meyer-Oseas, where the only difference is that the output H_{i-1} from the previous stage is also XORed to that of the current stage.

- Miyaguchi-Preneel: $H_i = E_{g(H_{i-1})}(X_i) \oplus X_i \oplus H_{i-1} \ .$

The most common *double block length hash functions* are MDC-2 and MDC-4 with a rate of respectively 1/2 and 1/4 (all proposed schemes with rate 1 have been broken [364]). MDC-2 requires two parallel block encryptions in each iteration step, and the compression function of MDC-4 has two sequential executions of the MDC-2 compression function. The level of security of these two schemes is less than suggested by the output size.

The Matyas-Meyer-Oseas scheme and MDC-2 are included in ISO/IEC 10118-2 [305], a standard for hash functions using an (unspecified) block cipher. The NESSIE submission Whirlpool (see Sect. 4.4.1) is based on the Miyaguchi-Preneel hashing mode of an internal block cipher.

4.3.2 Hash functions based on modular arithmetic

A hash function can also use modular arithmetic as the basis of its compression function, allowing the re-use of existing implementations of modular arithmetic (such as in public-key systems). The biggest disadvantage is the low speed of these constructions (especially compared to dedicated hash functions). Many of the proposals which use modular arithmetic have been broken. The experience with these attacks has led to the design of the MASH-1 and MASH-2 hash functions.

The compression function of MASH-1 is based on a modular squaring operation, where the modulus M is chosen as a composite of sufficient bitlength m (making it infeasible to factor M). The message block input X_i is first expanded with redundancy bits to a block Y_i of double length. The block Y_i is then added bitwise to the previous value of the chaining variable H_{i-1} (both Y_i and H_{i-1} have bitlength n, chosen as the largest multiple of 16 less than m). After the squaring operation a feed-forward is applied with H_{i-1}. This results in the following equation for the compression function:

$$H_i = ((((H_{i-1} \oplus Y_i) \vee A)^2 \bmod M) \dashv n) \oplus H_{i-1} \ ,$$

where A is a constant forcing the four most significant bits to 1, prior to squaring; and $\dashv n$ denotes truncation of the result of the squaring operation to the n least significant bits.

The security of this construction depends in part on the difficulty of extracting modular roots (for a composite of unknown factorisation). The redundancy bits are vital as well, resulting in the following security level: matching a given hash value requires $2^{n/2}$ operations; finding a collision requires $n \times 2^{n/4}$ operations.

MASH-2 is a variant of MASH-1, the only difference is that it uses a modular exponentiation with exponent $e = 2^8 + 1$ (instead of modular squaring with $e = 2$). Both MASH-1 and MASH-2 are included in ISO/IEC 10118-4 [307], a standard for hash functions using modular arithmetic. This standard also defines an additional output transformation that must be used to reduce the length of the hash value to $n/2$ bits or less.

4.3.3 Dedicated hash functions

Dedicated hash functions are functions specifically designed for the purpose of hashing, with optimised performance in mind. The hash functions of this class which have received the most attention are those based on the design of the MD4 algorithm. While MD4, which dates from 1990, has been broken with respect to collision-resistance, the algorithms derived from it have benefited from the experience gained with the cryptanalysis of this type of hash functions.

The MD4-based algorithms generally divide the message block X_i and the chaining variable H_{i-1} in words of 32 bits (or 64 bits for some of the new proposals). The compression function updates the chaining variable to a new value H_i, making use of the current message block. The computation is divided into a number of sequential steps, where each step updates the value of one register — containing one word of the chaining variable — applying one message word (and depending on the other registers). The compression function can also be divided into a number of rounds, where in general each round uses all words from the message block exactly once.

As an example we describe the step operation of MD4, which is of the following form:

$$A = (A + f(B, C, D) + X[j] + y)^{<<s[j]} \ .$$

Here (A, B, C, D) are the registers containing the four 32-bit words of the chaining variable. The notation $+$ means addition modulo 2^{32} and $(\cdot)^{<<s}$ means bitwise rotation (circular shift) over s positions to the left. f is a non-linear function working at bit level, $X[j]$ is the message word applied in step j, y an additive constant and $s[j]$ a rotation constant. The function f and constant y differ in different rounds (MD4 has three rounds using the multiplexer, majority and exclusive-or functions), as does the ordering in which the message words are applied throughout the steps of one round.

The step operations are reversible (one can calculate backwards easily), but at the end of the compression function there is a feed-forward operation which adds the initial value of the registers to their final value in order to compute the chaining variable H_i. This makes the compression function uninvertible. The chaining variable derived by the last application of the compression function (which works on a message block with padding and length information) serves as the result of the hash function.

Several algorithms have been derived from MD4, applying different ideas in order to increase the security against preimage and collision attacks. These ideas include the use of more rounds, slightly different step operations and longer chaining variables (which also means a longer hash result). In particular, the SHA family[1] uses a procedure for expansion of the message block in order to calculate a different word for every step (instead of just reordering the message words between different rounds). The RIPEMD family uses two parallel lines of computation consisting of modified versions of MD4. In Table 4.22 we summarise the main differences between the different algorithms.

Table 4.22. Summary of selected MD4-based hash functions. All sizes are given in bits. Note that for SHA-256, SHA-384 and SHA-512 there is no clear distinction in rounds.

Algorithm	output size	block size	word size	rounds × steps per round
MD4	128	512	32	3×16
MD5	128	512	32	4×16
RIPEMD-128	128	512	32	4×16 twice (in parallel)
RIPEMD-160	160	512	32	5×16 twice (in parallel)
SHA-1	160	512	32	4×20
SHA-256	256	512	32	1×64
SHA-384	384	1024	64	1×80
SHA-512	512	1024	64	1×80

4.3.4 Current standards

Several organisations have taken initiatives for the standardisation of hash functions. The SHA-1 hash function is a U.S. government standard, developed by NIST as Federal Information Processing Standard (FIPS) 180-1, and it can be

[1] A detailed description of these hash functions is given in Sect. 4.4.2 and Sect. 4.4.3.

used in conjunction with the DSA standard for digital signatures. Recently NIST has updated this standard to FIPS 180-2 [472] which includes, besides SHA-1, also three new hash functions — SHA-256, SHA-384 and SHA-512 — with longer output lengths (in order to match the security level of the new block cipher standard AES). ANSI has adopted hash functions in its public-key based banking standards: standard X9.30 [19] specifies SHA-1 to be used in conjunction with DSA; standard X9.31 [20] specifies MDC-2 to be used in conjunction with an RSA-based signature scheme.

ISO/IEC has developed standard 10118 for hash functions, with separate parts for different classes of hash functions. Part 10118-2 [305] details hash functions based on block ciphers, more specifically the Matyas-Meyer-Oseas construction, a block cipher independent MDC-2 and two more functions producing a hash value of double and triple block length respectively. Part 10118-3 [306] specifies three dedicated algorithms: RIPEMD-128, RIPEMD-160 and SHA-1. This part of the standard is currently being revised, with an ongoing assessment of new cryptographic primitives to be adopted as ISO standards. Besides the original three algorithms, the following hash functions are being considered: SHA-256, SHA-384, SHA-512 and WHIRLPOOL. Part 10118-4 [307] describes the MASH-1 and MASH-2 hash functions which use modular arithmetic.

4.4 Hash functions considered during Phase II

The WHIRLPOOL algorithm was submitted to NESSIE and selected for study during phase II of the NESSIE project. Furthermore, because they are standards of NIST [472], and under discussion in ISO [306], the algorithms SHA-1 and SHA-256, SHA-384 and SHA-512 were selected for study during NESSIE phase II.

4.4.1 Whirlpool

WHIRLPOOL is a hash function designed by Paulo Barreto and Vincent Rijmen [38] and submitted to the NESSIE project as a (conjectured) collision-resistant hash function. The algorithm is byte-oriented and operates on blocks of 512 bits, producing a message digest (hash value) of 512 bits.

4.4.1.1 The design

The design of WHIRLPOOL consists of the iterative application of a compression function which is based on an underlying dedicated 512-bit block cipher that uses a 512-bit key and runs in 10 rounds. In the following we first describe the individual components that build up WHIRLPOOL, and then specify the complete hash function in terms of these components.

Input and output.
The hash state is internally viewed as a matrix in $\mathcal{M}_{8\times 8}[\mathrm{GF}(2^8)]$. Therefore, 512-bit data blocks (externally represented as byte arrays by sequentially grouping

the bits in 8-bit chunks) must be mapped to and from this matrix format. This is done by the function $\mu : \mathrm{GF}(2^8)^{64} \to \mathcal{M}_{8\times 8}[\mathrm{GF}(2^8)]$ and its inverse:

$$\mu(a) = b \Leftrightarrow b_{ij} = a_{8i+j},\ 0 \leqslant i, j \leqslant 7.$$

The nonlinear layer γ.
The function $\gamma : \mathcal{M}_{8\times 8}[\mathrm{GF}(2^8)] \to \mathcal{M}_{8\times 8}[\mathrm{GF}(2^8)]$ consists of the parallel application of a nonlinear substitution box $S : \mathrm{GF}(2^8) \to \mathrm{GF}(2^8)$, $x \mapsto S[x]$ to all bytes of the argument individually:

$$\gamma(a) = b \Leftrightarrow b_{ij} = S[a_{ij}],\ 0 \leqslant i, j \leqslant 7.$$

The substitution box proposed in the tweaked version of Whirlpool is different from the one proposed in the original version. It is built in a simple way from smaller 4-bit substitution boxes. For its description we refer to the new algorithm specification [38].

The cyclical permutation π.
The permutation $\pi : \mathcal{M}_{8\times 8}[\mathrm{GF}(2^8)] \to \mathcal{M}_{8\times 8}[\mathrm{GF}(2^8)]$ cyclically shifts each column of its argument independently, so that column j is shifted downwards by j positions:

$$\pi(a) = b \Leftrightarrow b_{ij} = a_{(i-j) \bmod 8, j},\ 0 \leqslant i, j \leqslant 7.$$

The purpose of π is to disperse the bytes of each row among all rows.

The linear diffusion layer θ.
The diffusion layer $\theta : \mathcal{M}_{8\times 8}[\mathrm{GF}(2^8)] \to \mathcal{M}_{8\times 8}[\mathrm{GF}(2^8)]$ is a linear mapping based on the $[16, 8, 9]$ code with generator matrix $G_C = [I\, C]$ where

$$C = \begin{bmatrix}
\mathtt{01}_x & \mathtt{01}_x & \mathtt{03}_x & \mathtt{01}_x & \mathtt{05}_x & \mathtt{08}_x & \mathtt{09}_x & \mathtt{05}_x \\
\mathtt{05}_x & \mathtt{01}_x & \mathtt{01}_x & \mathtt{03}_x & \mathtt{01}_x & \mathtt{05}_x & \mathtt{08}_x & \mathtt{09}_x \\
\mathtt{09}_x & \mathtt{05}_x & \mathtt{01}_x & \mathtt{01}_x & \mathtt{03}_x & \mathtt{01}_x & \mathtt{05}_x & \mathtt{08}_x \\
\mathtt{08}_x & \mathtt{09}_x & \mathtt{05}_x & \mathtt{01}_x & \mathtt{01}_x & \mathtt{03}_x & \mathtt{01}_x & \mathtt{05}_x \\
\mathtt{05}_x & \mathtt{08}_x & \mathtt{09}_x & \mathtt{05}_x & \mathtt{01}_x & \mathtt{01}_x & \mathtt{03}_x & \mathtt{01}_x \\
\mathtt{01}_x & \mathtt{05}_x & \mathtt{08}_x & \mathtt{09}_x & \mathtt{05}_x & \mathtt{01}_x & \mathtt{01}_x & \mathtt{03}_x \\
\mathtt{03}_x & \mathtt{01}_x & \mathtt{05}_x & \mathtt{08}_x & \mathtt{09}_x & \mathtt{05}_x & \mathtt{01}_x & \mathtt{01}_x \\
\mathtt{01}_x & \mathtt{03}_x & \mathtt{01}_x & \mathtt{05}_x & \mathtt{08}_x & \mathtt{09}_x & \mathtt{05}_x & \mathtt{01}_x
\end{bmatrix},$$

so that $\theta(a) = b \Leftrightarrow b = a \cdot C$. The effect of θ is to mix the bytes in each state row.

The key addition $\sigma[k]$.
The affine key addition $\sigma[k] : \mathcal{M}_{8\times 8}[\mathrm{GF}(2^8)] \to \mathcal{M}_{8\times 8}[\mathrm{GF}(2^8)]$ consists of the bitwise addition (exor) of a key matrix $k \in \mathcal{M}_{8\times 8}[\mathrm{GF}(2^8)]$:

$$\sigma[k](a) = b \Leftrightarrow b_{ij} = a_{ij} \oplus k_{ij},\ 0 \leqslant i, j \leqslant 7.$$

This mapping is also used to introduce round constants in the key schedule.

The round constants $\boldsymbol{c^r}$.
The round constant for the r-th round, $r > 0$, is a matrix $c^r \in \mathcal{M}_{8\times 8}[\mathrm{GF}(2^8)]$, defined as:

$$\begin{array}{rcll} c^r_{0j} & \equiv & S[8(r-1)+j], & 0 \leqslant j \leqslant 7, \\ c^r_{ij} & \equiv & 0, & 1 \leqslant i \leqslant 7, 0 \leqslant j \leqslant 7. \end{array}$$

The round function $\boldsymbol{\rho[k]}$.
The r-th round function is the composite mapping $\rho[k] : \mathcal{M}_{8\times 8}[\mathrm{GF}(2^8)] \to \mathcal{M}_{8\times 8}[\mathrm{GF}(2^8)]$, parameterised by the key matrix $k \in \mathcal{M}_{8\times 8}[\mathrm{GF}(2^8)]$ and given by:

$$\rho[k] \equiv \sigma[k] \circ \theta \circ \pi \circ \gamma.$$

The key schedule.
The key schedule expands the 512-bit cipher key $K \in \mathcal{M}_{8\times 8}[\mathrm{GF}(2^8)]$ onto a sequence of round keys $K^0, \ldots, K^{10}$:

$$\begin{array}{rcl} K^0 & = & K, \\ K^r & = & \rho[c^r](K^{r-1}),\ 1 \leqslant r \leqslant 10, \end{array}$$

The internal block cipher $\boldsymbol{W}$.
The dedicated 512-bit block cipher $W_K : \mathcal{M}_{8\times 8}[\mathrm{GF}(2^8)] \to \mathcal{M}_{8\times 8}[\mathrm{GF}(2^8)]$, parameterised by the 512-bit cipher key K, is defined as

$$W_K = \left(\bigcirc_{1}^{r=10} \rho[K^r] \right) \circ \sigma[K^0],$$

where the round keys $K^0, \ldots, K^{10}$ are derived from K by the key schedule.

Padding and MD-strengthening.
Before being subjected to the hashing operation, a message M of bit length $L < 2^{256}$ is padded with a 1-bit, then with as few 0-bits as necessary to obtain a bit string whose length is an odd multiple of 256, and finally with the 256-bit right-justified binary representation of L, resulting in the padded message m, partitioned in t 512-bit blocks $m_1, \ldots, m_t$.

The compression function.
Whirlpool iterates the Miyaguchi-Preneel hashing scheme over the t padded message blocks m_i, $1 \leqslant i \leqslant t$, using the dedicated 512-bit block cipher W:

$$\begin{array}{rcl} \eta_i & = & \mu(m_i),\ 1 \leqslant i \leqslant t \\ H_0 & = & \mu(IV), \\ H_i & = & W_{H_{i-1}}(\eta_i) \oplus \eta_i \oplus H_{i-1},\ 1 \leqslant i \leqslant t, \end{array}$$

where IV (the *initialisation vector*) is a string of 512 0-bits.

Message digest computation.
The Whirlpool message digest for message M is defined as the output H_t of the compression function, mapped back to a bit string:

$$\text{Whirlpool}(M) \equiv \mu^{-1}(H_t).$$

4.4.1.2 Security analysis

The Miyaguchi-Preneel scheme is one of the few still unbroken methods to construct a hash function from an underlying block cipher. In particular, it is "provably secure" if certain ideal properties hold for the underlying block cipher (in [93] Black *et al.* prove tight upper and lower bounds for the collision-resistance and preimage-resistance of this construction, based on a black-box model of the encryption algorithm).

The block cipher W that forms the basis of Whirlpool is very similar to the AES algorithm Rijndael. The main difference between W and Rijndael is that Rijndael supports blocklengths of 128, 192 and 256 bits, while W only works on 512-bit blocks. Rijndael has received quite a bit of analysis during and after the AES process, and given the similarities between Rijndael and W, some of it may carry over to Whirlpool. The designers state three criteria that define the level of security. Assume we take as hash result any n-bit substring of the output of Whirlpool, then the criteria are:

- The workload of generating a collision is expected to be of the order $2^{n/2}$ (collision-resistant).
- Given an n-bit value, the workload of finding a message that hashes to that value is of the order 2^n (preimage-resistant).
- Given a message (and its hash result), the workload of finding a different message that has the same hash value is of the order 2^n (second preimage-resistant).

No attacks have been reported on Whirlpool that falsify these statements. In the following we discuss some observations that have been made and an attack on a reduced round version of Whirlpool.

Choice of the substitution box.
In its nonlinear layer Whirlpool uses a substitution box to map 8-bit inputs into 8-bit outputs. The originally submitted form of Whirlpool used a pseudo-randomly generated substitution box, chosen to satisfy certain conditions with respect to differential and linear analysis, and with respect to the non-linear order. However, a flaw that went unnoticed caused the value of the linear parameter to be incorrectly reported. Therefore, in the tweaked version of Whirlpool an alternative substitution box was described satisfying the design conditions. This new substitution box is built in a simple way from smaller 4-bit substitution boxes.[2]

[2] The new substitution box also leads to more efficient hardware implementations.

Non-random properties of reduced-round Whirlpool.

In [360] it is shown that a variant of the hash function Whirlpool where the number of rounds in the compression function is reduced from ten to six (or less), exhibits some non-random properties. The observations have not shown to be a weakness for Whirlpool. In the following we summarise the analysis of [360].

Consider a collection of 256 texts, which have different values in one byte and equal values in each of the remaining 63 bytes. Then it follows that after two rounds of encryption the texts take all 256 values in each of the 64 bytes, and that after three rounds of encryption the sum of the 256 bytes in each position is zero. Such a structure has been called an integral (see Sect. 2.2.3.15). Also, note that there are 63 other similar structures, since the position of the non-constant byte in the plaintexts can be in any of the 64 positions.

The three-round integral described above can be extended to four rounds by considering a structure with 2^{64} texts. The main observation is that after one round one has all 2^{64} values in the top row and that the remaining three rounds can be seen as a collection of 2^{56} variants of the 3-round integral. Since the texts in each integral sum to zero in any byte after the fourth round, so does the sum of all 2^{64} texts.

In much the same manner, one can define a three-round backwards integral through three rounds of the inverse cipher of W. In versions of W reduced to six rounds one can then combine the first three rounds of the four-round forwards integral and the three-round backwards integral to cover all rounds of the cipher with probability one. We do this by starting our computation after the third round of W. By a first glance it appears that the two three-round integrals cannot be combined.

However, choose a structure of 2^{120} texts such that the fifteen bytes consisting of the eight bytes of the top row and byte j in row $8 - j$ for $j = 1 \dots 7$ take all possible values. One can view these texts as a collection of 2^{56} 3-round forwards integrals, but one can also view this as a collection of 2^{56} 3-round backwards integrals. Therefore, both when one computes forward three rounds and backward three rounds, the resulting texts will take the values in each byte equally many times.

Thus, with time complexity 2^{120} one can find a collection of 2^{120} inputs to W reduced to six rounds, such that each byte in both the inputs and outputs takes all values equally many times. It is claimed that this distinguishes W reduced to six rounds from a randomly chosen 512-bit permutation. We conjecture that to find a structure of 2^{120} texts with properties similar to the ones outlined for W reduced to six rounds will require the generation of at least 2^{128} outputs and a considerable amount of memory.

In versions of W reduced to seven rounds one can combine the full four-round forwards integral and the three-round backwards integral to cover seven rounds with probability one. However it is unclear whether such an integral can be used to effectively distinguish W reduced to seven rounds from a randomly chosen 512-bit permutation. In recent work David Wagner shows techniques for solving a generalised birthday problem. Although these techniques do not seem to apply directly to our problem, it is possible that they can be adapted hereto.

Quadratic relations in Whirlpool.

In [361] it is examined whether there exist quadratic relations with certainty over the input and output bits of the substitution boxes of Whirlpool. It has been shown by Courtois and Pieprzyk [165] that for the substitution boxes of Rijndael and Serpent there exist quadratic equations in the input and output bits which hold with probability one. Such equations always exist for n-bit to n-bit S-boxes where $n \leq 6$, but not in general for $n > 6$.

Therefore we have investigated whether such quadratic relations exist for the S-box of Whirlpool. This S-box is a permutation of eight bits. There are 137 possible terms of degree at most two in a multivariate expression of the eight input and output bits. It is a simple matter to check whether such equations exist simply by computing the kernel of a 256 times 137 binary matrix. We implemented this in Maple. It was found that there are no quadratic relations for the full S-box of Whirlpool. However, after the tweak, the S-box of Whirlpool is built in a simple way from 4-bit S-boxes. This makes it easy to write a small system of multivariate quadratic equations for Whirlpool. Security of Whirlpool against algebraic attacks is a matter of further research.

Branch number and updated diffusion matrix.

In [579] Shirai and Shibutani announced a flaw in the Whirlpool diffusion matrix, that makes its branch number suboptimal (in other words, the underlying code is not an MDS code). Although this flaw *per se* does not seem to introduce an effective vulnerability, the designers decided to propose a replacement matrix displaying optimal branch number (and thus keeping the existing security analysis unchanged).

The diffusion matrix is used in the linear diffusion layer θ. The new matrix is based on the $[16, 8, 9]$ MDS code with generator matrix $G_C = [I\,C]$, where:

$$C = \begin{bmatrix}
\mathtt{01}_{\mathtt{x}} & \mathtt{01}_{\mathtt{x}} & \mathtt{04}_{\mathtt{x}} & \mathtt{01}_{\mathtt{x}} & \mathtt{08}_{\mathtt{x}} & \mathtt{05}_{\mathtt{x}} & \mathtt{02}_{\mathtt{x}} & \mathtt{09}_{\mathtt{x}} \\
\mathtt{09}_{\mathtt{x}} & \mathtt{01}_{\mathtt{x}} & \mathtt{01}_{\mathtt{x}} & \mathtt{04}_{\mathtt{x}} & \mathtt{01}_{\mathtt{x}} & \mathtt{08}_{\mathtt{x}} & \mathtt{05}_{\mathtt{x}} & \mathtt{02}_{\mathtt{x}} \\
\mathtt{02}_{\mathtt{x}} & \mathtt{09}_{\mathtt{x}} & \mathtt{01}_{\mathtt{x}} & \mathtt{01}_{\mathtt{x}} & \mathtt{04}_{\mathtt{x}} & \mathtt{01}_{\mathtt{x}} & \mathtt{08}_{\mathtt{x}} & \mathtt{05}_{\mathtt{x}} \\
\mathtt{05}_{\mathtt{x}} & \mathtt{02}_{\mathtt{x}} & \mathtt{09}_{\mathtt{x}} & \mathtt{01}_{\mathtt{x}} & \mathtt{01}_{\mathtt{x}} & \mathtt{04}_{\mathtt{x}} & \mathtt{01}_{\mathtt{x}} & \mathtt{08}_{\mathtt{x}} \\
\mathtt{08}_{\mathtt{x}} & \mathtt{05}_{\mathtt{x}} & \mathtt{02}_{\mathtt{x}} & \mathtt{09}_{\mathtt{x}} & \mathtt{01}_{\mathtt{x}} & \mathtt{01}_{\mathtt{x}} & \mathtt{04}_{\mathtt{x}} & \mathtt{01}_{\mathtt{x}} \\
\mathtt{01}_{\mathtt{x}} & \mathtt{08}_{\mathtt{x}} & \mathtt{05}_{\mathtt{x}} & \mathtt{02}_{\mathtt{x}} & \mathtt{09}_{\mathtt{x}} & \mathtt{01}_{\mathtt{x}} & \mathtt{01}_{\mathtt{x}} & \mathtt{04}_{\mathtt{x}} \\
\mathtt{04}_{\mathtt{x}} & \mathtt{01}_{\mathtt{x}} & \mathtt{08}_{\mathtt{x}} & \mathtt{05}_{\mathtt{x}} & \mathtt{02}_{\mathtt{x}} & \mathtt{09}_{\mathtt{x}} & \mathtt{01}_{\mathtt{x}} & \mathtt{01}_{\mathtt{x}} \\
\mathtt{01}_{\mathtt{x}} & \mathtt{04}_{\mathtt{x}} & \mathtt{01}_{\mathtt{x}} & \mathtt{08}_{\mathtt{x}} & \mathtt{05}_{\mathtt{x}} & \mathtt{02}_{\mathtt{x}} & \mathtt{09}_{\mathtt{x}} & \mathtt{01}_{\mathtt{x}}
\end{bmatrix}.$$

4.4.2 SHA-1

SHA-1 [472] has been a NIST hash function standard (FIPS 180-1, recently updated to FIPS 180-2) since 1995, and is also included in the ISO/IEC standard 10118-3. Although the output length of 160 bits is too short for a collision-resistant hash function as requested by the NESSIE call for primitives, many current applications (such as digital signature schemes) use a 160-bit hash function. Therefore SHA-1 is studied as a *legacy* hash function. The algorithm operates on 512-bit message blocks divided in words of 32 bits, and produces a message digest (hash value) of 160 bits.

4.4.2.1 The design

SHA-1 is a dedicated hash function, clearly influenced by the design of MD4 but a strengthened version of this algorithm. It is defined as the iteration of a compression function which we specify below. The computation starts with the initial value

$$IV = 67452301_x \quad \texttt{efcdab89}_x \quad \texttt{98badcfe}_x \quad 10325476_x \quad \texttt{c3d2e1f0}_x \ .$$

Each application of the compression function uses five words as initial values and 16 words of the message as input and it gives five words output, which are then used as initial values for the next application. The final output is the hash value. This works because there is a padding rule which adds bits to the message (including a representation of the message length) so that its length becomes a multiple of 512 bits.

Compression function of SHA-1.
Let the message block of 512 bits be denoted $M = [W_0 \,\|\, W_1 \,\|\, \ldots \,\|\, W_{15}]$, where W_i are 32-bit words. SHA-1 uses an expansion procedure which is defined as

$$W_i = (W_{i-3} \oplus W_{i-8} \oplus W_{i-14} \oplus W_{i-16})^{<<1} \ , \ 16 \le i \le 79 \ .$$

Define the following constants

$$\begin{aligned} K_i &= 5a827999_x \ , \ 0 \le i \le 19 \ , \\ K_i &= 6ed9eba1_x \ , \ 20 \le i \le 39 \ , \\ K_i &= 8f1bbcdc_x \ , \ 40 \le i \le 59 \ , \\ K_i &= ca62c1d6_x \ , \ 60 \le i \le 79 \ , \end{aligned}$$

and the boolean functions

$$\begin{aligned} f_i(X,Y,Z) &= f_{if}(X,Y,Z) &&= (X \,\&\, Y) \,|\, (\bar{X} \,\&\, Z) \ , \ 0 \le i \le 19 \ , \\ f_i(X,Y,Z) &= f_{xor}(X,Y,Z) &&= X \oplus Y \oplus Z \ , \ 20 \le i \le 39 \ , \ 60 \le i \le 79 \ , \\ f_i(X,Y,Z) &= f_{maj}(X,Y,Z) &&= (X \,\&\, Y) \,|\, (X \,\&\, Z) \,|\, (Y \,\&\, Z) \ , \ 40 \le i \le 59 \ . \end{aligned}$$

Suppose now that the initial values A_0, B_0, C_0, D_0, E_0 are given. Then the compression function proceeds by the following steps for $0 \le i \le 79$ (additions are mod 2^{32}):

$$\begin{aligned} A_{i+1} &= A_i^{<<5} + f_i(B_i, C_i, D_i) + E_i + W_i + K_i \ , \\ B_{i+1} &= A_i \ , \\ C_{i+1} &= B_i^{<<30} \ , \\ D_{i+1} &= C_i \ , \\ E_{i+1} &= D_i \ . \end{aligned}$$

Finally compute the output of the compression function as

$$A = A_0 + A_{80}\,, \ \ B = B_0 + B_{80}\,, \ \ C = C_0 + C_{80}\,, \ \ D = D_0 + D_{80}\,, \ \ E = E_0 + E_{80}\,.$$

4.4.2.2 Security analysis

SHA-1 is conjectured to be a collision-resistant hash function. For an output length of 160 bits this means that the expected workload of generating a collision is of the order 2^{80}. The workload of finding a (second) preimage should be of the order 2^{160}. There have not been reported any attacks on SHA-1 falsifying these statements. An important effect of the expansion of the 16-word message block to 80 words in the compression function, is that for any two distinct 16-word blocks, the resulting 80-word values differ in a large number of bit positions. This makes the attack strategy that has been used on other algorithms of the MDx-family very difficult and certainly increases the strength of the algorithm. In the following we discuss some observations that have been made and an attack on a previous version of SHA.

Collisions for SHA-0.
An earlier version of SHA — commonly known as SHA-0 — has been cryptanalysed by Chabaud and Joux [138], resulting in a theoretical attack finding collisions (two messages hashing to the same value) with a complexity of about 2^{61} evaluations of the compression function. The only difference between these two versions of SHA is in the expansion procedure which is defined for SHA-0 by

$$W_i = W_{i-3} \oplus W_{i-8} \oplus W_{i-14} \oplus W_{i-16} \ , \ \ 16 \leq i \leq 79$$

(in contrast to SHA-1 there is no rotation by one bit position to the left).

The general idea of the attack on SHA-0 is to track the propagation of local perturbations and to look for differential masks that can be added to the input block with non-trivial probability of keeping the output of the compression function unchanged. In [138] Chabaud and Joux have first studied a simplified variant of SHA-0 called SHI1 which replaces the modular additions (+) and the nonlinear functions (f_i) by bitwise addition ($\oplus$). Single bit errors or perturbations are introduced to the input of SHI1 and the perturbations are traced through the compression function. These perturbations are made to disappear by introducing five other bit errors or corrections. This allows an attack on SHI1 via differential masking. A second variant of SHA-0 called SHI2 is then analysed, where SHI2 replaces modular addition by $\oplus$ but this time keeps the nonlinear functions f_i. However we can still view f_i as acting like $\oplus$ with some probability, and the probability of a successful perturbation attack can be computed as 2^{-24}. A third variant of SHA-0, SHI3, is then analysed, where SHI3 uses modular addition as in SHA-0, but uses $\oplus$ instead of the nonlinear f_i. In this case the additions mod 2^{32} cause the perturbations to spread out due to carry propagation. However one is still able to devise a perturbation attack on SHI3 with probability 2^{-44}. Finally SHA-0 itself is analysed by taking into account the analyses of SHI2 and SHI3, and this leads to a perturbation based attack on SHA-0 requiring 2^{61} evaluations of the compression function.

It should be emphasised that, although SHA-1 and SHA-0 are similar, this attack does not carry over to SHA-1. The rotation by one bit position to the left which is added in the expansion procedure of SHA-1 means that the linear code of the expansion no longer operates on bit level: a modification of a single bit influences bits at other positions in the words as well. This makes the attack strategy of [138] completely ineffective and provides strong evidence that the transition from SHA-0 to SHA-1 raised the level of security.

Slid pairs in SHA-1.
Recently Saarinen [553] has noted that a slide attack can be mounted on SHA-1 with about 2^{32} effort. Although it is difficult to see how this attack could be exploited to find collisions or preimages for the hash function, the analysis demonstrates an unexpected property of the compression function. Consider two messages $M = [W_0 \,\|\, W_1 \,\|\, \ldots \,\|\, W_{15}]$ and $M' = [W'_0 \,\|\, W'_1 \,\|\, \ldots \,\|\, W'_{15}]$. The main observation is that the procedure for message expansion can be slid. We simply choose $W'_i = W_{i+1}$ for $0 \leq i \leq 14$ and $W'_{15} = (W_0 \oplus W_2 \oplus W_8 \oplus W_{13})^{<<1}$. After message expansion the following is true: $W'_i = W_{i+1}$ for $0 \leq i \leq 78$. The second observation is that for 20 steps in each round of the compression function, the boolean function f_i and the additive constant K_i are unchanged. Hence any two consecutive steps (step i and step $i+1$) of the compression function are similar, except for three transitions between different rounds (this happens for $i = 19, 39, 59$).

Suppose now that the hashing computation for M and M' starts from initial values A, B, C, D, E and A', B', C', D', E' respectively, which are related as follows:

$$B' = A \ , \ C' = B^{>>30} \ , \ D' = C \ , \ E' = D \ .$$

Then the purpose of the attack is to find messages and initial values for which the same relation (between the chaining variables) still holds at the end of the compression function. Such a pair of messages and corresponding initial values is called a “slid pair”. The method for finding a slid pair is rather technical but the general strategy is to choose suitable values for the chaining variables in steps $i = 19$ and $i = 39$, and perform a meet-in-the-middle match. This procedure is repeated until the transition for $i = 59$ is also dealt with, which happens with probability 2^{-32}. The overall time and space complexity of the attack are of the order 2^{32}.

Differential analysis.
Differential cryptanalysis has been applied to the encryption mode of SHA-1 (see below) and is relevant for the hash function as well. Although the propagation of a difference in the initial value through the compression function does not immediately help in finding preimages or collisions for the hash function, it may point to unwanted properties of the compression function.

For SHA-1, there exists a 5-step differential characteristic over any 5 steps in the compression function with probability 1. Handschuh and Naccache [281] conjecture that over 80 steps the best differential characteristic has probability around 2^{-116}. It is emphasised that this estimation is over-favourable to the cryptanalyst as it would be impossible to connect up all the constituent characteristics so as to achieve these biases. Van den Bogaert and Rijmen [606] have searched for optimal differential characteristics under the requirement that the

Hamming weight of every 32-bit word of the input is upper bounded by 2. It was found that there are two 10-step characteristics for f_{if} with probability 2^{-12} (this is a factor of 2 better than [281]), a 10-step characteristic for f_{xor} with best case probability 2^{-12}, and a 20-step characteristic for f_{if} and f_{maj} with probabilities 2^{-42} and 2^{-41} respectively (these figures agree with [281]).

Kim *et al.* [347] have found a 28-step differential characteristic with probability 2^{-107}, and a 30-step differential characteristic with probability 2^{-138}. This shows that when the number of steps increases, the probability of a differential characteristic decreases rapidly (this is due to the fact that the Hamming weight in the difference words grows larger). Overall the security margin of SHA-1 against this type of analysis appears very large.

Encryption mode of SHA-1.
The SHA-1 compression function can also be used in encryption mode, by inserting a key as message (so the expansion procedure is used as key schedule) and a plaintext as initial value, while leaving out the feed-forward operation at the end of the compression function. The resulting 160-bit block cipher, called SHACAL-1, has been submitted to NESSIE (see Chapter 2). One may also view SHA-1 as a Davies-Meyer hashing scheme based on SHACAL-1.

In the submission document Handschuh and Naccache [281] conjecture that a linear cryptanalytic attack on SHACAL-1 would require at least 2^{80} known plaintexts and that a differential attack would require at least 2^{116} chosen plaintexts. These estimations cannot be considered a *break* of the algorithm because Handschuh and Naccache construct very loose lower bounds. Differential cryptanalysis including boomerang attacks [347] and rectangle attacks [76] have been applied to SHACAL-1. The best known attack works for 49 steps of the compression function with a data complexity of $2^{151.9}$ chosen plaintexts and a time complexity of $2^{508.5}$ [76].

Saarinen [553] recently showed that the slide attack on SHA-1 also points to a weakness in the key schedule of SHACAL-1, and this can be exploited in a related-key attack. Given access to two SHACAL-1 encryption oracles whose keys are “slid” (in the same way that the message expansion can be slid for the hash function) the cipher can be distinguished from a randomly chosen 160-bit permutation. This requires about 2^{128} chosen plaintexts. When certain properties hold for the (related) keys, the complexity can be further reduced to about 2^{96} chosen plaintexts.

4.4.3 SHA-256, SHA-384 and SHA-512

These three hash functions have recently been included in the new NIST hash function standard (FIPS 180-2 [472]), and generate hash values of 256, 384 or 512 bits. The main reason for the new standard is to provide a security level comparable to the security level of the new NIST block cipher standard AES (with keylength of 128, 192 or 256 bits respectively). Therefore, they will serve as a benchmark for the submissions in this category.

4.4.3.1 SHA-256

The SHA-256 hash function operates on blocks of 512 bits divided in words of 32 bits and produces a message digest (hash value) of 256 bits. The algorithm is defined as the iteration of a compression function which we specify below. The computation starts with the initial value

$$\begin{array}{rcl} IV & = & \texttt{6a09e667}_x \ \texttt{bb67ae85}_x \ \texttt{3c6ef372}_x \ \texttt{a54ff53a}_x \\ & & \texttt{510e527f}_x \ \texttt{9b05688c}_x \ \texttt{1f83d9ab}_x \ \texttt{5be0cd19}_x \ . \end{array}$$

Each application of the compression function uses eight words as initial values and 16 words of the message as input and it gives eight words output, which are then used as initial values for the next application. The final output is the hash value. This works because there is a padding rule which adds bits to the message (including a representation of the message length) so that its length becomes a multiple of 512 bits.

Compression function of SHA-256.
Let the message block of 512 bits be denoted $M = [W_0 \,\|\, W_1 \,\|\, \ldots \,\|\, W_{15}]$, where W_i are 32-bit words. SHA-256 uses an expansion procedure defined by

$$W_i = \sigma_1(W_{i-2}) + W_{i-7} + \sigma_0(W_{i-15}) + W_{i-16} \ , \ \ 16 \le i \le 63 \ ,$$

with σ_0 and σ_1 defined as follows (where $(\cdot)^{>}$ denotes shift and $(\cdot)^{>>}$ rotation or circular shift to the right):

$$\begin{array}{rcl} \sigma_0(X) & = & X^{>>7} \oplus X^{>>18} \oplus X^{>3} \ , \\ \sigma_1(X) & = & X^{>>17} \oplus X^{>>19} \oplus X^{>10} \ . \end{array}$$

Define the following functions:

$$\begin{array}{rcl} Ch(X,Y,Z) & = & (X \,\&\, Y) \oplus (\bar{X} \,\&\, Z) \ , \\ Maj(X,Y,Z) & = & (X \,\&\, Y) \oplus (X \,\&\, Z) \oplus (Y \,\&\, Z) \ , \\ \Sigma_0(X) & = & X^{>>2} \oplus X^{>>13} \oplus X^{>>22} \ , \\ \Sigma_1(X) & = & X^{>>6} \oplus X^{>>11} \oplus X^{>>25} \ . \end{array}$$

Suppose now that the initial values $A_0, B_0, C_0, D_0, E_0, F_0, G_0, H_0$ are given. Then the compression function proceeds by the following steps for $0 \le i \le 63$ (additions are mod 2^{32}):

$$\begin{array}{rcl} A_{i+1} & = & \Sigma_0(A_i) + Maj(A_i, B_i, C_i) + \Sigma_1(E_i) + Ch(E_i, F_i, G_i) + H_i + W_i + K_i \ , \\ B_{i+1} & = & A_i \ , \\ C_{i+1} & = & B_i \ , \\ D_{i+1} & = & C_i \ , \\ E_{i+1} & = & D_i + \Sigma_1(E_i) + Ch(E_i, F_i, G_i) + H_i + W_i + K_i \ , \\ F_{i+1} & = & E_i \ , \\ G_{i+1} & = & F_i \ , \\ H_{i+1} & = & G_i \ . \end{array}$$

The 32-bit constants K_i are different in each of the 64 steps. We refer to the description of SHA-256 [472] for their exact value. Finally compute the output of the compression function as

$$A = A_0 + A_{64} \ , \ B = B_0 + B_{64} \ , \ C = C_0 + C_{64} \ , \ D = D_0 + D_{64} \ ,$$
$$E = E_0 + E_{64} \ , \ F = F_0 + F_{64} \ , \ G = G_0 + G_{64} \ , \ H = H_0 + H_{64} \ .$$

4.4.3.2 SHA-512

The main difference between SHA-256 and SHA-512 is that the latter uses a wordlength of 64 bits (instead of 32 bits). This allows the computation of a message digest which is twice as long compared to SHA-256, without changing the structure of the algorithm. Another distinction is that the number of steps in the compression function has been changed from 64 to 80. Hence, the SHA-512 hash function operates on blocks of 1024 bits divided in words of 64 bits and produces a message digest (hash value) of 512 bits. The algorithm is defined as the iteration of a compression function which we specify below. The computation starts with the initial value

$$\begin{array}{rcl} IV & = & \texttt{6a09e667f3bcc908}_x \ \texttt{bb67ae8584caa73b}_x \ \texttt{3c6ef372fe94f82b}_x \ \texttt{a54ff53a5f1d36f1}_x \\ & & \texttt{510e527fade682d1}_x \ \texttt{9b05688c2b3e6c1f}_x \ \texttt{1f83d9abfb41bd6b}_x \ \texttt{5be0cd19137e2179}_x \ . \end{array}$$

Each application of the compression function uses eight words as initial values and 16 words of the message as input and it gives eight words output, which are then used as initial values for the next application. The final output is the hash value. This works because there is a padding rule which adds bits to the message (including a representation of the message length) so that its length becomes a multiple of 1024 bits.

Compression function of SHA-512.
Let the message block of 1024 bits be denoted $M = [W_0 \,\|\, W_1 \,\|\, \ldots \,\|\, W_{15}]$, where W_i are 64-bit words. SHA-512 uses an expansion procedure defined by

$$W_i = \sigma_1(W_{i-2}) + W_{i-7} + \sigma_0(W_{i-15}) + W_{i-16} \ , \ 16 \le i \le 79 \ ,$$

with σ_0 and σ_1 defined as follows (where $(\cdot)^{>}$ denotes shift and $(\cdot)^{>>}$ rotation or circular shift to the right):

$$\begin{array}{rcl} \sigma_0(X) & = & X^{>>1} \oplus X^{>>8} \oplus X^{>7} \ , \\ \sigma_1(X) & = & X^{>>19} \oplus X^{>>61} \oplus X^{>6} \ . \end{array}$$

Define the following functions:

$$\begin{array}{rcl} Ch(X,Y,Z) & = & (X \,\&\, Y) \oplus (\bar{X} \,\&\, Z) \ , \\ Maj(X,Y,Z) & = & (X \,\&\, Y) \oplus (X \,\&\, Z) \oplus (Y \,\&\, Z) \ , \\ \Sigma_0(X) & = & X^{>>28} \oplus X^{>>34} \oplus X^{>>39} \ , \\ \Sigma_1(X) & = & X^{>>14} \oplus X^{>>18} \oplus X^{>>41} \ . \end{array}$$

Suppose now that the initial values $A_0, B_0, C_0, D_0, E_0, F_0, G_0, H_0$ are given. Then the compression function proceeds by the following steps for $0 \le i \le 79$ (additions are mod 2^{64}):

$$\begin{array}{rcl}
A_{i+1} & = & \Sigma_0(A_i) + Maj(A_i, B_i, C_i) + \Sigma_1(E_i) + Ch(E_i, F_i, G_i) + H_i + W_i + K_i \ , \\
B_{i+1} & = & A_i \ , \\
C_{i+1} & = & B_i \ , \\
D_{i+1} & = & C_i \ , \\
E_{i+1} & = & D_i + \Sigma_1(E_i) + Ch(E_i, F_i, G_i) + H_i + W_i + K_i \ , \\
F_{i+1} & = & E_i \ , \\
G_{i+1} & = & F_i \ , \\
H_{i+1} & = & G_i \ .
\end{array}$$

The 64-bit constants K_i are different in each of the 80 steps. We refer to the description of SHA-512 [472] for their exact value. Finally compute the output of the compression function as

$$\begin{array}{c}
A = A_0 + A_{80} \ , \ B = B_0 + B_{80} \ , \ C = C_0 + C_{80} \ , \ D = D_0 + D_{80} \ , \\
E = E_0 + E_{80} \ , \ F = F_0 + F_{80} \ , \ G = G_0 + G_{80} \ , \ H = H_0 + H_{80} \ .
\end{array}$$

4.4.3.3 SHA-384

The SHA-384 hash function is defined in the exact same manner as SHA-512 with the following two exceptions:

The computation starts with a different initial value:

$$\begin{array}{rcl}
IV & = & \texttt{cbbb9d5dc1059ed8}_x \ \texttt{629a292a367cd507}_x \ \texttt{9159015a3070dd17}_x \ \texttt{152fecd8f70e5939}_x \\
 & & \texttt{67332667ffc00b31}_x \ \texttt{8eb44a8768581511}_x \ \texttt{db0c2e0d64f98fa7}_x \ \texttt{47b5481dbefa4fa4}_x \ .
\end{array}$$

The 384-bit message digest is obtained by truncating the final hash value to its left-most 384 bits.

4.4.3.4 Security analysis

SHA-256, SHA-384 and SHA-512 are conjectured to be collision-resistant hash functions. This means that the expected workload of generating a collision is of the order $2^{n/2}$ and that the workload of finding a (second) preimage should be of the order 2^n, where n denotes the output length which, for these hash function, is equal to 256, 384 or 512 bits respectively. There have not been reported any attacks falsifying this statement.

These new hash standards are a new design that has some similarities to SHA-1, but there are important differences in the structure. We may note that for the 256-bit version the number of steps in the compression function is lower than for SHA-1 (64 steps compared to 80). On the other hand, two variables are updated in every step of the compression function where for SHA-1 only one variable is updated in a step. With respect to differential cryptanalysis, there exists a 4-step characteristic over any 4 steps in the compression function with probability 1. The probability of differential characteristics appears to decrease faster than for SHA-1. This is due to the multiple rotations in the functions $\Sigma_0(\cdot)$ and $\Sigma_1(\cdot)$. The slide attack on SHA-1 does not extend to SHA-256, SHA-384 or SHA-512 because every step of the compression function uses a unique additive constant. They are recently designed primitives of which the design strategy was not made public, so more time is needed to perform a careful and thorough security evaluation.

Encryption mode of SHA-256 and SHA-512.
The SHA-256 and SHA-512 compression functions can also be used in encryption mode, by inserting a key as message (so the expansion procedure is used as key schedule) and a plaintext as initial value, while leaving out the feed-forward operation at the end of the compression function. For SHA-256 this results in a 256-bit block cipher, called SHACAL-2, which has been submitted to NESSIE (see Chapter 2). One may also view SHA-256 as a Davies-Meyer hashing scheme based on SHACAL-2. No security flaws have been identified for SHACAL-2. The weakness in the key schedule of SHACAL-1 does not extend to SHACAL-2.

4.5 Conclusion

The NESSIE project has studied the submitted algorithm Whirlpool and has also studied the NIST hash function standards SHA-1 and SHA-256, SHA-384 and SHA-512. No significant security weaknesses have been found for any of these hash functions. The best result on Whirlpool is an attack which finds non-random properties in versions of Whirlpool where the compression function is reduced to six rounds or less (the complete function uses ten rounds). For SHA-1 a slide attack has been found that demonstrates an unexpected property of the compression function, but this is not a threat for any normal use of the hash function. However, this attack also points to a weakness in the key schedule of the encryption mode of SHA-1. SHA-256, SHA-384 and SHA-512 are a new design with significant differences from SHA-1; no weaknesses have been reported for it.

5. Message authentication codes

5.1 Introduction

Message Authentication Codes, also known as MACs, are cryptographic primitives used for information authentication. A MAC is a function that takes an input of arbitrary length and produces an output of a fixed length. Contrary to hash functions, the computation of a MAC depends on a secret key. In practical applications this key has to be shared between two parties (a sender and a receiver) so MACs are used in a symmetric setting, contrary to digital signatures (see Chapter 7) which are used for authentication in asymmetric settings. We informally give the conditions we require of a message authentication code:

- **A MAC** with an unknown key should be "hard" to forge on a new message, even when many messages and corresponding MAC values are known.

Before presenting a detailed analysis of the message authentication codes studied by the NESSIE project, we first discuss the security requirements and give an overview of common designs and current standards. For a more comprehensive overview of cryptographic primitives for information authentication (including message authentication codes) we refer to the treatment by Preneel in [530].

5.2 Security requirements

In this section we give practical and formal definitions for message authentication codes and describe the general model of an iterated MAC. We then discuss different types of attacks on message authentication codes and describe the assessment process followed by the project.

5.2.1 Security model

The following informal definition for message authentication codes was given by Preneel in [530]. A MAC is a function h satisfying the following conditions:

1. The argument X can be of arbitrary length and the result $h(K, X)$ has a fixed length of n bits, where the secondary input K denotes the secret key.

[0] Coordinator for this chapter: KUL — Bart Van Rompay

2. Given a message X (but with unknown K), it must be 'hard' to determine $h(K, X)$. Even when a large set of pairs $\{X_i, h(K, X_i)\}$ is known, it is 'hard' to determine the key K or to compute $h(K, X')$ for any new message $X' \neq X_i$.

Most MACs are iterated constructions, in the sense that they are based on a compression function with fixed size inputs; they process every message block in a similar way. The input X is padded by an unambiguous padding rule to a multiple of the block size. Typically this also includes adding the total length in bits of the input. The padded input is then divided into t blocks denoted X_1 through X_t. The MAC involves a *compression function* f and a *chaining variable* H_i between stage $i-1$ and stage i:

$$\begin{aligned} H_0 &= IV_K \ , \\ H_i &= f_K(H_{i-1}, X_i) \ , \ 1 \le i \le t \ , \\ h(K, X) &= g_K(H_t) \ . \end{aligned}$$

Here IV denotes the Initial Value and g the *output transformation*. The secret key K may be employed in the IV, in the compression function, and/or in the output transformation.

5.2.1.1 Formal definitions

Before discussing security aspects we first give more precise definitions of a MAC and its cryptographic strength. These definitions are similar to the definitions given by Krovetz in [378]. For an iterated construction similar definitions can be given as for iterated hash functions in Sect. 4.2.1.1 (taking into account the MAC key, and the fact that there usually is an additional output transformation). It is assumed that the adversary already knows the MAC values corresponding to a certain set of messages, and that his goal is to forge a MAC value for a new message, that is, a message not included in this set.

Definition 5.1. *A* **MAC** *is a function $h : \mathcal{K} \times \mathcal{M} \to \mathcal{R}$ where the key space $\mathcal{K} = \{0,1\}^k$, the message space $\mathcal{M} = \{0,1\}^*$ and the range $\mathcal{R} = \{0,1\}^n$ for some $k, n \ge 1$. When given a key $K \in \mathcal{K}$ and a message $X \in \mathcal{M}$, the function produces a MAC value $Y \in \mathcal{R}$.*

Definition 5.2 (Unforgeability). *An adversary has forged a message for a MAC h if, without knowledge of a random key K, he is able to produce a new message X and MAC value Y such that $h(K, X) = Y$. A MAC $h : \mathcal{K} \times \mathcal{M} \to \mathcal{R}$ is (t, ϵ, q)-secure if, under a randomly chosen key K, an adversary cannot forge a new message in time t with probability better than ϵ even if he is provided with the MAC values of q other messages of his choice.*

5.2.2 Classification of attacks

Depending on the information available to an adversary, the following types of attacks are distinguished for MACs:

- *Known text attack.* An attacker is able to examine some plaintexts and their corresponding MAC values.

- *Chosen text attack.* An attacker is able to select a set of texts and subsequently obtains a list of MAC values corresponding to these texts.
- *Adaptive chosen text attack.* This is a chosen text attack but where an attacker can make the choice of a text depend on the outcome of previous queries.

"Breaking" a MAC algorithm can have different meanings:

- *Existential forgery.* An attacker can determine the MAC value for at least one text. As he has no control over this text, it may be random or nonsensical.
- *Selective forgery.* An attacker can determine the MAC value for a particular text chosen a priori by him. Note that practical attacks often require that a forgery is *verifiable*, which means that the attacker knows that the forged MAC is correct with probability close to 1.
- *Key recovery.* This means that an attacker can determine the secret key K. Such a break is more powerful than a forgery, since it allows for arbitrary selective forgeries.

In practice, an adaptive chosen text attack may not always be feasible; moreover, forgeries typically need to have specific redundancy to be of any practical use. However, it is in general better to be conservative and to require that MAC algorithms resist against the strongest attacks possible. We now briefly describe the best known attacks on message authentication codes (see [530]).

Guessing of the MAC

A straightforward attack on a MAC algorithm consists of choosing an arbitrary new message, and subsequently guessing the MAC value. This can be done in two ways: either one guesses the MAC value directly, with a success probability of 2^{-n}, or one guesses the key, and then computes the MAC value, with success probability 2^{-k}. Here n denotes the size in bits of the MAC value and k the size in bits of the secret key. This is a non-verifiable attack: an attacker does not know a priori whether his guess was correct. The feasibility of the attack depends on the number of trials that can be performed and on the expected value of a successful attack; both are strongly application dependent.

Exhaustive Key Search

This is another straightforward attack that can be applied to any algorithm. The attack requires approximately k/n known text-MAC pairs for a given key; one attempts to determine the key by trying one-by-one all the keys. The expected number of trials is equal to 2^{k-1}. In contrast to the previous attack, this attack is carried out off-line and it yields a complete break of the MAC algorithm.

Internal Collision Based Forgery

Iterated MAC constructions are vulnerable to attacks based on internal collisions. The observation behind this forgery attack is that if one can find an internal collision (that is, a collision which occurs before the output transformation g), this can be used to construct a verifiable MAC forgery based on a single chosen text. Preneel and van Oorschot [532] have shown that an internal collision for

h can be found using u known text-MAC pairs and v chosen texts, where the expected values for u and v are as follows (here l denotes the size in bits of the chaining variable): $u = \sqrt{2} \cdot 2^{l/2}$ and $v = 0$ if the output transformation g is a permutation; otherwise, v is approximately

$$2\left(2^{l-n} + \left\lfloor \frac{l-n}{n-1} \right\rfloor + 1\right) \enspace .$$

Further optimisations of this attack are possible if the set of text-MAC pairs has a common sequence of s trailing blocks.

Internal Collision Based Key Recovery

For some compression functions, one can extend the internal collision attack to a key recovery attack [533]. The idea is to identify one or more internal collisions; for example, if f is not a permutation for fixed H_i, an internal collision after the first message block gives the equation $f_K(H_0, X_1) = f_K(H_0, X'_1)$, in which K and possibly H_0 are unknown ($H_0 = IV$ may be key dependent). For some compression functions f, one can obtain information on the secret key based on such relations.

Divide-and-Conquer Attack

This attack is a special case of an internal collision based key recovery. For some compression functions that use two separate keys, it is possible to exploit internal collisions for a divide-and-conquer key recovery attack [532]. Let the keys be denoted by K_1, K_2 and assume that the IV and f depend on only K_1, and that g depends on only K_2. The general idea is that an attacker first looks for some internal collisions, and then searches exhaustively for a key K_1 that produces these collisions. Once K_1 is determined, an exhaustive search is used to find K_2. Therefore, the strength of such a MAC comes from its individual keys and not from their combined length (although the attack is less practical than a simple exhaustive key search, as it needs a large number of known text-MAC pairs).

Exor Forgery

This type of forgery only works if the value of H_i is computed as a function of $H_{i-1} \oplus X_i$, and if no output transformation is present. The easiest variant of the attack requires only a single known text-MAC pair. Assume that the input X and its padded version $\bar{X}$ consist of a single block. Assume that one knows $h(K, X)$; it follows immediately that $h(K, \bar{X} \,\|\, (X \oplus h(K, X))) = h(K, X)$. This implies that one can construct a new message with the same MAC value, which is a forgery.

Side-Channel Attacks

Side-channel attacks are a major threat for all implementations of cryptographic algorithms. Side-channel attacks on MAC algorithms are similar as for other symmetric primitives. For a detailed discussion on side-channel attacks and countermeasures that can be applied to protect an implementation we refer to Annex A.

5.2.3 Assessment process

The MAC submissions were assessed with reference to the above generic common MAC attacks and by specific attacks when appropriate. Statistical analysis was carried out for various input lengths. MACs were submitted to the dependence test and linear factors test described in Sect. 2.2.4. If the NESSIE submissions were based on compression functions, these were also submitted to the two tests. Furthermore the stream cipher tests described in Sect. 2.2.4 were also applied to the MACs both in output feedback and counter mode. None of the MACs tested exhibited any anomalous behaviour.

5.3 Overview of the common designs

There are few algorithms that are designed for the specific purpose of message authentication. In most cases a message authentication code is constructed from a block cipher or from a hash function as discussed below. A different approach is the use of families of universal hash functions. We also summarise the procedures which have been undertaken by several organisations for the standardisation of message authentication codes.

5.3.1 MACs based on block ciphers

The most common way of basing a MAC on a block cipher is by using the cipher in CBC-mode (cipher block chaining): the MAC key is used as cipher key in each step of the iteration, and the message block to be processed in the current step serves as plaintext input to the cipher, after being added bit by bit to the ciphertext output from the previous step:

$$H_1 = E_K(X_1) \ , \ \ H_i = E_K(X_i \oplus H_{i-1}) \ \ (2 \leq i \leq t) \ .$$

Here we assume that the message X (after padding) is divided into blocks $X_1, \ldots, X_t$ of lengths appropriate for the block cipher used. E_K denotes encryption with secret key K and H_t forms the output of the MAC algorithm (in this case there is no output transformation).

The basic CBC-construction is susceptible to the exor forgery attack described in Sect. 5.2.2, therefore it can only be used in applications where the messages have a fixed length. Several more secure variations on the scheme exist however. One example, known as EMAC[1], is the use of an additional encryption as output transformation, where the key for this encryption operation may be derived from the MAC key. Another example, commonly known as the retail-MAC, replaces the last encryption by a two-key triple encryption. The security of these constructions can be proven based on the assumption that the underlying block cipher is pseudo-random [518].

All of these schemes are included in ISO/IEC 9797-1 [303], a standard for MACs using an (unspecified) block cipher.

[1] EMAC stands for Encrypted-MAC. This construction is also known as DMAC (Double-MAC).

5.3.2 MACs based on hash functions

A message authentication code can also be constructed from a hash function. This is a common approach because these MACs are usually faster than MACs which are based on a block cipher. HMAC is a nested construction that computes a MAC, for an underlying hash function h, message X and secret key K, as follows:

$$HMAC(K, X) = h((K \oplus opad) \,\|\, h((K \oplus ipad) \,\|\, X)) \enspace .$$

The key K is first padded with zero bits to a full block, and *opad* and *ipad* are constant values. Bellare *et al.* [44] have proven the security of this construction under the following assumptions: the underlying hash function is collision-resistant for a secret initial value; the compression function keyed by the initial value is a secure MAC algorithm (for messages of one block); the compression function is a weak pseudo-random function.

An alternative to HMAC are the MDx-MAC constructions [532] which can be based on MD5, SHA, RIPEMD or similar hash functions. Here, the underlying hash function is converted into a MAC by small modifications, involving the secret key at the beginning, at the end and in every iteration of the hash function. This is achieved by key-dependent modification of the initial value and the additive constants used by the hash function, and by a key-dependent output transformation. The security of MDx-MAC can be proven based on the assumption that the underlying compression function is pseudo-random.

Both HMAC and MDx-MAC are included in ISO/IEC 9797-2 [304], a standard for MACs using a dedicated hash function. The NESSIE submission TTMAC (see Sect. 5.4.1) is also based on a hash function, more specifically the RIPEMD-160 hash function.

5.3.3 MACs based on universal hashing

A family of hash functions $H = \{h : \mathcal{D} \to \mathcal{R}\}$ is a finite set of functions with common domain $\mathcal{D}$ and (finite) range $\mathcal{R}$. We may also denote this by $H : \mathcal{K} \times \mathcal{D} \to \mathcal{R}$ where $H_K : \mathcal{D} \to \mathcal{R}$ is a function in the family for each $K \in \mathcal{K}$. In the latter case, one chooses a random function h from the family by choosing $K \in \mathcal{K}$ uniformly and letting $h = H_K$.

A universal hash function family is a family of hash functions with some combinatoric property. For example, a hash function family $H = \{h : \mathcal{D} \to \mathcal{R}\}$ is "ϵ-almost-universal" if for any distinct $X, X' \in \mathcal{D}$, the probability that $h(X) = h(X')$ is no more than ϵ, when $h \in H$ is chosen at random.

This can be used for message authentication, for example by hashing a message with a function drawn from a universal hash function family, encrypting the output of the hash function, and then producing the encrypted hash output as MAC value. It can be proven that the security of the resulting MAC scheme depends on the security of the cipher used for encrypting the hash output. The combinatoric property of the universal hash function family is often not difficult to prove, and the resulting MAC schemes are the fastest MACs around. The NESSIE submission UMAC (see Sect. 5.4.2) is an example of a MAC based on universal hashing.

5.3.4 Current standards

Several organisations have taken initiatives for the standardisation of message authentication codes. ISO/IEC has developed standard 9797 for MACs, with two separate parts. Part 9797-1 [303] describes MACs based on a block cipher, more specifically the CBC-MAC for an unspecified block cipher (with some optional extensions including EMAC and retail-MAC). Part 9797-2 [304] details MACs based on a dedicated hash function, more specifically the HMAC and MDx-MAC constructions for an unspecified hash function (and a variant of MDx-MAC for short input strings only).

ANSI has adopted the DES-based CBC-MAC (including retail-MAC) in its banking standard X9.19 [18] and HMAC (with unspecified hash function) in standard X9.71 [22]. NIST has developed FIPS 113 [462] for DES-based CBC-MAC and FIPS 198 [473] for SHA-1-based HMAC.

5.4 MAC primitives considered during Phase II

The TTMAC and UMAC algorithms were submitted to NESSIE and selected for study during phase II of the NESSIE project. Furthermore, the schemes EMAC and HMAC, based on AES and SHA-1 respectively, and RMAC (a variant of EMAC) were selected for study during NESSIE phase II.

5.4.1 Two-Track-MAC

Two-Track-MAC (also known as TTMAC) is a message authentication code designed by Bert den Boer and Bart Van Rompay [609] and submitted to the NESSIE project. The design is based on the RIPEMD-160 hash function with modifications. The algorithm operates on blocks of 512 bits divided into words of 32 bits, uses a secret key of 160 bits, and generates an output of up to 160 bits.

5.4.1.1 The design

The design of Two-Track-MAC is based on the hash function RIPEMD-160. First, the message to be authenticated is padded with a 1-bit, and then 0-bits until its length is 448 mod 512. Then the binary representation of the length of the original message (mod 2^{64}) is appended, so the length of the message becomes a multiple of 512. Each 512-bit block is split into a set of sixteen 32-bit words, $W_0, \ldots, W_{15}$. The secret key is a set of five 32-bit words, $K_0, \ldots, K_4$. The algorithm works by iterating a compression function as follows.

Two sets of five 32-bit words $X_0, \ldots, X_4$ and $Y_0, \ldots, Y_4$ and a set of 16 message words are input to two different functions f_L and f_R that output five 32-bit words each:

$$\begin{aligned} A_0, \ldots, A_4 &= f_L(X_0, \ldots, X_4, W_0, \ldots, W_{15}) \ , \\ B_0, \ldots, B_4 &= f_R(Y_0, \ldots, Y_4, W_0, \ldots, W_{15}) \ . \end{aligned}$$

These two functions are identical to the ones used in RIPEMD-160 (the details are given below). Then compute two new sets of five words each by subtracting the input words from the results of the previous step:

$$\begin{array}{rcl} C_i & = & A_i - X_i \bmod 2^{32} \ , \ 0 \leq i \leq 4 \ , \\ D_i & = & B_i - Y_i \bmod 2^{32} \ , \ 0 \leq i \leq 4 \ . \end{array}$$

To finish the compression function, the ten words C_i and D_i are mixed in two linear transformations g_L and g_R:

$$\begin{array}{rcl} E_0, \ldots, E_4 & = & g_L(C_0, \ldots, C_4, D_0, \ldots, D_4) \ , \\ F_0, \ldots, F_4 & = & g_R(C_0, \ldots, C_4, D_0, \ldots, D_4) \ . \end{array}$$

E_i and F_i together with the next set of message words, form the inputs to the next application of the compression function. In the first iteration, the five secret keywords K_i are input as both X_i and Y_i, $0 \leq i \leq 4$.

In the final iteration, where the last message words are input, f_L and f_R swap places. After the subtraction of the input words, instead of executing g_L and g_R at the end of this iteration, compute

$$E_i = C_i - D_i \bmod 2^{32} \ , \ 0 \leq i \leq 4 \ .$$

The results from these subtractions form the output of Two-Track-MAC. An optional output transformation is defined for the computation of shorter MAC values.

The functions f_L and f_R.
The functions f_L and f_R, which are known as the left and right trail of the compression function, are identical to the functions used in the compression function of RIPEMD-160. They consist of 80 sequential steps which we describe below. We first define the constants and functions that are used.

Additive constants:

$$\begin{array}{rclrcll} k_i & = & \mathrm{00000000}_x, & k'_i & = & \mathrm{50a28be6}_x, & 0 \leq i \leq 15, \\ k_i & = & \mathrm{5a827999}_x, & k'_i & = & \mathrm{5c4dd124}_x, & 16 \leq i \leq 31, \\ k_i & = & \mathrm{6ed9eba1}_x, & k'_i & = & \mathrm{6d703ef3}_x, & 32 \leq i \leq 47, \\ k_i & = & \mathrm{8f1bbcdc}_x, & k'_i & = & \mathrm{7a6d76e9}_x, & 48 \leq i \leq 63, \\ k_i & = & \mathrm{a953fd4e}_x, & k'_i & = & \mathrm{00000000}_x, & 64 \leq i \leq 79. \end{array}$$

Non-linear functions at bit level:

$$\begin{array}{rcl} f_i(x,y,z) & = & x \oplus y \oplus z \ , \ 0 \leq i \leq 15 \ , \\ f_i(x,y,z) & = & (x \,\&\, y) \,|\, (\bar{x} \,\&\, z) \ , \ 16 \leq i \leq 31 \ , \\ f_i(x,y,z) & = & (x \,|\, \bar{y}) \oplus z \ , \ 32 \leq i \leq 47 \ , \\ f_i(x,y,z) & = & (x \,\&\, z) \,|\, (y \,\&\, \bar{z}) \ , \ 48 \leq i \leq 63 \ , \\ f_i(x,y,z) & = & x \oplus (y \,|\, \bar{z}) \ , \ 64 \leq i \leq 79 \ . \end{array}$$

Selection of message word:

$$
\begin{aligned}
r[i] &= i,\ \ 0 \le i \le 15\\
r[i] &= 7, 4, 13, 1, 10, 6, 15, 3, 12, 0, 9, 5, 2, 14, 11, 8,\ \ 16 \le i \le 31\\
r[i] &= 3, 10, 14, 4, 9, 15, 8, 1, 2, 7, 0, 6, 13, 11, 5, 12,\ \ 32 \le i \le 47\\
r[i] &= 1, 9, 11, 10, 0, 8, 12, 4, 13, 3, 7, 15, 14, 5, 6, 2,\ \ 48 \le i \le 63\\
r[i] &= 4, 0, 5, 9, 7, 12, 2, 10, 14, 1, 3, 8, 11, 6, 15, 13,\ \ 64 \le i \le 79\\
r'[i] &= 5, 14, 7, 0, 9, 2, 11, 4, 13, 6, 15, 8, 1, 10, 3, 12,\ \ 0 \le i \le 15\\
r'[i] &= 6, 11, 3, 7, 0, 13, 5, 10, 14, 15, 8, 12, 4, 9, 1, 2,\ \ 16 \le i \le 31\\
r'[i] &= 15, 5, 1, 3, 7, 14, 6, 9, 11, 8, 12, 2, 10, 0, 4, 13,\ \ 32 \le i \le 47\\
r'[i] &= 8, 6, 4, 1, 3, 11, 15, 0, 5, 12, 2, 13, 9, 7, 10, 14,\ \ 48 \le i \le 63\\
r'[i] &= 12, 15, 10, 4, 1, 5, 8, 7, 6, 2, 13, 14, 0, 3, 9, 11,\ \ 64 \le i \le 79
\end{aligned}
$$

Rotation constants:

$$
\begin{aligned}
s[i] &= 11, 14, 15, 12, 5, 8, 7, 9, 11, 13, 14, 15, 6, 7, 9, 8,\ \ 0 \le i \le 15\\
s[i] &= 7, 6, 8, 13, 11, 9, 7, 15, 7, 12, 15, 9, 11, 7, 13, 12,\ \ 16 \le i \le 31\\
s[i] &= 11, 13, 6, 7, 14, 9, 13, 15, 14, 8, 13, 6, 5, 12, 7, 5,\ \ 32 \le i \le 47\\
s[i] &= 11, 12, 14, 15, 14, 15, 9, 8, 9, 14, 5, 6, 8, 6, 5, 12,\ \ 48 \le i \le 63\\
s[i] &= 9, 15, 5, 11, 6, 8, 13, 12, 5, 12, 13, 14, 11, 8, 5, 6,\ \ 64 \le i \le 79\\
s'[i] &= 8, 9, 9, 11, 13, 15, 15, 5, 7, 7, 8, 11, 14, 14, 12, 6,\ \ 0 \le i \le 15\\
s'[i] &= 9, 13, 15, 7, 12, 8, 9, 11, 7, 7, 12, 7, 6, 15, 13, 11,\ \ 16 \le i \le 31\\
s'[i] &= 9, 7, 15, 11, 8, 6, 6, 14, 12, 13, 5, 14, 13, 13, 7, 5,\ \ 32 \le i \le 47\\
s'[i] &= 15, 5, 8, 11, 14, 14, 6, 14, 6, 9, 12, 9, 12, 5, 15, 8,\ \ 48 \le i \le 63\\
s'[i] &= 8, 5, 12, 9, 12, 5, 14, 6, 8, 13, 6, 5, 15, 13, 11, 11,\ \ 64 \le i \le 79
\end{aligned}
$$

Suppose that the initial values a_0, b_0, c_0, d_0, e_0 are given.[2] The function f_L (left trail of the compression function) consists of the following steps for $0 \le i \le 79$ (additions are mod 2^{32}):

$$
\begin{aligned}
a_{i+1} &= e_i\ ,\\
b_{i+1} &= (a_i + f_i(b_i, c_i, d_i) + W_{r[i]} + k_i)^{<<s[i]} + e_i\ ,\\
c_{i+1} &= b_i\ ,\\
d_{i+1} &= c_i^{<<10}\ ,\\
e_{i+1} &= d_i\ .
\end{aligned}
$$

Similarly, when the initial values a_0, b_0, c_0, d_0, e_0 are given[3], the function f_R (right trail of the compression function) consists of the following steps for $0 \le i \le 79$ (additions are mod 2^{32}):

[2] These values are $X_0, \dots, X_4$ in the description above.

[3] These values are $Y_0, \dots, Y_4$ in the description above.

$$\begin{aligned} a_{i+1} &= e_i \ , \\ b_{i+1} &= (a_i + f_{79-i}(b_i, c_i, d_i) + W_{r'[i]} + k'_i)^{<<s'[i]} + e_i \ , \\ c_{i+1} &= b_i \ , \\ d_{i+1} &= c_i^{<<10} \ , \\ e_{i+1} &= d_i \ . \end{aligned}$$

The functions g_L and g_R.

The linear transformations g_L and g_R are used to mix the outputs of the two trails of the compression function. For inputs $C_0, \ldots, C_4$ and $D_0, \ldots, D_4$, the function g_L computes five words $E_0, \ldots, E_4$ as follows (operations are mod 2^{32}):

$$\begin{aligned} E_0 &= (C_1 + C_4) - D_3 \ , \\ E_1 &= C_2 - D_4 \ , \\ E_2 &= C_3 - D_0 \ , \\ E_3 &= C_4 - D_1 \ , \\ E_4 &= C_0 - D_2 \ . \end{aligned}$$

For inputs $C_0, \ldots, C_4$ and $D_0, \ldots, D_4$, the function g_R computes five words $F_0, \ldots, F_4$ as follows (operations are mod 2^{32}):

$$\begin{aligned} F_0 &= C_3 - D_4 \ , \\ F_1 &= (C_4 + C_2) - D_0 \ , \\ F_2 &= C_0 - D_1 \ , \\ F_3 &= C_1 - D_2 \ , \\ F_4 &= C_2 - D_3 \ . \end{aligned}$$

5.4.1.2 Security analysis

The security of Two-Track-MAC can be proven based on the assumption that the underlying compression function is pseudo-random. This function is very similar to the compression function used by RIPEMD-160 [531], which is a well studied primitive for which no weaknesses have been reported.

The functions f_L and f_R consist of 80 step iterations, and use the different bit operations AND, OR, XOR and bit complementation. The step functions also include bit rotations and addition mod 2^{32}, and iterated 80 times it seems very hard to trace unknown key bits through it.

It is worth noting that the only place where key material is used is in the initial value. The 160-bit output of the algorithm is the difference between two 160-bit quantities, so the knowledge of this difference still gives 160 bits of uncertainty about which two values produced it. In other words, guessing what these two values are, and computing backwards to find the input, i.e. the key, is no faster than guessing on the key directly.

The large size of the internal state (320 bits) in Two-Track-MAC gives the algorithm a high level of security against attacks based on internal collisions.

5.4.2 UMAC

UMAC is a message authentication code designed by Ted Krovetz, John Black, Shai Halevi, Hugo Krawczyk and Phillip Rogaway [379] and submitted to the NESSIE project. The design is based on families of universal hash functions (see Sect. 5.3.3 for a definition) and offers provable security in the sense that there are provable collision bounds for the compression, so that the security depends on the AES cipher which is used for the encipherment of a nonce and the derivation of key material. Compared to conventional MAC algorithms, UMAC offers the benefits of faster speed (especially for long messages) and provable security at the cost of greater complexity.

5.4.2.1 The design

The UMAC message authentication code evolved from an earlier version UMAC (1999). We first describe this first version, next the additions for the new version, and we also describe some practical specifications and parameter sets. Due to the complexity of the scheme we give only a general outline, for details we refer to the algorithm specification in [379].

The previous version of UMAC.
UMAC (1999) computes the MAC by first compressing the message by a fixed ratio using the NH universal hash function family. A nonce is then concatenated to the compressed message and the result processed by a PRF (pseudo-random function) to obtain the authentication tag. HMAC (based on a conventional hash function) and CBC-MAC (based on a block cipher) were proposed as PRF. In short we can say that the universal hash function family NH is used as an accelerant to HMAC or CBC-MAC.

NH works by dividing the message into blocks of a certain length (except for the last block which can be shorter). Each block is processed by adding key material with the same length to it (the same key is used for every block), and compressing it by multiplying pairs of words and adding the results (e.g., starting from a block of 1024 32-bit words one can obtain a compressed value of 64 bits, which means a compression factor of 512). All compressed blocks are then concatenated and length information is appended.

The algorithm has some parameters, like the block size and word size, the PRG (pseudo-random generator) used to compute the needed key material from the user key (NH needs a key with the same length as the blocks in which the message is divided), and the PRF used to process the compressed message and nonce. Furthermore it is possible to use the Toeplitz construction to reduce the chance of forgery (by applying NH several times with keys that are shifted versions of each other, and concatenating the results), and/or to use two-level hashing to reduce the amount of needed key material. There are some other variations that allow optimisation for certain architectures (e.g., MMX).

Most of the limitations of UMAC (1999) come from the fact that the message is compressed by a fixed ratio rather than to a fixed length. In the first case the authentication tag is computed with $\mathrm{PRF}(hash||nonce)$, in the second case it can be computed with $hash \oplus \mathrm{PRF}(nonce)$, which has some advantages (the use of the PRF is limited to the minimum, it does not have an input of unbounded length). NH, the universal hash function family used in UMAC (1999), could also compress to a fixed output length, but then it would need a key (generated by the PRG) with length equal to the message (the entire message would be treated as a single 'block' in the description given above).

The submitted version of UMAC.

The new version of UMAC (2000) introduces extra complexity to solve the problem of compressing the message to a fixed length so the MAC can be computed with $hash \oplus \mathrm{PRF}(nonce)$. The PRF part works by enciphering the nonce with a block cipher. The hash part (also called UHash), which compresses the message, consists of three different layers:

– Compression: The first layer uses the fast NH hash family to compress the message by a fixed ratio.
– Hash-to-fixed-length: The second layer uses the RP hash family, which is not as fast as NH but generates an output of fixed length using a fixed-length key.
– Strengthen-and-fold: The third layer uses the IP hash family, which reduces the length of its input to a more appropriate size.

The RP universal hash function family is polynomial-based. A string made of n words of bitlength w can be viewed as a polynomial of degree n over a finite field, where each word of the string serves as a coefficient. To compute the hash, one evaluates the polynomial for a randomly chosen point (the key). For efficiency reasons, the computations are performed in a prime field, using the largest prime less then 2^w. The function has been tweaked in order to allow expansion of the domain to arbitrary strings (allowing variations in length, and dealing with strings outside the prime field). RP stands for ramped polynomial hashing: small prime fields are used for short messages, and larger prime fields for longer messages, the reason being that computations in a small prime field are more efficient, but can only handle messages up to a certain length for a given collision probability (for polynomial hashing the collision bounds degrade linearly with the length of the message being hashed, for a given size of the key set). In fact a hybrid scheme is used, where for long messages a small prime field is used on the first part of the message. The result is a hash family with arbitrary length inputs and fixed-length outputs, using a fixed-length key. From a security point of view it adds little to the collision probability compared to the NH hash layer of the algorithm.

The layer with the IP universal hash function family reduces the length of its input because the RP hash layer generates outputs which are quite long compared to the collision probability which is offered (for all but the longest messages being authenticated many of the leading bits of the output string of the RP layer will be zeros). It is based on computing the inner-product over a prime field (multiplying input words with key words and adding the results). While doing this the collision probability from the previous layer of the algorithm is maintained.

Specifications.

Many options are available in an implementation of UMAC but two named parameter sets have been specified: **UMAC16** and **UMAC32**. They are based on the three-layer hash schemes UHash16 and UHash32 respectively, and use the AES (Rijndael) block cipher for enciphering the nonce. The PRG which computes (from the user key) the key material needed in the internal operation of UHash is also based on the AES, in output-feedback mode.

UHash16 uses 16-bit words, representing them as signed integers. The NH hash layer operates on blocks of 2Kbytes, which are compressed to 32-bit values (this corresponds to a compression ratio of 512). The collision probability is proved to be no more than 2^{-15}. The result is passed to the RP hash layer which computes an output string of a fixed length of 128 bits. The RP hash family is a ramped construction using three prime fields with a 32-bit, 64-bit and 128-bit prime modulus respectively. The message length is restricted to a maximum of 2^{64} bits, and it is proved that this layer adds only little (around 2^{-19}) to the collision probability. When the message being authenticated is short to begin with, the RP layer is not needed and it is skipped as an optimisation. The IP hash layer folds its 128-bit input into a 16-bit output, maintaining the collision probability of nearly 2^{-15}. The three-layer construction is iterated a number of times, with independent keys, to increase the length of the authentication tag and decrease the chance of MAC forgery. The default number is four times, and concatenating the 16-bit output values one obtains a 64-bit MAC with forging probability 2^{-60}.

The main difference in UHash32, compared to UHash16, is that it uses 32-bit words and iterates over the three-layer scheme only twice (default). This gives differences in the implementation (the use of larger prime fields) but the analysis is mainly the same.

The advantages over the previous UMAC (1999) version are that the use of the cryptographic primitive (AES) is minimised, that (as a result) it is more efficient on short messages, and that it offers extra flexibility for the verifier: he can choose how many of the parallel iterations he wants to perform in the MAC computation, thereby trading computation time for assurance level.

5.4.2.2 Security analysis

The UMAC message authentication code is based on families of universal hash functions and offers provable security in the sense that there are provable collision bounds for the hashing part of the MAC computation, so the security in the end depends on the cryptographic primitive used for enciphering the nonce. The primitive stated in the specification is the AES (Rijndael) block cipher, which has had a lot of analysis during and after the AES process supporting its security claims. There is the added advantage that the encryption is only performed on a short nonce.

No flaws have been found in the security proof of UMAC. For the layer using the NH universal hash function family, a first security proof states that NH is 2^{-w}-almost-universal (this means that the collision probability is no more than 2^{-w}) for strings of equal length, when it operates on words of bitlength w. This corresponds to the use of NH on one block of fixed length of the message. A second security proof allows to extend this result to NH working on any pair of strings, like in the UMAC setting. The reason that the collision probability after the NH hash layer in the UHash16 scheme is 2^{-15} rather than 2^{-16} is that a signed rather than an unsigned version of NH is used. A variant of the first security proof shows that the signed version of NH is 2^{-w+1}-almost-universal.

UHash16 and UHash32 have two additional layers using the RP and IP universal hash function families, and they iterate the three-layer scheme four, respectively two times. It is proven that the hash family UHash16 is 4-wise $(2^{-15}+2^{-18}+2^{-28})$-almost-universal, and that the hash family UHash32 is 2-wise $(2^{-31}+2^{-33})$-almost-universal[4].

5.4.3 CBC-constructions: EMAC and RMAC

The DES-based CBC-MAC [462] is an old NIST MAC standard that will probably be upgraded to AES-based CBC-MAC and the general scheme is also included in the ISO/IEC standard 9797-1. Therefore AES-based CBC-MAC is considered as a benchmark for the submissions in this category. The scheme uses the AES (Rijndael) block cipher in a black box model and generates a MAC value of up to 128 bits. We consider EMAC[5] [518], a variant of CBC-MAC with an additional encryption at the end (this is one of the extensions included in ISO/IEC 9797-1). We also discuss another recently proposed variant called RMAC [318, 471].

5.4.3.1 The design

The computation of EMAC for a secret key K and a message X — divided (after padding) in 128-bit blocks $X_1, \ldots, X_t$ — proceeds as follows (here E_K denotes encryption with the 128-bit block cipher AES using key K):

1. Compute $H_1 = E_{K1}(X_1)$.
2. For $i = 2, \ldots, t$: compute $H_i = E_{K1}(X_i \oplus H_{i-1})$.
3. Compute the output transformation: $H_{out} = E_{K2}(H_t)$. The key $K2$ may be derived from $K1$ by the following procedure: $K2 = K1 \oplus \texttt{f0f0 ...f0}_x$.
4. To obtain an m-bit MAC value, select the leftmost m bits of H_{out}.

RMAC is a randomised variant of this scheme, offering improved resistance against attacks based on internal collisions. The only difference is in the output transformation where one encrypts with a key that is obtained by bitwise addition of $K2$ and a salt R:

$$H_{out} = E_{K2 \oplus R}(H_t) \ .$$

For RMAC, the keys $K1$ and $K2$ may be independent or they can be derived from one master key in a standard way. The salt R is r bits long and should be padded with 0-bits if it is shorter than $K2$. For RMAC as specified by NIST in [471], five different parameter sets have been defined for the sizes m and r.

[4] Stronger universality properties are also proven for UHash16 and UHash32, see [379].

[5] This construction was previously known as DMAC.

5.4.3.2 Security analysis

The security of EMAC can be proven based on the assumption that the underlying block cipher is pseudo-random [518], in this case Rijndael which has received a lot of analysis during and after the AES process supporting its security claims.

It is worth noting that without the additional encryption at the end, a simple (adaptive chosen text) existential forgery would be possible for the CBC-MAC scheme (due to an exor forgery attack) Existential forgeries are also possible in the case where an additional encryption is used but where $K2$ is chosen equal to $K1$ If $K1$ and $K2$ are chosen independently, rather than setting $K2 = K1 \oplus \texttt{f0f0 ...f0}_x$ the level of security against key recovery attacks is less than suggested by the combined key size (due to a divide-and-conquer attack). An internal collision based forgery for EMAC needs about 2^{64} known text-MAC pairs and 1 chosen text when the output length is 128 bits. More chosen texts are required when the MAC output is truncated, e.g., when the leftmost 64 bits are chosen, the attack needs about 2^{64} known pairs and 2^{64} chosen texts. On the other hand this increases the probability of success for an attack where one tries to guess the MAC value.

The main advantage of the randomised variant RMAC is that it offers improved resistance against attacks that are based on internal collisions. For example, when parameters $m = 128$ and $r = 128$ are chosen, such an attack requires about 2^{128} known pairs and 1 chosen text. On the other hand RMAC needs stronger assumptions for its security proof; for instance, the underlying block cipher must be secure against related-key attacks. In Appendix A of [471] it is noted that for RMAC with two independent keys $K1$ and $K2$ an exhaustive search for the keys is expected to require the generation of 2^{2k-1} MACs, where k is the size of one key. However, as noted in [358, 363], this can be done much faster for parameters $m = 128$ and $r = 128$: under a chosen message attack with just one known message and one chosen message $K2$ can be found with about 2^k decryption operations, subsequently $K1$ can be found in roughly the same time. [358,363] also describe an alternative attack on RMAC ($m = 128$, $r = 128$) requiring 2^{123} chosen texts (in running time 2^{124}), and a serious attack on RMAC using three-key Triple-DES as underlying block cipher instead of AES (in certain cases this attack works with a complexity of about 2^{56} operations and success probability of 2^{-8}).

5.4.4 HMAC

The SHA-1 based HMAC [473] has been standardised by NIST as FIPS-198 and the general scheme is also included in the ISO/IEC standard 9797-2. Therefore it is considered as a benchmark for the submissions in this category. The scheme uses the SHA-1 hash function in a black box model and generates a MAC value of up to 160 bits. SHA-1 operates on blocks of 512 bits that are divided in 32-bit words, computing a 160-bit hash value.

5.4.4.1 The design

The computation of the MAC for a secret key K and a message X proceeds by the following steps (here h denotes hashing with the hash function SHA-1):

1. Compute a key value K' of 512 bits long. Suppose K has bitlength l. If $l = 512$ set $K' = K$; if $l < 512$ obtain K' by appending $512 - l$ zero bits to K; if $l > 512$ obtain K' by computing the hash $h(K)$ (160 bits long) and appending 352 zero bits to this hash value.
2. Exor K' with the 512-bit constant *ipad* and append the message X: $(K' \oplus \mathit{ipad}) \,\|\, X$.
3. Compute the hash of the string resulting from step 2: $h((K' \oplus \mathit{ipad}) \,\|\, X)$.
4. Exor K' with the 512-bit constant *opad* and append the 160-bit result from step 3: $(K' \oplus \mathit{opad}) \,\|\, h((K' \oplus \mathit{ipad}) \,\|\, X)$.
5. Compute the hash of the string resulting from step 4:
   $h((K' \oplus \mathit{opad}) \,\|\, h((K' \oplus \mathit{ipad}) \,\|\, X))$.
6. To obtain an m-bit MAC value, select the leftmost m bits of the result of step 5.

The string *ipad* is defined as the concatenation of 64 times the hexadecimal value ‘36’, and the string *opad* is defined as the concatenation of 64 times the hexadecimal value ‘5c’.

5.4.4.2 Security analysis

Bellare *et al.* [44] give theoretical support for HMAC, relating the security of the MAC scheme to the security of the underlying hash function, in this case SHA-1 which is a well studied primitive for which no weaknesses have been reported. More specifically, it has been proved that HMAC is secure if the following assumptions hold (here f is the compression function that is iterated by the hash function for each 512-bit block):

- The hash function h is collision-resistant when the initial value is secret.
- The compression function f keyed by the initial value is a strong MAC algorithm (this means that its output is hard to predict).
- The values $f(K' \oplus \mathit{ipad})$ and $f(K' \oplus \mathit{opad})$ cannot be distinguished from truly random values. This means that the compression function f is a ‘weak’ pseudorandom function (‘weak’ because the opponent has no direct access to K').

It may be noted that if the HMAC construction is used with two independent keys (rather than using $K_1 = K' \oplus \mathit{ipad}$ and $K_2 = K' \oplus \mathit{opad}$), the level of security against key recovery attacks would be less than suggested by the algorithms key size (due to a divide-and-conquer attack). An internal collision based forgery for SHA-1-based HMAC needs about 2^{80} known text-MAC pairs and 1 chosen text when the output length is 160 bits. More chosen texts are required when the MAC output is truncated, e.g., when the leftmost 80 bits are chosen, the attack needs about 2^{80} known pairs and 2^{80} chosen texts. On the other hand this increases the probability of success for an attack where one tries to guess the MAC value.

5.5 Comparison of studied MAC primitives

In this section we compare the security levels of the MAC algorithms studied in the NESSIE project. No short-cut attacks have been found for any of the algorithms, except RMAC. The estimated complexity for an exhaustive key search depends on the bitlength k of the secret key: about 2^{k-1} off-line MAC computations are required. Likewise, the estimated complexity for an attack guessing the MAC output depends on the bitlength n of the output: about 2^{n-1} on-line MAC verifications are needed for a (non-verifiable) forgery. Note that for UMAC the (provable) forging probability is only near optimal: 2^{-60} for an output size of 64 bits. Table 5.23 below compares the possible values of k and n for the different algorithms[6].

Table 5.23. Key length k and output length n for MAC primitives.

Algorithm	k	n
UMAC	128	64
TTMAC	160	≤ 160
EMAC-AES	128,192,256	≤ 128
RMAC-AES	128,192,256	≤ 128
HMAC-SHA-1	≤ 512	≤ 160

The most effective generic attack on MAC algorithms is the birthday forgery attack (based on internal collisions): for an internal state size of l bits and output size of n bits, this attack requires about $2^{l/2}$ known text-MAC pairs and 2^{l-n} chosen texts. Table 5.24 below compares this complexity for the different algorithms. The entries in the table are denoted (α, β), where α is the number of known pairs and β the number of chosen texts that are needed. Maximum values are chosen for the output size n of the algorithms (this minimises the number of chosen texts that are needed in the attack).

Table 5.24. Estimated (minimal) complexity of birthday forgery attacks on MACs.

Algorithm	birthday forgery
TTMAC	$(2^{160}, 2^{160})$
EMAC-AES	$(2^{64}, 1)$
RMAC-AES	$(2^{128}, 1)$
HMAC-SHA-1	$(2^{80}, 1)$

It can be seen from Table 5.24 that for EMAC-AES and HMAC-SHA-1 the birthday attack requires respectively 2^{64} or 2^{80} known pairs and (in both cases) 1 chosen text (when the output size n is equal to the internal state size). The randomised RMAC offers better resistance than EMAC: when both the output and the salt value are 128 bits long, the attack needs 2^{128} known pairs and

[6] Note that other parameter sets can be chosen for UMAC.

1 chosen text. TTMAC offers the highest level of security because of its large internal state size: 2^{160} known pairs and 2^{160} chosen texts are needed (when $n = 160$). Note that the birthday attack does not apply to the UMAC algorithm.

For RMAC-AES there is an alternative attack as described in [358, 363]. This attack can be used to find one of the two keys in the system faster than by an exhaustive search (after which RMAC reduces to a simple CBC-MAC for which it is well known that simple forgeries can be found). The estimated complexity of the attack is 2^{123} chosen texts and 2^{124} running time (considering RMAC with output and salt value of 128 bits).

6. Asymmetric encryption schemes

6.1 Introduction

Asymmetric encryption, also known as public-key encryption, is a method of sending messages securely between two people who do not share a common secret. This is in contrast to symmetric encryption, where the communicating parties are assumed to share a common secret key. Examples of symmetric encryption include block ciphers (see Sect. 2) and stream ciphers (see Sect. 3). Asymmetric cryptography was developed out of the ideas of Diffie and Hellman [201] and was first properly realised as the ever popular RSA cryptosystem [543] in 1978.

In the twenty-five years since then the area has received copious attention from researchers, who have tightened security definitions and requirements, proposed and broken new schemes, and expanded the range of applications.

During the three-year lifespan of the NESSIE project, there have been several important shifts of focus within this research area. Much effort has been expended by the research community in the field of provable security and no new primitive is really taken seriously these days unless its security can in some way be related to a "hard" problem. This has also led to an increase in research on so called side-channel attacks: attacks that take advantage of information that may be available in the real world but is not available to an attacker in a mathematical model. These security requirements and limitations will be discussed in Sect. 6.2.

Also within the last three years, the International Organisation for Standardization (ISO) has developed a new framework for asymmetric encryption that attempts to better model the real-world use of public-key cryptography. The new KEM-DEM framework, discussed in Sect. 6.3, was popular enough that almost all the primitives selected for further study in phase II of the NESSIE project were tweaked to fit into this model. Only EPOC-2, discussed in Sect. 6.4.4, remained completely outside of this framework.

6.2 Security Requirements

6.2.1 Preliminaries

We start with a formal definition of an asymmetric encryption algorithm. In order to fulfil the security requirements of Sect. 6.2.2 we will see that it is necessary

[0] Coordinator for this chapter: RHUL — Alex Dent

for encryption algorithms to be probabilistic. We choose here to represent these probabilistic algorithms as deterministic algorithms that take some fixed length random seed as input. This serves to accentuate the problem of adaptively creating multiple random bit strings, e.g. generating one bit string, testing it for some property and then discarding it and generating a new random bit string. However, when there is no danger of confusion, we assume this input is implicit and whenever an algorithm generates some randomness it actually derives the required randomness from this seed.

Definition 6.1. *An asymmetric encryption system is a triple of deterministic algorithms $(\mathcal{G}, \mathcal{E}, \mathcal{D})$. The first algorithm $\mathcal{G}$ is called the key generation algorithm and need only be run once to set up the system. It takes as input a unary string 1^λ, where λ is called the security parameter, and a fixed length random seed r, and outputs a key-pair (pk, sk). The key pk is called the public key and needs to be distributed to all people who wish to encrypt messages. The key sk is called the secret key or private key*[1] *and should only be known to those people who are permitted to read encrypted messages.*

The second algorithm $\mathcal{E}$ is the encryption algorithm. It takes as input a message m, the public key pk and a fixed length random seed r, and outputs a ciphertext C.

The last algorithm $\mathcal{D}$ is the decryption algorithm. It takes as input a ciphertext and the secret key (which may include elements from the public key), and outputs a message m' or the error symbol $\perp$.

For the system to be useful we require that it is *sound*, i.e. for any message m, random seed r and valid key-pair (pk, sk) we have that $\mathcal{D}(\mathcal{E}(m, r, pk), sk) = m$. We also require that some kind of security result holds that limits an attacker's power to recover information about a message m or the secret key sk from an encipherment.

It is impossible for an asymmetric encryption scheme to be perfectly secure; an attacker that has access to unlimited (time and computational) resources can always recover a secret key. This is because an attacker with unbounded resources can just search the (finite) space of possible private keys and check their ability to decrypt messages. So, in order to prove any meaningful results, we have to limit the attacker's computational power and there are two approaches to this problem.

The first uses the field of complexity theory. We can assume that the attacker is represented by a (probabilistic) Turing machine that runs in polynomial-time in the security parameter, and then derive results about its ability to break the scheme. This is a very elegant theory but is only useful as an asymptotic approximation. An attacker that runs in polynomial time is not guaranteed to be practical in real terms, and an attacker that doesn't run in polynomial time is not guaranteed to be impractical for all useable security parameters. Since the NESSIE call for primitives [475] required that the security level of a candidate

[1] Some standardisation bodies reserve the term "secret key" for a key used within a symmetric algorithm and insist upon the use of the term "private key" for asymmetric applications.

be of a certain specified level we must regretfully put aside this theory and concentrate on something more concrete.

Definition 6.2. *A (t, ϵ) solver for a problem is a probabilistic Turing machine that runs in time bounded above by t and outputs a solution for the problem with probability at least ϵ.*

A (t, ϵ, q_D) attacker for an asymmetric encryption scheme is a probabilistic Turing machine $\mathcal{A}$ that runs in time bounded above by t, makes at most q_D queries to a decryption oracle and succeeds in breaking the scheme with probability at least ϵ.

A further discussion of the access an attacker might have to a decryption oracle and the definition of "breaking the scheme" can be found in Sect. 6.2.2.

Of course, the success probability ϵ depends upon the units we use for the time t — if we measure time in years then we would expect a higher success probability in 1 unit time than if we measure time in seconds! We decide that one unit of time is equal to the time taken for one decryption operation.

However there is still a problem with this approach. In order to prove that a system is as secure as the NESSIE call requires, the submitters had to prove the *absence* of a strong attacker. To do this the submitters were allowed to submit a proof that the existence of a (t, ϵ, q_D) attacker for their scheme implied the existence of a (t', ϵ') solver for some trusted cryptographic problem. The relationship between t, t', ϵ and ϵ' defines the *efficiency* of the security reduction. A discussion of those problems which are trusted to be "hard" by the cryptographic community can be found in Sect. 6.2.3.

6.2.2 The Security Models

So far we have specifically avoided stating what it means for an asymmetric encryption scheme to be secure. In order to do this we have to define two things: the conditions an attacker must fulfil for the scheme to be considered broken, and the access that an attacker has to the system.

There are many different ways in which a cryptosystem might be considered weak. Whether these weaknesses actually "break" the system, i.e. give an attacker some useful information, depends upon the application for which the cryptosystem is being used. We model these various success criteria as games that an attacker plays against a mythical system that controls the encryption scheme and measure the attacker's success as the probability that he wins the game.

The most obvious, and most naive, way that an encryption scheme can be considered weak is if there exists an attacker that can, given some ciphertext, recover the associated message.

Definition 6.3 (One-way (OW)). *Consider the following game that an attacker plays against a system which is using an asymmetric encryption scheme $(\mathcal{G}, \mathcal{E}, \mathcal{D})$ with a security parameter λ.*

1. *The system picks a random seed r, runs $\mathcal{G}(1^\lambda, r)$ to generate a key-pair (pk, sk) and passes the value pk to the attacker $\mathcal{A}$.*

2. *The attacker runs until it is ready to receive a challenge ciphertext.*
3. *The system picks a random seed r and a message m uniformly at random from the set of possible messages and calculates the challenge ciphertext $C = \mathcal{E}(m, r, pk)$. The system then passes C back to the attacker.*
4. *The attacker outputs a guess m' for the message m.*

The attacker wins the above game if $m' = m$.

An asymmetric encryption scheme is said to be one-way *or* OW *if the probability that the attacker wins the above game is small.*

This is a fairly weak definition of security. For example, if an encryption scheme is only used to encrypt a message from a certain small known subset of possible messages then it might be enough for an attacker to tell whether a ciphertext is the encryption of one given message or another. This leads to the stronger definition of message-indistinguishable encryption schemes [271, 536].

Definition 6.4 (Message-indistinguishable (IND)). *Consider the following game that an attacker plays against a system which is using an asymmetric encryption scheme $(\mathcal{G}, \mathcal{E}, \mathcal{D})$ with security parameter λ.*

1. *The system picks a random seed r, runs $\mathcal{G}(1^\lambda, r)$ to generate a key-pair (pk, sk) and passes the value pk to the attacker $\mathcal{A}$.*
2. *The attacker generates two distinct messages m_0 and m_1, and submits them to the system.*
3. *The system*
   a) *Chooses a bit σ uniformly at random from $\{0, 1\}$ and a random seed r.*
   b) *Calculates the challenge ciphertext $C = \mathcal{E}(m_\sigma, r, pk)$ and returns this to the attacker.*
4. *The attacker outputs a guess σ' for σ.*

The attacker wins the above game if $\sigma' = \sigma$.

An attacker $\mathcal{A}$ has an advantage *$Adv_\mathcal{A}$ of winning the above game where*

$$Adv_\mathcal{A} = Pr[\sigma' = \sigma] - 1/2 \tag{6.1}$$

and the scheme is said to have advantage

$$Adv = \max_\mathcal{A} Adv_\mathcal{A} \ . \tag{6.2}$$

An asymmetric encryption scheme is said to be message-indistinguishable *or* IND *if the scheme's advantage is small.*

Consequently it is easy to see that if a system is message-indistinguishable then it is certainly one-way; however there are schemes which are thought to be one-way that are definitely not message-indistinguishable (such as the original RSA cryptosystem [543]). Of course it is difficult to show that any scheme is one-way as a proof that a scheme is one-way would imply a proof that $P \neq NP$, which would be a major mathematical achievement. However we can show that there exists the scope for a scheme that is one-way but not message indistinguishable.

Consider an asymmetric encryption scheme $(\mathcal{G}, \mathcal{E}', \mathcal{D}')$ constructed from a one-way asymmetric encryption scheme $(\mathcal{G}, \mathcal{E}, \mathcal{D})$ in the following manner:

$$\begin{aligned}
\mathcal{E}'(m, pk) &= b||\mathcal{E}(m, pk) \text{ where } b \text{ is the leftmost bit of } m\,, \\
\mathcal{D}'(b||C, sk) &= \begin{cases} \mathcal{D}(C, sk) & \text{provided the leftmost bit of } \mathcal{D}(C, sk) \text{ is } b\,, \\ \perp & \text{otherwise}\,. \end{cases}
\end{aligned}$$

This scheme is one-way (possibly slightly less one-way than the original scheme) and yet it is easy to distinguish between messages that have different leftmost bits. Hence the scheme is not message indistinguishable.

Next we will consider the access an attacker has to the system. In the simplest case an attacker might only have access to the challenge ciphertext and the public key. However it is possible that an attacker might be able to gain decryptions of certain messages, so we have to define more relaxed attack models.

Definition 6.5 (Attack models). *We assume that the attack algorithm $\mathcal{A}$ runs in two stages: pre-challenge and post-challenge. Let the attacker have access to an oracle $\mathcal{O}_1$ up until the challenge is issued, and access to an oracle $\mathcal{O}_2$ after this time.*

- *The attack is said to be a* chosen plaintext attack (CPA) *if the oracles are both trivial, i.e. $\mathcal{O}_1 = \mathcal{O}_2$ and both return the error symbol $\perp$ for any input.*
- *The attack is said to be a* chosen ciphertext attack (CCA1) *or* lunchtime attack *if the oracle $\mathcal{O}_1$ decrypts messages (so $\mathcal{O}_1(C) = \mathcal{D}(C, sk)$) but the oracle $\mathcal{O}_2$ is trivial.*
- *The attack is said to be an* adaptive chosen ciphertext attack (CCA2) *if both oracles $\mathcal{O}_1$ and $\mathcal{O}_2$ decrypt messages, with the exception that the oracle $\mathcal{O}_2$ returns $\perp$ if it is queried on the challenge ciphertext.*

When the oracles $\mathcal{O}_1$ and $\mathcal{O}_2$ are not trivial, they are referred to as decryption oracles*.*

Again it is easy to see that if a system is secure against an attack in the CCA2 attack model then it is also resistant to that attack in the CCA1 and CPA models.

There has been much discussion about the appropriateness of each of the different types of attack model. It is relatively easy to envisage a situation where it is necessary for an encryption scheme to be resistant to attacks in the CCA1 model – for example, a malicious employee who only attempts to attack a system after he has been fired and therefore had his decryption privileges revoked. It is a lot harder to envision a situation where the CCA2 model is appropriate. It is difficult to see why an attacker with such strong decryption access would not be able to just decrypt the challenge ciphertext. However this is not the best way to think of the model. It is better to think of the CCA2 model as proving that the only way to break the system is to apply the decryption function to the challenge ciphertext.

The combination of success criteria and attack model is usually referred to by the combination of their abbreviations. For example, a scheme that is message-indistinguishable in the adaptive chosen ciphertext model is said to be IND-CCA2. This is the strongest measure of security and is the *de facto* standard for an asymmetric encryption scheme. Formally we will use the notion of an attacker derived in Sect. 6.2.1. A (t, ϵ, q_D) attacker in the IND-CCA2 model is a probabilistic Turing machine $\mathcal{A}$ that runs in time at most t, makes at most q_D queries to the decryption oracles (i.e. the total number of queries made to the oracles $\mathcal{O}_1$ and $\mathcal{O}_2$ is at most q_D), and distinguishes messages with advantage at least ϵ.

We will consider the security of the submitted schemes primarily in the IND-CCA2 model, i.e. we will consider the resistance to message-distinguishing attacks in the adaptive chosen ciphertext model. However, we will also consider the security of some of the submitted schemes in the IND-CPA model, i.e. in a model where the attacker does not have access to a decryption oracle. Obviously, any security reduction valid in the IND-CCA2 model is also valid in the IND-CPA model. Therefore the IND-CPA model is only of interest if it yields a tighter security proof or a security proof that reduces to a better underlying problem. In the IND-CPA model we will talk about a (t, ϵ) attacker: a probabilistic Turing machine $\mathcal{A}$ that runs in time at most t and distinguishes messages with advantage at least ϵ.

For a further discussion of security models the reader is referred to [45, 418].

6.2.3 Trusted cryptographic problems

As we have already mentioned, it is impossible to prove the security of an asymmetric encryption scheme directly. The best that can be done is to relate the security of a scheme back to some problem that is thought to be hard by the research community and trusted by developers. In this section we will discuss the various trusted problems to which the security of the submitted primitives can be reduced.

The NESSIE primitives use two distinct ‘flavours’ of trusted problems: those based on the difficulty of solving the discrete logarithm problem and its related problems, and those based on the difficulty of factoring composite numbers. For a further discussion of the trusted cryptographic problems used by the NESSIE primitives see [418].

6.2.3.1 Factoring based problems

All of the schemes based on factoring problems use arithmetic in the equivalence class $\mathbb{Z}/n\mathbb{Z}$ where n is some composite number. In all cases the ability to factor the number (or ‘modulus’) n implies the ability to solve the hard problem. For simplicity we define $\lambda(n) = l.c.m.(p-1, q-1)$ when $n = pq$.

Definition 6.6. *The following are trusted cryptographic problems and are all in some way related to the difficulty of factoring a large composite number.*

- *The factoring problem is the problem of finding the component factors of a composite integer n. In particular, we will concentrate on finding the component factors of an integer n of the form $p^d q$, where p and q are prime and $d \geq 1$.*

- *An RSA key is a pair* (n, e) *where* $n = pq$ *with* p *and* q *primes, and* $1 < e < n$ *with g.c.d.*$(\lambda(n), e) = 1$*. The RSA problem is the problem of finding an integer* $1 \leq x \leq n$ *such that* $x^e = y \bmod n$ *when given an RSA key* (n, e) *and a randomly selected integer* $1 \leq y \leq n$*.*
- *The e-th root problem is similar to the RSA problem, except that the exponent e is thought of as a constant. The e-th root problem is the problem of finding an integer* $1 \leq x \leq n$ *such that* $x^e = y \bmod n$ *when given a modulus* $n = pq$ *(with* p *and* q *primes) and a randomly selected integer* $1 \leq y \leq n$*.*

The factoring problem has been of interest to mathematicians for centuries and there are many algorithms that efficiently solve it for specific classes of 'modulus' n. However there are no known efficient algorithms for solving the factoring problem on a modulus $n = p^d q$ for small d (i.e. for $d < \sqrt{\log p}$). Again it is quite easy to see that if the RSA problem is hard for a modulus n then the factoring problem is hard for n too. The RSA problem was implicitly defined when the original RSA cryptosystem was proposed, and has been widely studied.

Whilst all of the above problems are well established, some of the schemes studied in phase I reduce to one of the following less well known problems.

Definition 6.7. *The following are trusted cryptographic problems:*

- *Consider a modulus of the form* $n = p^2 q$ *where the factorisation is unknown. The p-subgroup problem is the problem of deciding, given* $h \in (\mathbb{Z}/n\mathbb{Z})^*$*, if there exists a* $g \in (\mathbb{Z}/n\mathbb{Z})^*$ *such that* $g^p = h$*.*
- *For our purposes, the gap-factoring problem is the problem of finding the component factors of a composite integer* $n = p^2 q$ *when given access to the following oracle. The oracle returns the bit* b *when queried with* $g^b h^r$ *for unknown* $b \in \{0, 1\}$ *and* $r \in \{0, \ldots, p - 1\}$*, and fixed public* $g, h \in (\mathbb{Z}/n\mathbb{Z})^*$ *where*
  - g^p *has order* $p - 1$ *modulo* p^2*,*
  - $h = h_0^n \bmod n$ *for some* h_0*.*

These problems have not been subject to the same level of peer review by the scientific community as the factoring problem or the RSA problem; however there is no obvious reason to suspect that the p-subgroup problem or the gap factoring problem is particularly easier to solve than the RSA or factoring problems.

6.2.3.2 Discrete logarithm based problems

Problems related to the discrete logarithm problem are usually phrased in terms of a group $G = \langle g \rangle$ where g has prime order q. Technically, since all groups of prime order are isomorphic, the hardness of the problem depends not upon the group itself but upon the representation of the group. For example, the discrete logarithm problem is thought to be hard in elliptic curve subgroups but is definitely easy in the additive group of integers modulo q.

Definition 6.8. *Let* g *be an element of a group, and let* g *have order* q*.*

- *The discrete logarithm problem (DLP) is the problem of finding* a *when given* (g, g^a)*.*

- *The computational Diffie-Hellman problem (CDH) is the problem of finding g^{ab} when given (g, g^a, g^b).*
- *The decisional Diffie-Hellman problem (DDH) is the problem of deciding if $g^{ab} = g^c$ when given (g, g^a, g^b, g^c).*
- *The gap Diffie-Hellman problem (Gap-DH) is the problem of finding g^{ab} when given (g, g^a, g^b) and an oracle $\mathcal{O}$ that correctly solves the decisional Diffie-Hellman problem.*

Obviously if the DDH problem is hard in the group $\langle g \rangle$ then so is the CDH problem and the DLP. All of these problems are well established, except for the Gap-DH problem, which was only formally introduced in 2001 [498].

In situations where the security of an algorithm can be reduced to the difficulty of solving the CDH problem on some group it is not uncommon for the algorithm constructed to solve the CDH problem to output not a single answer to the problem but a small list of some L elements that contains the answer. Obviously, if one could solve the DDH problem on that group (either by means of a dedicated algorithm or by the use of some oracle) then selecting the correct answer from the list would be trivial, but we cannot assume that the DDH problem is tractable on the group so we are forced to use other means to find the correct answer.

The simplest, and fastest, method of selecting an answer is to pick an element of the list uniformly at random and output that element as the algorithm's final answer. The probability that this technique outputs the correct answer is $1/L$ times the probability that the correct answer appears on the list. Since L is typically of order related to q_D, the number of decryption oracle queries, this can have a significant impact on the efficiency of the reduction (see Sect. 6.2.9).

A more sophisticated approach has been suggested by Shoup [581]. Suppose that there exists an algorithm which, when given an instance of the CDH problem (g, g^a, g^b), outputs a (small) list of L elements that contains the solution to the CDH problem with probability at least ϵ. Then we may construct an algorithm that will output the correct answer to the CDH problem with probability at least $1 - (1/2)^k$ for some $k > 2$. However, this involves running the original algorithm $2k \lceil 1/\epsilon \rceil$ times with randomised inputs. Formally, if there exists an algorithm which outputs a list of L elements that contains the solution to the CDH problem with probability at least ϵ, and this algorithm runs in time bounded by t, then there exists a (t', ϵ') solver for the CDH problem with

$$\begin{aligned} \epsilon' &\approx 1 - \frac{1}{2^k}\,, & (6.3)\\ t' &\approx 2k \lceil 1/\epsilon \rceil\, t + 2kL \lceil 1/\epsilon \rceil\, T\,, & (6.4) \end{aligned}$$

where

- k is an integer greater than two,
- and T is the time taken to check an equation of the form

$$\left\{h \cdot g^{-ax_1y_2} \cdot g^{-bx_2y_1} \cdot g^{x_2y_2}\right\}^{x'_1y'_1} = \left\{h' \cdot g^{-ax'_1y'_2} \cdot g^{-bx'_2y'_1} \cdot g^{x'_2y'_2}\right\}^{x_1y_1}.$$

for random $x_1, y_1, x_2, y_2, x'_1, y'_1, x'_2, y'_2$ and fixed g^a and g^b.

6.2.3.3 Solving trusted cryptographic problems

As has been mentioned several times, even the trusted cryptographic problems in the previous two sections can be solved given enough computing power. Therefore, if we estimate the different amounts of computing power it would take to solve these problems then we may gain some measure of the comparative difficulty of seemingly unrelated problems. The notation $L_q[\alpha, c] = O(\exp((c + o(1))(\ln q)^{\alpha}(\ln \ln q)^{1-\alpha}))$ is used for estimating asymptotic complexity.

- **Integer factorisation.** The fastest known algorithms for factorising large integers are the Number Field Sieve [393] and the Elliptic Curve Method [397]. The asymptotic time taken by the number field sieve to factor an integer n is approximately $L_n[\frac{1}{3}, c]$, where c depends on the variant of the number field sieve used. The asymptotic time taken by the elliptic curve method to factor an integer whose smallest factor is p is $L_p[\frac{1}{2}, \sqrt{2}]$. Both of these algorithms are subexponential in the size of their input.
  An improvement of the elliptic curve method exists for $n = p^2 q$ [213, 517] and a special algorithm exists for $n = p^r q$ with large r [108].
- **RSA problem** The fastest method known for solving the general RSA problem involves factoring the modulus.
- ***e*-th root problem** The fastest method known for solving the e-th root problem (for $e \not\equiv 1 \bmod \lambda(n)$) involves factoring the modulus.
- **Discrete logarithm over $\mathbb{F}_p$.** The index-calculus method [144,221,492] is the fastest known method for solving the discrete logarithm problem over $\mathbb{F}_p$. It is closely related to the number field sieve factoring algorithm and has expected asymptotic running time of $L_p[\frac{1}{3}, c]$, which is subexponential in the input size.
- **Elliptic curve discrete logarithm.** The fastest general methods for solving the elliptic curve discrete logarithm problem are the Pollard ρ and the Pollard λ methods [524]. For a group with q elements, the Pollard ρ runs in time $\sqrt{\pi q/2}$ and the Pollard λ runs in time $2\sqrt{q}$ but can be faster in some special cases. Both can be efficiently parallelised [607] and have been slightly improved [250, 624]. No subexponential algorithm has been found for solving the elliptic curve discrete logarithm problem.
  However, there are subexponential attacks for specific elliptic curves: supersingular curves [236, 439, 550] and anomalous curves [554, 564, 587].

NESSIE has concluded that the following key sizes are roughly equivalent. For a further discussion on this subject, the reader is referred to Sect. 7.2.2.3.

Equivalent symmetric key size	56	64	80	112	128	160
Elliptic curve key size	112	128	160	224	256	320
Modulus length (pq)	512	768	1536	4096	6000	10000
Modulus length (p^2q)	570	800	1536	4096	6000	10000

It should be noted that these are estimates for classical computers. If it proves possible to build a quantum computer then there exist quantum algorithms that successfully solve both the discrete logarithm problem and the factorisation problem [580]. None of the asymmetric encryption algorithms submitted to NESSIE should be considered secure against attacks made by quantum computers.

6.2.4 The Random Oracle Model

Owing to the complex nature of asymmetric encryption schemes, it is very difficult to prove results about their security without making some assumptions about the properties of the components that make up the cipher. A security proof that does not make any assumptions is called a proof in the *standard model*.

The most common assumption used to simplify a proof is that a good hash function will behave exactly like a completely random function. This is the *random oracle model*, and was introduced by Bellare and Rogaway in 1993 [52]. Random functions (or oracles) and good hash functions share many properties, for example in both cases it is difficult to compute the pre-image of a given output or to find two elements that have the same image, and so this might be considered a reasonable modelling assumption. One of the common interpretations of a proof in the random oracle model used to be that if a scheme had a security proof in the random oracle model then that scheme was secure unless the particular hash function used interacted badly with the rest of the cryptosystem. This was considered unlikely to happen as hash functions are usually composed on the bit level whilst asymmetric encryption schemes take advantage of higher-level properties such as group structures.

Doubt, however, was cast on the random oracle model in a paper by Canetti, Goldreich and Halevi [135]. This paper proved that if there exists a cryptosystem that is secure in the random oracle model then there exists a cryptosystem that is secure in the random oracle model but insecure when the random oracle is replaced with *any* hash function. Whilst this means that the above interpretation is, in fact, incorrect, many people still accept it owing to the highly technical and theoretical nature of the results in [135].

For the purposes of NESSIE, proofs of security that were given in the random oracle model were accepted but regarded as heuristic.

6.2.5 Other models

There are several other models that have been used to examine cryptographic algorithms, but all of the NESSIE submissions were provided with proofs of security in either the standard or random oracle model. The only other model that could be of interest is the *generic group model* [474, 581].

The generic group model examines the security of schemes that can be implemented on many different groups, such as ECIES (see Sect. 6.4.2). A proof of security in the generic group model intends to show that a scheme is secure up to attacks that take advantage of the specific nature of the group on which the scheme operates. It models this by only giving the scheme access to a random encoding of a group element, rather than to the group element itself.

Unfortunately the generic group model was shown to have the same weaknesses as the random oracle model: if there exists a scheme that is provably secure in the generic group model then there exists a scheme that is provably secure in the generic group model but insecure when the random encoding function is replaced by *any* fixed encoding function [194, 230]. Also the generic security proofs cannot be provided for the schemes based on factoring problems, as the factoring problems tend to be defined on specific group encodings (usually $\mathbb{Z}/n\mathbb{Z}$). For this reason NESSIE has given less weight to security proofs given in the generic group model.

6.2.6 Validation of subgroup elements

Many asymmetric encryption schemes, particularly those based on discrete logarithm problems (see Sect. 6.2.3.2), involve group operations that take place in a prime order subgroup H of some much larger group G. Typically G will be either an elliptic curve group or the group of multiplicative integers modulo some value n, and H will be a cyclic subgroup generated by some element $g \in G$ of prime order. In situations where an algorithm takes an element of the subgroup H as input, either as part of a message, a ciphertext or as part of the secret key, it is important that the algorithm check that:

- the given element is an element of the group G,
- and the given element is a member of the subgroup H generated by g.

Failure to check that the given input is a member of the subgroup H may lead to attacks like the *small subgroup attacks* presented by Lim and Lee [399]. Failure to check that the given input is a member of the group G may lead to attacks such as those given by Biehl, Meyer and Müller [58] and Antipa *et. al.* [23].

It is usually fairly easy to avoid these attacks by checking that the given element is in G and in H. We separate these two conditions because the methods usually used for the checks are different in each case.

Suppose that an algorithm wishes to check that an element h is in G. If G is the group of multiplicative integers modulo n then the algorithm merely checks that h is an integer satisfying $1 \leq h \leq n$ and that $\gcd(h, n) = 1$. If the group G is an elliptic curve group over a field $\mathbb{F}$ then the algorithm merely checks that h defines a pair $(x, y) \in \mathbb{F} \times \mathbb{F}$ and that (x, y) satisfies the defining equation for the elliptic curve.

Suppose that an algorithm wishes to check that an element $h \in G$ is in $H \leq G$. Let p be the (prime) order of H and let r be the index of H in G. If G is cyclic (as is the case when G is the multiplicative group of a finite field) or $\gcd(p, r) = 1$ (as is often the case in elliptic curve groups) then the algorithm can check if $h \in H$ by checking to see if $h^p = 1$. If G is not cyclic and $\gcd(p, r) \neq 1$ then further information about the group is required. Note that these are not necessarily the most efficient tests. Note further that if the attacker has the ability to modify the elements of the key then the attacker is necessarily performing a side channel attack (see Sect. 6.2.7 and Annex A).

In our description of the submitted primitives (see Sect. 6.4 and Sect. 6.5) we will implicitly assume that all group elements have been validated before any computations are made that use them.

6.2.7 Side-channel attacks

Because of the increasing prominence of proofs of security in the field of asymmetric encryption, there has been an increase in interest in side-channel attacks. A side-channel attack is an attack that uses information or abilities that are assumed to be unavailable to an attacker in the normal attack models. This means that, essentially, a side-channel attack is the only method of breaking a provably secure asymmetric encryption scheme without breaking the underlying problem. For further information on side-channel attacks, the reader is referred to Annex A. For a further discussion of side-channel attacks against the asymmetric encryption primitives submitted to NESSIE, the reader is referred to [197, 207, 508].

6.2.8 Current standards

There are several bodies that are currently standardising asymmetric encryption schemes. These include RSA Laboratories, which produce the PKCS (Public-Key Cryptography Standards) [548], the SECG (Standards for Efficient Cryptography Group) [136] and the IEEE P1363 group [299, 300]. All of these groups also standardise other asymmetric schemes, including digital signature schemes (see Chapter 7).

The International Organisation for Standardisation (ISO) is also currently developing a standard that includes asymmetric encryption schemes (ISO/IEC 18033) [312,584]. This draft standard currently includes several of the asymmetric encryption schemes considered during Phase II of the NESSIE process, including RSA-KEM (see Sect. 6.4.6), PSEC-KEM (see Sect. 6.4.5), ECIES-KEM (see Sect. 6.4.3), ACE-KEM (see Sect. 6.4.1) and EPOC-2 (see Sect. 6.4.4).

6.2.9 Assessment criteria

The most important criterion in the NESSIE evaluation process is security. The submitters were encouraged to submit their asymmetric encryption primitives with a proof that they are secure in the IND-CCA2 model, although this proof could use the heuristic random oracle model. The security proofs were evaluated in terms of the hardness of the problem to which the security of the scheme reduces (see Sect. 6.2.3) and the efficiency of that reduction.

The efficiency of the reduction is the relationship between a (t, ϵ, q_D) attacker that breaks the cryptosystem and the implied (t', ϵ') solver that would solve the underlying trusted cryptographic problem. If we know the efficiency of the reduction then we can, by examining the best known methods for solving the underlying problem, estimate the maximum advantage ϵ that an attacker could have for attacking the encryption scheme. Normally we will also be able to determine the minimum size of the security parameter for which the cryptosystem has the security level specified in the NESSIE call. We classify the efficiency of the security reductions in the following way:

- the security reduction is tight if $\frac{t'}{\epsilon'} \approx \frac{t}{\epsilon}$,
- the security reduction is not so tight if $\frac{t'}{\epsilon'} \approx q_D \frac{t}{\epsilon}$,
- the security reduction is loose if $\frac{t'}{\epsilon'} \gg \frac{t}{\epsilon}$.

Usually we assume that $q_D \ll 2^\lambda$, where λ is the security parameter.

In order for a scheme to be practical we must balance the security of the scheme against its performance (speed and size). In order for these calculations to be fair and accurate we need to make certain comparisons between the various parameter sizes of the underlying trusted problems. We wish to use parameters that ensure that each of the problems can be solved with the same amount of computational power and that this computational difficulty is sufficient to protects the data. A discussion of these issues can be found in [573].

In accordance with the wishes of the NESSIE Project Industry Board, Intellectual Property Right (IPR) issues were also considered.

Lastly we used vulnerability to side-channel attacks only as a final factor in determining suitability. This is because an algorithm that is vulnerable to a side-channel attack can be protected by a careful implementation; indeed this is the most common solution according to the PIB. Only if it could be shown that a side channel attack applies to a primitive, regardless of the implementation (i.e. there is no known defence, or the primitive has an inherent weakness when confronted with side channel attacks), was it be taken into account as a selection criterion.

6.3 KEM-DEM cryptosystems

6.3.1 Hybrid encryption

In comparison with symmetric encryption algorithms, asymmetric encryption schemes are often slow and tend to have smaller message spaces. Consequently, in practice, asymmetric encryption schemes are often only used to encrypt a randomly generated symmetric key that is then used to encrypt a longer message. Asymmetric encryption schemes that use this technique are known as hybrid encryption schemes. The KEM-DEM model is a formalisation of this idea. It was introduced in a later version of [171], which also included the security analysis of ACE-KEM described in Sect. 6.4.1, and in the draft ISO proposal [584].

A KEM-DEM cryptosystem is composed of two algorithms: a *key encapsulation mechanism (KEM)* and a *data encapsulation mechanism (DEM)*. A key encapsulation mechanism is a scheme that, given a public-key, derives a random key and provides a method of encrypting (encapsulating) and decrypting (decapsulating) that random key. This typically uses asymmetric techniques. The data encapsulation mechanism uses that random key to encrypt a message. This typically uses symmetric techniques. Formally,

Definition 6.9. *A* key encapsulation mechanism (KEM) *is a triple of deterministic algorithms $(\mathcal{G}, KEM.Encrypt, KEM.Decrypt)$. As before, $\mathcal{G}$ is a key generation algorithm that takes as input a unary string 1^λ, where λ is called the security parameter, and a fixed length random seed r, and outputs a key-pair (pk, sk). The encapsulation algorithm, $KEM.Encrypt$, takes the public-key pk and a random seed as input and outputs a pair (K, ψ). The decapsulation algorithm, $KEM.Decrypt$, takes as input an encapsulation ψ and the secret-key sk, and outputs a key value K' or the error symbol $\perp$.*

As before we require that the KEM is sound, i.e. for any valid key-pair (pk, sk) the decapsulation of an encapsulated key is the key itself, or, in other words, if $KEM.Encrypt(pk, r) = (K, \psi)$ then $KEM.Decrypt(\psi, sk) = K$. We will also need some kind of security result that limits an attacker's ability to derive information about the key K from the public-key pk and the encapsulation ψ. Notice that the KEM encapsulation algorithm does only take as input a random seed and the public-key pk, which is often considered to be a system parameter, and never has access to the message.

Again, although we define both the key generation algorithm and the encapsulation algorithm as deterministic, it is often easier to think of them as probabilistic algorithms. So, when the context is sufficiently clear, we will implicitly assume the presence of a random input and that any randomness required by an algorithm is actually derived from the random input provided.

Definition 6.10. *A* data encapsulation mechanism (DEM) *is a pair of deterministic algorithms $(DEM.Encrypt, DEM.Decrypt)$.*[2] *The encryption algorithm $DEM.Encrypt$ takes as input a message m and a key K and computes a ciphertext χ. The decryption algorithm $DEM.Decrypt$ takes as input a ciphertext χ and a key K and outputs a message m' or the error symbol $\perp$.*

The DEM must also be sound, i.e. for all valid keys K and messages m we have that $DEM.Decrypt(DEM.Encrypt(m, K), K) = m$, and it must satisfy some security condition. It should also be noted here that the DEM does not have access to the public key used by the KEM, only the symmetric key produced by the KEM and the message itself.

Definition 6.11. *A KEM-DEM based cryptosystem is a hybrid asymmetric encryption scheme composed of a KEM ($\mathcal{G}$, $KEM.Encrypt$, $KEM.Decrypt$) and a DEM ($DEM.Encrypt$, $DEM.Decrypt$) where the output key-space of the KEM is the same as the key-space of the DEM. To encrypt a message m the hybrid scheme runs as follows:*

1. *Generate a random seed r.*
2. *Run $KEM.Encrypt(pk, r)$ to produce an encapsulation pair (K, ψ).*
3. *Run $DEM.Encrypt(m, K)$ to produce a ciphertext χ.*

[2] Technically there is no reason why the DEM should not be composed of probabilistic algorithms. However, because of the computational problems associated with generating random or pseudo-random bits, we do not recommend that probabilistic algorithms are used.

4. *Output* $C = (\psi, \chi)$.

Hence the decryption algorithm for a ciphertext C *is*

1. *Parse* C *as appropriately sized* ψ *and* χ.
2. *Run* $KEM.Decrypt(\psi, sk)$ *to obtain a key* K' *or* $\perp$.
3. *Run* $DEM.Decrypt(\chi, K')$ *to obtain a message* m' *or* $\perp$.
4. *Output* m' *or* $\perp$.

Key generation for the hybrid scheme is provided by $\mathcal{G}$.

At this point it might be helpful to consider an elementary example of a KEM-DEM based cryptosystem. The first asymmetric encryption scheme that can be modelled as a KEM-DEM cryptosystem is the ElGamal scheme [220]. This uses a KEM based on the Diffie-Hellman key agreement protocol and a DEM based on modular multiplication.[3]

This provides a good example of the differences between a key encapsulation mechanism and a key agreement or key exchange protocol. Although a key agreement protocol can be used as the basis for a KEM, the security requirements for a KEM are different from those of a key agreement protocol. A KEM is only required to produce an encapsulation of a key that will be secure *when used once.* The ElGamal scheme is a good example: the scheme is provably as secure as the decisional Diffie-Hellman problem (see Sect. 6.2.3) when a new key is generated for each encryption, but it is totally insecure when keys are reused.

6.3.2 KEM Security

One of the aims of the KEM-DEM model was to provide a security analysis based on the component parts. The security of the KEM is based on the inability of an attacker to distinguish a proper encapsulation pair from a random pair.

Definition 6.12. *Consider the following game an attacker* $\mathcal{A}$ *plays against a system using a KEM* $(\mathcal{G}, KEM.Encrypt, KEM.Decrypt)$ *with a security parameter* λ.

1. *The system runs* $\mathcal{G}(1^\lambda, r)$ *(for some suitably random seed* r*) to generate a random key-pair* (pk, sk) *and passes* pk *to the attacker.*
2. *The attacker runs until it requests a challenge encapsulation.*
3. *The system generates the challenge in the following way:*
   a) *The system generates a suitably random seed* r.
   b) *Next, it runs* $KEM.Encrypt(pk, r)$ *to generate a pair* (K_0, ψ).
   c) *The system then generates a key* K_1 *uniformly at random from the entire output space of the KEM.*
   d) *Lastly, it picks a bit* σ *uniformly at random from* $\{0, 1\}$ *and returns* (K_σ, ψ) *to the attacker.*

[3] Technically, ElGamal does not fit into the formal KEM-DEM model as the DEM requires access to the public key in order to encrypt the message. However the structures are sufficiently similar to be enlightening.

4. *The attacker outputs a guess σ' for σ.*

The attacker wins the above game if $\sigma' = \sigma$.

The advantage of an attacker $\mathcal{A}$ is

$$Adv_{\mathcal{A}} = Pr[\sigma' = \sigma] - 1/2 \tag{6.5}$$

and the advantage of the KEM is said to be

$$AdvKEM = \max_{\mathcal{A}} Adv_{\mathcal{A}} \ . \tag{6.6}$$

A KEM is said to be indistinguishable *or* IND *if its advantage is small.*

A (t, ϵ) attacker for a KEM in the IND-CPA model is a probabilistic Turing machine $\mathcal{A}$ that runs in time bounded above by t and has advantage at least ϵ. A (t, ϵ, q_D) attacker for a KEM in the IND-CCA2 model is a probabilistic Turing machine $\mathcal{A}$ that runs in time bounded above by t, makes at most q_D decryption queries and has advantage at least ϵ.

Most key encapsulation mechanisms are composed of some kind of security mechanism which is highly algebraic, and some kind of key derivation function (KDF). The purpose of the key derivation function is more than just formatting: it takes some raw key or seed produced by the security mechanism and produces an appropriately sized bit string that has been stripped of all its algebraic properties. This usually involves some kind of hash function and is often modelled as a random oracle. A security mechanism is defined as follows.

Definition 6.13. *A security mechanism is a pair (Mech.Encrypt,Mech.Decrypt) of algorithms along with some key generation algorithm $\mathcal{G}$. The encryption function Mech.Encrypt takes as input a random seed r and the public-key pk, and outputs a pair (K_{raw}, ψ). The decryption function Mech.Decrypt inverts this operation by returning K_{raw} when given ψ and the secret key sk.*

This 'pared down' version of the KEM allows us to show that, in the random oracle model at least, key encapsulation mechanisms are fairly abundant. The following is shown in [193].

Theorem 6.1. *If (Mech.Encrypt, Mech.Decrypt) is a security mechanism that is OW-CCA2 and KDF is a key derivation function, then we may define a KEM $(\mathcal{G}, KEM.Encrypt, KEM.Decrypt)$ in the following manner. Key generation is provided by the key generation algorithm of the security mechanism. We define the encapsulation function $KEM.Encrypt(pk, r)$ as follows:*

1. *Compute $Mech.Encrypt(pk, r) = (K_{raw}, \psi)$.*
2. *Compute $K = KDF(K_{raw})$.*
3. *Output (K, ψ).*

The corresponding decryption function $KEM.Decrypt(\psi, sk)$ can then be defined as follows:

1. *Compute $Mech.Decrypt(\psi, sk) = K_{raw}$.*

2. Compute $K = KDF(K_{raw})$.
3. Output K.

If the security mechanism is OW-CCA2 (in the obvious sense, i.e. that an attacker is unable to recover K_{raw} from ψ) then, in the random oracle model, the KEM is IND-CCA2.

As will be seen, both the RSA protocol and the Diffie-Hellman key agreement protocol make good security mechanisms and this construction will be used in RSA-KEM (see Sect. 6.4.6) and ECIES-KEM (see Sect. 6.4.3). Further generic constructions for KEMs from low-level primitives can be found in [195].

There is also a further distinction that one can make with regard to key encapsulation mechanisms. Of the KEMs presented there seem to be two separate flavours: *authenticated* key encapsulation mechanisms and *unauthenticated* key encapsulation mechanisms. An authenticated KEM is a KEM where a decapsulated key is only released if the encapsulation satisfies some kind of extra criteria that give the user some assurance that the encapsulation was properly constructed and not just some kind of guess. A good example of this is the difference between ECIES-KEM (see Sect. 6.4.3) and ACE-KEM (see Sect. 6.4.1). ACE-KEM uses a method similar to ECIES-KEM for the actual key encapsulation but adds extra elements to authenticate the fact that a key was encapsulated properly. The difference is typified by the fact that an unauthenticated KEM will generally decapsulate any encapsulated key, whilst an authenticated KEM will reject most randomly formed encapsulations. Although authenticated KEMs usually have better security proofs, i.e. security proofs that work in stronger models or reduce more efficiently to weaker assumptions, some concern has been raised as to whether it is necessary to authenticate a key that will be used to decrypt a message that will, most likely, itself be authenticated [193].

6.3.3 Key derivation functions

Almost all of the asymmetric encryption primitives use some kind of key derivation function (KDF) or mask generating function (MGF). A mask generating function is a function used to create a mask for some secret value. Whilst the uses of key derivation functions and mask generating functions are slightly different, the properties that these functions must have and the methods used to construct them seem to be the same. Hence we will only refer to key derivation functions in this section, with the understanding that all of the following discussion is relevant to mask generating functions as well.

Key derivation functions are similar to hash functions in that they map bit strings of any length to fixed length bit strings in an almost random way. Key derivation functions are very similar to hash functions. Hash functions map a bit string to a bit string of a fixed length determined by the hash function. KDFs and MGFs are families of functions that map a bit string onto a bit string of a fixed length but this length is determined by the public key. KDFs are usually based on hash functions (see Chapter 4) and are often modelled as random oracles (see Sect. 6.2.4) for simplicity.

Whether we regard KDFs (and MGFs) as distinct cryptographic entities or modes of operation of a hash function, they are still outside the initial scope of NESSIE and so have not in themselves received a high level of scrutiny from the NESSIE partners.

With the exception of ACE-KEM (see Sect. 6.4.1), each of the primitives that uses a KDF (or MGF) models it as a random oracle. Hence if more than one random oracle is used, for example if a scheme uses a hash function and a KDF and the security analysis models both as random oracles, the security analysis is only valid if each of the implemented functions are suitably independent of each other. For ACE-KEM the KDF is required to have an output that is indistinguishable from random even when some of the leftmost bits of the input are known.

All of the key derivation functions submitted to NESSIE use one of two techniques to construct a KDF $KDF(\cdot)$ from a hash function $Hash(\cdot)$. Suppose $Hash(\cdot)$ is a hash function with output size $HashLen$ and that we wish to evaluate $KDF(\cdot)$ on an input x and get an output of length $KDFLen(\lambda, pk)$. Set

$$BlockNum(\lambda, pk) := \left\lceil \frac{KDFLen(\lambda,pk)}{HashLen} \right\rceil .$$

The first technique, known as *KDF1* [584] or *MGF1* [549], is to use the leftmost $KDFLen(\lambda, pk)$ bits of the string

$$Hash(x||0_{32}) \,||\, \ldots \,||\, Hash(x||(BlockNum - 1)_{32}) \,,$$

where i_{32} is the 32-bit representation of the integer i. The second technique, known as $KDF2$ [584], is to use the leftmost $KDFLen(\lambda, pk)$ bits of the string

$$Hash(x||1_{32}) \,||\, \ldots \,||\, Hash(x||BlockNum_{32}) \,,$$

where i_{32} is the 32-bit representation of the integer i. Notice that both of these techniques fail if the output size of the KDF is required to be too large, i.e. if $KDFLen(\lambda, pk) > 2^{32} \cdot HashLen$.

These techniques have the advantage that the output of the KDF is random providing the underlying hash function is random (and unavailable to the attacker). However these functions have been criticised by Shoup [584] because of the way some hash functions work. Whilst this criticism appears valid, it does not appear to be a critical flaw in the design.

Another problem with modelling both hash functions and key derivation functions as random oracles is that the outputs of each of these functions need to be independent of each other. Obviously if the KDF is either *KDF1* or *KDF2* then the output will be correlated to the output of the hash function $Hash(\cdot)$. The simplest solution would be for each primitive to use a different hash function, but this is often impractical. Another good solution is to assign each component that uses the hash function, including the hash function itself, a unique fixed length identifier *id* and prefix all inputs to the hash function with this value. In this case, the outputs of the hash function and the key derivation functions *KDF1* and *KDF2* will be random and uncorrelated.

6.3.4 DEM Security

It might be reasoned from some of the discussion in this chapter that the security of the KEM is in some way more important than the security of the DEM. This is, of course, not true. As we shall see in Sect. 6.3.5, the security of the hybrid scheme depends in equal measure on the security of the KEM and the DEM.

We have seen that the key produced by a KEM is very similar to a random key, so it makes sense to examine the security of the DEM under the action of a random key. Furthermore, since it makes no sense to query a DEM decryption oracle before a challenge key has been produced, we will have to tweak the attack models for the DEM slightly.

Definition 6.14. *Consider the following game an attacker plays against a system, using a DEM* $(DEM.Encrypt, DEM.Decrypt)$ *with a security parameter* λ.

1. *The system generates a key* K *for the DEM uniformly at random from the key-space of the DEM (which is defined in terms of the security parameter* λ*).*
2. *The attacker generates two distinct messages* m_0 *and* m_1 *of the same length, and submits these to the system.*
3. *The system*
   a) *Chooses a bit* σ *uniformly at random from* $\{0, 1\}$.
   b) *Calculates the challenge ciphertext* $\chi = DEM.Encrypt(m_\sigma, K)$ *and returns this to the attacker.*
4. *The attacker outputs a guess* σ' *for* σ.

The attacker wins the game if $\sigma' = \sigma$.

An attacker $\mathcal{A}$ *has an* advantage $Adv_{\mathcal{A}}$ *of winning the above game where*

$$Adv_{\mathcal{A}} = Pr[\sigma' = \sigma] - 1/2 \tag{6.7}$$

and the DEM is said to have advantage

$$AdvDEM = \max_{\mathcal{A}} Adv_{\mathcal{A}} \; . \tag{6.8}$$

A DEM is said to be message-indistinguishable *or* IND *if its advantage is small.*

The attack model for a DEM also follows the standard ideas.

Definition 6.15. *We allow the attacker* $\mathcal{A}$ *access to an oracle* $\mathcal{O}$ *after the challenge has been issued. The attack is said to be* passive (PAS) *if the oracle always returns the error symbol* $\perp$. *The attack is said to be a* chosen ciphertext attack (CCA) *if the oracle decrypts messages under the challenge key* K *(hence* $\mathcal{O}(\chi) = DEM.Decrypt(\chi, K)$*) with the exception that the oracle returns* $\perp$ *if it is queried with the challenge ciphertext.*

Obviously if a scheme is secure against an attack in the chosen ciphertext model then it is secure in the passive model, and it is easier to demonstrate the security of a scheme in the passive model than in the chosen ciphertext model. A result of [171] shows that it is possible to combine a DEM that is secure in the passive model with a message authentication code (MAC) (see Chapter 5) to give a DEM that is secure in the chosen ciphertext model.

6.3.5 Hybrid Security

Of course the aim of this entire section is to provide some insight into the security of a hybrid KEM-DEM based cryptosystem. So, whilst it might be very interesting to talk about the security of the KEM and the DEM in abstract models, it is useless unless we can combine these results to prove something about the security of the hybrid scheme. In particular we wish to show that a KEM-DEM based cryptosystem achieves the minimum security requirements specified by the NESSIE call, i.e. that the scheme is IND-CCA2 secure.

A security proof for the KEM-DEM construction was given in [171]. It basically shows that a KEM-DEM scheme composed of a secure DEM and a secure KEM will itself be secure. This shows that the method of construction is in some sense a "black-box" construction: any secure KEM and secure DEM can be chosen to give a secure scheme without reference to each other.

Theorem 6.2. *Suppose* $(\mathcal{G}, \mathcal{E}, \mathcal{D})$ *is a KEM-DEM scheme composed of a KEM and a DEM in the usual manner. If the KEM is IND-CCA2 secure and the DEM is IND-CCA secure then the overall hybrid scheme will be IND-CCA2 secure in the usual sense.*

There has been some work [275] that shows that if the public key *pk* used by the KEM is considered a system parameter available to all parties then the above theorem no longer holds. A separate security proof for this case was also proposed in that paper but, unfortunately, this security proof no longer allows a black-box construction.

Theorem 6.3. *Suppose* $(\mathcal{G}, \mathcal{E}, \mathcal{D})$ *is a KEM-DEM scheme composed of a KEM and a DEM in the usual manner. Suppose further that the DEM takes the public key as an extra input. If the KEM is IND-CCA2 secure and the DEM is IND-CCA secure* **when the attacker has access to a decryption oracle for the KEM**, *then the overall hybrid scheme is IND-CCA2 secure in the standard sense.*

In practice it seems very unlikely that a DEM would consider using the public-key in any way that would compromise the security of the system. However it should be noted that the ElGamal scheme and the EPOC-2 scheme (see Sect. 6.4.4) both fit into this formal model of a KEM-DEM based scheme, as both contain modular arithmetic operations in the DEM phase.

One generic problem associated with KEM-DEM constructions is message expansion. Since only the ciphertext produced by the DEM depends on the message, the complete ciphertext is always bigger than the plaintext by at least a number of bits equal to the size of the encapsulation produced by the KEM. In practice the amount of message expansion may be larger even than this as the DEM ciphertext may contain bits that do not directly relate to the encryption of the message but are instead some kind of assurance of the integrity of the message (such as a MAC). Therefore in situations where message expansion is a critical factor, it might be best to avoid using a hybrid construction.

6.3.6 Assessment criteria

The assessment process for KEM-DEM based cryptosystems was roughly the same as for general asymmetric encryption schemes (see Sect. 6.2.9). The most important criterion is security. However, since this section of the NESSIE project is concerned with asymmetric techniques, we have concentrated our resources on examining the security of the key encapsulation mechanisms. We therefore assume the existence of a DEM that is equally secure for each KEM even in the presence of a decryption oracle for that KEM. This greatly simplifies matters as the security of the KEM can be examined independently of the DEM. Again, each of the submitted primitives came with a proof of security and these proofs were evaluated in terms of the efficiency of their reductions and the hardness of the underlying trusted problems.

Next, performance was considered, on platforms of equal security, and lastly side-channel attacks were considered only if they could be applied to multiple platforms. Intellectual Property Rights (IPR) were also considered.

6.4 Asymmetric encryption primitives considered during Phase II

The following algorithms were selected for study during phase II of the NESSIE project:

- ACE-KEM,
- ECIES,
- EPOC-2,
- PSEC-KEM,

and, because they are under discussion in ISO [584], the following algorithms were selected for study during NESSIE phase II as *de facto* standards for asymmetric encryption algorithms:

- ECIES-KEM,
- RSA-KEM.

We deal with the security considerations for each algorithm in turn. Note that what is given here are not a complete specifications but rather mathematical bases for the algorithms. In particular we assume that variables are stored in binary form even if they are integers, elliptic curve points, etc. We also assume that all hash functions, mask generating functions, key derivation functions and symmetric encryption schemes take inputs and produce outputs of a "correct" length for the asymmetric scheme. References are given to complete specifications which may be found on the NESSIE website.

6.4.1 ACE-KEM

The ACE-KEM cryptosystem is based on the work of Cramer and Shoup [171] and is purposely designed to be secure without needing the heuristic random oracle model. It has been submitted to NESSIE by IBM. The scheme consists loosely of the following algorithms. A complete specification is given in [584].

6.4.1.1 The design

Key Generation. ACE-KEM is described on an abstract group and so can be realised either as an elliptic curve scheme or as a scheme working in the multiplicative group of integers for some modulus. We will represent the group as an additive group where group elements are represented as capital letters (as in an elliptic curve group). We assume that the group is a cyclic group generated by some element P of order p and that p has length equal to the security parameter λ. The key generation algorithm is a probabilistic algorithm that takes the group parameters (P, p, λ) as input. It runs as follows.

1. Generate random and independent integers $w, x, y, z \in \{0, \ldots, p-1\}$.
2. Set $W := wP$, $X := xP$, $Y := yP$ and $Z := zP$.
3. Set $pk := (P, p, W, X, Y, Z, \lambda)$ and $sk := (w, x, y, z, pk)$.
4. Output the key-pair (pk, sk).

Encapsulation Algorithm. The encapsulation algorithm is a probabilistic algorithm that takes the public-key pk as input. It uses a public and pre-agreed hash function $Hash(\cdot)$ and key derivation function $KDF(\cdot)$. It runs as follows.

1. Generate a random integer $r \in \{0, \ldots, p-1\}$.
2. Set $C_1 := rP$.
3. Set $C_2 := rW$.
4. Set $Q := rZ$.
5. Set $\alpha := Hash(C_1 || C_2)$.
6. Set $C_3 := rX + \alpha rY$.
7. Set $C := (C_1, C_2, C_3)$.
8. Set $K := KDF(C_1 || Q)$.
9. Output the encapsulated key-pair (K, C).

The ACE-KEM encapsulation algorithm is also shown pictorially in Fig. 6.19.

Decapsulation Algorithm. The decapsulation algorithm is a deterministic algorithm that takes a key encapsulation C and the secret-key sk as input. It also uses the pre-agreed hash function $Hash(\cdot)$ and key derivation function $KDF(\cdot)$. It runs as follows.

1. Parse the encapsulated key C as (C_1, C_2, C_3).
2. Set $\alpha := Hash(C_1 || C_2)$.
3. Set $t := x + y\alpha \bmod p$.
4. Check that $C_2 = wC_1$. If not, output `Invalid Ciphertext` and halt.
5. Check that $C_3 = tC_1$. If not, output `Invalid Ciphertext` and halt.
6. Set $Q := zC_1$.
7. Set $K := KDF(C_1 || Q)$.
8. Output K.

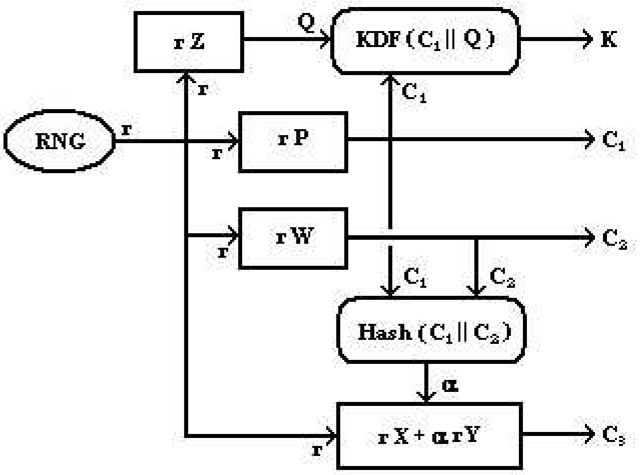


Fig. 6.19. The ACE-KEM encapsulation algorithm.

6.4.1.2 Security analysis

The following is a summary of the security analysis of ACE-KEM; for more details see [141, 193].

The main advantage of ACE-KEM is that the security of the scheme can be proven without the use of the heuristic random oracle model (see Sect. 6.2.4). This combines nicely with the security proof for the generic hybrid construction discussed in Sect. 6.3.5 to form a hybrid scheme that is provably secure without the need for random oracles.

The security proof for ACE-KEM [171] reduces the problem of breaking ACE-KEM in the IND-CCA2 sense to the problem of solving the decisional Diffie-Hellman problem in the group generated by P. As we do not model the hash function or the key derivation function as random oracles we must formally define their security conditions.

Definition 6.16. *A hash function $Hash(\cdot)$ is 2^{nd} pre-image resistant if, given a value x, it is hard to find a value $y \neq x$ such that $Hash(x) = Hash(y)$. We define a (t, ϵ) attacker for a hash function to be a probabilistic Turing machine that runs in time bounded above by t and, given a randomly generated x, finds a second pre-image with probability at least ϵ.*

Definition 6.17. *A key derivation function $KDF(\cdot)$ is indistinguishable if, given x, it is computationally infeasible to distinguish between $KDF(x||y)$ and a randomly generated bit string of the same length when y is unknown. We define a (t, ϵ) attacker for a key derivation function to be a probabilistic Turing machine that runs in time bounded above by t and solves the above problem with probability at least $1/2 + \epsilon$.*

If there exists a (t, ϵ, q_D) attacker for ACE-KEM in the IND-CCA2 sense then there exists a (t_1, ϵ_1) solver for the decisional Diffie-Hellman problem on the group generated by P, a (t_2, ϵ_2) attacker for the hash function $Hash(\cdot)$ and a (t_3, ϵ_3) attacker for the key derivation function $KDF(\cdot)$ with

$$\begin{aligned} \epsilon &\approx \epsilon_1 + \epsilon_2 + \epsilon_3 + \frac{q_D}{p}\,, && (6.9)\\ t &\approx t_i \text{ for } i \in \{1,2,3\}\,. && (6.10) \end{aligned}$$

Whilst the reduction to the decisional Diffie-Hellman (DDH) problem is tight, the assumption that the DDH problem is hard is quite a strong one. For better comparison to existing schemes the submitters have also shown that ACE-KEM is at least as secure as ECIES-KEM (see Sect. 6.4.3). Formally, if there exists a (t, ϵ, q_D) attacker for ACE-KEM then there exists a (t, ϵ, q_D) attacker for ECIES-KEM. This serves to show that, in the random oracle model, the security of ACE-KEM in the IND-CCA2 sense tightly reduces to the problem of solving the gap Diffie-Hellman problem on the group generated by P.

The relationship between ACE-KEM and ECIES-KEM also means that in the IND-CPA model, and with the help of the random oracle model, the security of ACE-KEM can be reduced to the problem of breaking the computational Diffie-Hellman (CDH) problem in the group $\langle P \rangle$.[4] Formally, if there exists a (t, ϵ) attacker for ACE-KEM then there exists an algorithm running in time t' that outputs a list of L elements and contains a solution to the CDH problem with probability at least ϵ' with

$$\begin{aligned} \epsilon' &\approx \epsilon\,, && (6.11)\\ t' &\approx t\,, && (6.12)\\ L &\leq q_K\,, && (6.13) \end{aligned}$$

where the attacker makes at most q_K queries of the random oracle simulating the key derivation function. Of course, we may now use the techniques of Sect. 6.2.3.2 to construct a (t'', ϵ'') solver for the CDH problem with

$$\begin{aligned} \epsilon'' &\approx 1 - \frac{1}{2^k}\,, && (6.14)\\ t'' &\approx 2k\lceil 1/\epsilon' \rceil t' + 2kL\lceil 1/\epsilon' \rceil T\,, && (6.15) \end{aligned}$$

where T is the time taken to compute a group element of the form

$$x_1^{-1} y_1^{-1} \big\{ P' - a x_1 y_2 P - b x_2 y_1 P + x_2 y_2 P \big\}\,,$$

for random integers x_1, y_1, x_2, y_2 and a random elliptic curve point P'.

It is unclear to what extent the small subgroup attacks of Lim and Lee are applicable to ACE-KEM (see Sect. 6.2.6). A naive analysis suggests that to find

[4] In the IND-CCA2 attack model, and using the random oracle model, the security of ACE-KEM can be directly reduced to the problem of breaking the CDH problem in the group generated by P [582]. However this reduction is far from tight.

z mod n requires at most n^2 trial decapsulations and n evaluations of the key derivation function (if the elliptic curve points are not validated).

Along with most of the other asymmetric encryption schemes, ACE-KEM is vulnerable to a fault attack that recovers the secret key [197]. It also appears to be vulnerable to power analysis during the scalar multiplications of the elliptic curve points. This is a common problem with schemes based on elliptic curves: for more information see Sect. A.1.2.3.

Recently a new version of the ACE-KEM scheme has been proposed by Lucks [409]. This new scheme works in the multiplicative group of integers modulo n, where $n = PQ$, $P = 2p + 1$, $Q = 2q + 1$ and P, Q, p, q are prime numbers. This scheme has the disadvantage of working in a very specific, multiplicative group (and so will probably have longer keys than the elliptic curve version) but has the advantage of a stronger security proof. The Lucks scheme is secure in the standard model providing the factoring problem is secure and the DDH problem is hard in the group of quadratic residues modulo n. This is worse than ACE-KEM as ACE-KEM does not require the factoring problem to be difficult. However, in the random oracle model, the Lucks scheme reduces to solving the CDH problem in a very efficient manner. This is a significant improvement over ACE-KEM in theoretical security.

6.4.2 ECIES

ECIES is a hybrid encryption scheme submitted to NESSIE by Certicom Corp. It loosely consists of the following algorithms. A complete specification can be found in [136].

6.4.2.1 The design

Key Generation. Since ECIES is an elliptic curve based cryptosystem, it is necessary to generate a suitably secure elliptic curve E and a point $P \in E$ that has prime order p. We will assume that the length of p is equal to the security parameter λ. The key generation algorithm for ECIES is a probabilistic algorithm that takes (E, P, p, λ) as input. It runs as follows. (For notational purposes, let $\mathcal{O}$ be the 'point at infinity' – the identity element of an elliptic curve group.)

1. Randomly generate an integer $s \in \{1, \ldots, p-1\}$.
2. Set $W := sP$.
3. Set $pk := (E, P, p, W, \lambda)$ and $sk := (s, pk)$.
4. Output the key-pair (pk, sk).

Encryption Algorithm. The encryption algorithm is a probabilistic algorithm that takes as input a message m and the public-key pk. The scheme also relies on a set of public and pre-agreed functions that are available to all parties. These include a key derivation function $KDF(\cdot)$, a message authentication code algorithm $MAC(\cdot, \cdot)$ and a symmetric encryption scheme (*Sym.Encrypt*, *Sym.Decrypt*). It runs as follows.

1. Generate a random integer $r \in \{1, \ldots, p-1\}$.

2. Set $C_1 := rP$.
3. Set $Q := rW$.
4. Check that $Q \neq \mathcal{O}$. If so, output `Invalid Encryption` and halt.
5. Set x to be the x-coordinate of Q.
6. Set $K := KDF(x)$.
7. Parse K as EK and MK, where EK is a suitably sized key for the symmetric encryption scheme and MK is a suitably sized key for the message authentication scheme. [5]
8. Encrypt the message m using the symmetric encryption scheme under the key EK, i.e. $C_2 := \mathit{Sym.Encrypt}(m, EK)$.
9. Generate a message authentication code (MAC) for the ciphertext C_2 using the message authentication scheme under the key MK, i.e. $C_3 := MAC(C_2, MK)$.
10. Output the ciphertext $C := (C_1, C_2, C_3)$.

Decryption Algorithm. The decryption algorithm is a deterministic algorithm that takes a ciphertext C and the secret-key sk as input. It also uses the same pre-agreed key derivation function $KDF(\cdot)$, message authentication code algorithm $MAC(\cdot, \cdot)$ and symmetric encryption scheme (*Sym.Encrypt*, *Sym.Decrypt*) as the encryption algorithm. It runs as follows.

1. Parse C as (C_1, C_2, C_3).
2. Set $Q := sC_1$.
3. Check that $Q \neq \mathcal{O}$. If so, output `Invalid Ciphertext` and halt.
4. Set x to be the x-coordinate of Q.
5. Set $K := KDF(x)$.
6. Parse K as EK and MK, where EK is a suitably sized key for the symmetric encryption scheme and MK is a suitably sized key for the message authentication scheme.
7. Check that C_3 is a valid message authentication code for C_2 under the key MK, i.e. that $C_3 = MAC(C_2, MK)$. If not, output `Invalid Ciphertext` and halt.
8. Decrypt the ciphertext C_2 using the symmetric encryption scheme under the key EK, i.e. $m := \mathit{Sym.Decrypt}(C_2, EK)$.
9. Output m.

6.4.2.2 Security analysis

The following is a summary of the security analysis of ECIES; for more details see [571].

The first thing to notice about the ECIES algorithm as it stands is that it is not secure in the IND-CCA2 sense. If an attacker is given a challenge ciphertext (C_1, C_2, C_3) then he may submit the ciphertext $(-C_1, C_2, C_3)$ to the decryption oracle (providing $C_1 \neq -C_1$, a condition which occurs with overwhelming probability) and the oracle will return the correct message m. Therefore if we are going

[5] Note that if the lengths of EK and MK are not predetermined, then the attack of Shoup [584, 15.6.4] applies.

to find any meaningful result on the security of ECIES in this model then we will have to formally deny the attacker the power to make these queries. Shoup [584] calls this property *benign malleability.*

The security proof for ECIES is sketched in [3]. This paper describes the security of the DHAES scheme, a scheme similar to ECIES but defined on an abstract cyclic group of order p rather than specifically on an elliptic curve group. It differs from ECIES in the generation of the symmetric key K, which, in DHAES, is derived from a complete representation of Q and from C_1. The security proof shows that, provided the symmetric encryption scheme and the MAC are in some sense secure, the problem of breaking the scheme reduces to the problem of differentiating the output of the key derivation function from a random string of the same size.

A security proof for ECIES in the generic group model (see Sect. 6.2.5) has also been proposed [588].

ECIES is essentially an example of ECIES-KEM (see Sect. 6.4.3) used with a particular data encapsulation mechanism (known as DEM-1 [584]). We consider ECIES-KEM to be more flexible than ECIES without having any computational or security loss.

6.4.3 ECIES-KEM

ECIES-KEM is essentially the asymmetric heart of ECIES described in the KEM-DEM framework. Although ECIES-KEM was never formally submitted to the NESSIE project, it is considered by NESSIE as a *de facto* standard for a KEM-DEM based cryptosystem and because it has been proposed for standardisation in the ISO/IEC standard 18033-2 [584]. It consists loosely of the following algorithms. A complete specification can be found in [584].

6.4.3.1 The design

Key Generation. ECIES-KEM, like ECIES, is an elliptic curve scheme. This means that before the scheme can be implemented a suitably secure elliptic curve E must have been generated and a point $P \in E$ with prime order p must have been chosen. We assume that the length of p is equal to the security parameter λ. The key generation algorithm is a probabilistic algorithm that takes the elliptic curve parameters (E, P, p, λ) as input and runs as follows.

1. Randomly generate an integer $s \in \{1, \ldots, p-1\}$.
2. Set $W := sP$.
3. Set $pk := (E, P, p, W, \lambda)$ and $sk := (s, pk)$.
4. Output the key-pair (pk, sk).

Encapsulation Algorithm. The encapsulation algorithm is a probabilistic algorithm that takes the public-key as input. It uses a public and pre-agreed key derivation function $KDF(\cdot)$ that must be available to all parties wishing to use the scheme. Its encapsulation algorithm runs as follows.

1. Generate a random integer $r \in \{1, \ldots, p-1\}$.

2. Set $C := rP$.
3. Set x to be the x-coordinate of rW.
4. Set $K := KDF(C||x)$.
5. Output the encapsulated key-pair (K, C).

Decapsulation Algorithm. The decapsulation algorithm is a deterministic algorithm that takes as input an encapsulated key C and the secret-key sk. It also uses the pre-agreed key derivation function $KDF(\cdot)$ that was used in the encapsulation process. The algorithm runs as follows. (For notational purposes, let $\mathcal{O}$ be the 'point at infinity' – the identity element of an elliptic curve group.)

1. Set $Q := sC$.
2. Check that $Q \neq \mathcal{O}$. If so, output `Invalid Ciphertext` and halt.
3. Set x to be the x-coordinate of Q.
4. Set $K := KDF(C||x)$.
5. Output K.

6.4.3.2 Security analysis

It is very easy to show that ECIES-KEM is IND-CCA2 secure in the random oracle model [195]. The security of the scheme can be very efficiently reduced to the problem of solving the gap Diffie-Hellman problem in the elliptic curve subgroup generated by P. Formally, if there exists a (t, ϵ, q_D) attacker for ECIES-KEM then there exists a (t', ϵ') solver for the gap Diffie-Hellman problem in the group $\langle P \rangle$ with

$$\epsilon' \approx \epsilon , \tag{6.16}$$

$$t' \approx t + 2q_K T , \tag{6.17}$$

where

– q_K is the number of queries that the attacker submits to the random oracle simulating the key derivation function,
– and T is the time taken to check a Diffie-Hellman triple (i.e. the time taken for the oracle to check whether $g^c = g^{ab}$ for a triple (g^a, g^b, g^c)).

In the IND-CPA model, the security of ECIES-KEM can be improved. The security of the scheme, in the random oracle model, can be reduced to the problem of breaking the computational Diffie-Hellman (CDH) problem in the group $\langle P \rangle$. Formally, if there exists a (t, ϵ) attacker for ECIES-KEM then there exists an algorithm running in time t' that outputs a list of L elements and contains a solution to the CDH problem with probability at least ϵ' with

$$\epsilon' \approx \epsilon , \tag{6.18}$$

$$t' \approx t , \tag{6.19}$$

$$L \leq q_K , \tag{6.20}$$

where the attacker makes at most q_K queries of the random oracle simulating the key derivation function. Of course, we may now use the techniques of Sect. 6.2.3.2 to construct a (t'', ϵ'') solver for the CDH problem with

$$\epsilon'' \approx 1 - \frac{1}{2^k}, \tag{6.21}$$

$$t'' \approx 2k\lceil 1/\epsilon' \rceil t' + 2kL\lceil 1/\epsilon' \rceil T, \tag{6.22}$$

where T is the time taken to compute a group element of the form

$$x_1^{-1}y_1^{-1}\{P' - ax_1y_2P - bx_2y_1P + x_2y_2P\},$$

for random integers x_1, y_1, x_2, y_2 and a random elliptic curve point P'.

ECIES-KEM is vulnerable to the small subgroup attacks of Lee and Lim (see Sect. 6.2.6). If the elliptic curve points are not validated then an attacker can compute the secret key s modulo n with one trial decapsulation and n evaluation of the key derivation function.

The ECIES-KEM mechanism can be viewed as a subroutine of both the PSEC-KEM (see Sect. 6.4.5) and the ACE-KEM algorithms (see Sect. 6.4.1). This means that the performance costs of PSEC-KEM and ACE-KEM are equal to or greater to than those of ECIES-KEM.

The only side-channel attacks to which ECIES-KEM seems to be vulnerable are a fault attack [197] and a power attack based on its use of elliptic curve groups (see Sect. A.1.2.3).

6.4.4 EPOC-2

EPOC-2 is a hybrid asymmetric encryption scheme submitted by NTT Corporation of Japan. It consists loosely of the following algorithms. A complete specification is given in [485].

6.4.4.1 The design

Key Generation. The key generation algorithm is a probabilistic algorithm that takes a security parameter λ as input and runs as follows.

1. Generate (usually randomly) two distinct λ-bit primes p and q. Set $n := p^2q$.
2. Generate (usually randomly) an element $g \in (\mathbb{Z}/n\mathbb{Z})^*$ such that $g_p := g^{p-1} \bmod p^2$ has order p in $(\mathbb{Z}/p^2\mathbb{Z})^*$.
3. Set $h := g^n \bmod n$.
4. Set $w := (g_p - 1)/p \bmod p$.
5. Set $pk := (n, g, h, \lambda)$ and $sk := (p, q, w, pk)$.
6. Output the key-pair (pk, sk).

Encryption Algorithm. The encryption algorithm takes as input the public-key pk and a message m. It also relies on a set of pre-agreed system parameters. In order for the scheme to work, the two parties must have agreed on the use of a hash function $Hash(\cdot)$, a mask generating function $MGF(\cdot)$, a key derivation function $KDF(\cdot)$ and a symmetric encryption scheme (*Sym.Encrypt*, *Sym.Decrypt*). It runs as follows.

1. Generate a random octet string $R \in \{0, \ldots, 255\}^{\lfloor(\lambda-1)/8\rfloor}$.
2. Derive a suitable symmetric key $K := KDF(R)$.

3. Encrypt the message m using the symmetric scheme and the key K, $C_2 := Sym.Encrypt(m, K)$.
4. Set $DB := m||R||C_2$.
5. Set $H := MGF(Hash(DB))$.
6. Set $C_1 := g^R h^H \bmod n$.
7. Output the ciphertext $C = (C_1, C_2)$.

This process is also shown pictorially in Fig. 6.20.

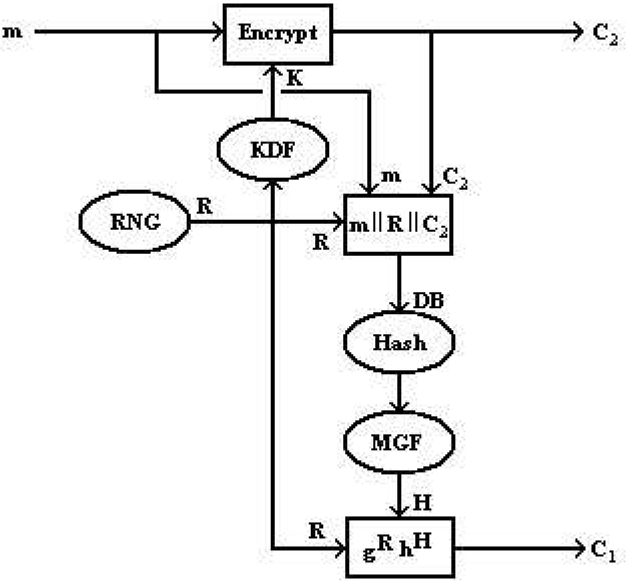


Fig. 6.20. The EPOC-2 encryption algorithm.

Decryption Algorithm. The decryption algorithm uses the same set of system parameters as the encryption algorithm, i.e. a pre-agreed hash function $Hash(\cdot)$, mask generating function $MGF(\cdot)$, key derivation function $KDF(\cdot)$ and symmetric encryption scheme ($Sym.Encrypt$, $Sym.Decrypt$). It takes as input a ciphertext C and the secret-key sk and runs as follows.

1. Parse the ciphertext C into (C_1, C_2).
2. Check that $0 \leq C_1 < n$. If not, output `Invalid Ciphertext` and halt.
3. Set $C_1' := C_1^{p-1} \bmod p^2$.
4. Set $C_1'' := (C_1' - 1)/p \bmod p$.
5. Set $R := C_1''/w \bmod p$.
6. Check that $0 \leq R < 256^{\lfloor(\lambda-1)/8\rfloor}$. If not, output `Invalid Ciphertext` and halt.
7. Derive a suitable symmetric key $K := KDF(R)$.

8. Decrypt the ciphertext C_2 using the symmetric encryption scheme and the key K, $m := Sym.Decrypt(C_2, K)$.
9. Set $DB := m||R||C_2$.
10. Set $H := MGF(Hash(DB))$.
11. Derive a reduced public key $\hat{PK}$ with $\hat{g} := g \bmod q$, $\hat{h} := h \bmod q$ and $\hat{H} := H \bmod q-1$.
12. Calculate $\hat{C_1} = \hat{g}^R \hat{h}^{\hat{H}} \bmod q$.
13. Check that $\hat{C_1} = C_1 \bmod q$. If not, output `Invalid Ciphertext` and halt.
14. Output m.

6.4.4.2 Security analysis

The following is a summary of the security analysis of EPOC-2; for more details see [192, 621]. Some arguments as to why EPOC-2 was selected to be studied in NESSIE phase II over the other EPOC candidates (see Sect. 6.5.1 and Sect. 6.5.2) can be found in [576].

It should be noted that the structure of EPOC-2 is very similar to that of a KEM-DEM based cryptosystem. It has, presumably, not been phrased in these terms because the 'KEM' output would not be indistinguishable from a random key in the CCA2 model. This is a good example of a hybrid scheme that is secure but does not satisfy the security requirements for a KEM-DEM based scheme. It should also be noted that the 'DEM' section uses group operations derived from the public key as a MAC tag, and this is formally excluded from a DEM construction since the DEM can have no access to the public key. In the security proof the symmetric cipher is assumed to be a Vernam cipher, in which the key is bitwise XORed with the message to form the ciphertext.

The security of EPOC-2 is based on the Okamoto-Uchiyama cryptosystem [501], which is provably as secure as factoring in the IND-CPA model, and an improved version of the Fujisaki-Okamoto transform [242] which is specific to the Okamoto-Uchiyama cryptosystem. The security proof [237] for the scheme proves that, in the random oracle model, one can reduce the problem of attacking the scheme in the IND-CCA2 setting to the problem of factoring the modulus $n = p^2 q$. For convenience we define $|n|$ to be the size of the integer n in bits. Formally, if there exists a (t, ϵ, q_D) IND-CCA2 attacker for EPOC-2 in the random oracle model then there exists a (t', ϵ') solver for the problem of factoring the modulus n with

$$\epsilon' \approx \frac{\epsilon}{3}(1 - 2^{-3\lambda+3})(1 - 2^{-\gamma})^{q_D} , \tag{6.23}$$

$$t' \approx t + q_H T_1 + q_H q_D T_2 , \tag{6.24}$$

where

- q_H is the number of queries the attacker makes of the random oracle,
- γ is a constant that depends only upon the public key,
- T_1 is the time taken to compute the greatest common divisor of two $|n|$-bit numbers,
- and T_2 is the time taken to check an equation of the form $C_1 = \hat{g}^R \hat{h}^{\hat{H}} \bmod q$.

The problem of factoring a number of the form $n = p^2q$ is one of the less well researched trusted cryptographic problems used by the NESSIE phase II submissions. Most of the research into factoring algorithms concentrates on attempting to factor integers of the form pq. The attack of [108], whilst not directly applicable to the Okamoto-Uchiyama modulus n, only serves to underline the fact that factoring a number of the form p^2q is unlikely to be harder than factoring a number of the form pq. Furthermore, factoring the modulus is enough to recover the secret key (which depends only on p and q). So any attack that can reliably distinguish messages in the CCA2 model also provides, in the random oracle model, an attack that can recover the secret key.

EPOC-2 is also vulnerable to a few important side-channel attacks (see Sect. 6.2.7). Whilst it is true that EPOC-2 is the only primitive in Phase II that does not seem susceptible to fault attacks [197], it is vulnerable to a Hamming weight attack [197] and to an error message attack [196]. This seems to be indicative of an inherent weakness of EPOC-2; the security proof for the primitive shows that if any information can be gained about the value derived from the Okamoto-Uchiyama decryption process (the number R calculated in step 5 of the decryption process) then an attacker is likely to be able use this information to launch a key recovery attack. For this reason we consider EPOC-2 to be weak against side-channel attacks.

For fair comparison EPOC-2 was not only evaluated against the other NESSIE candidates but also against existing schemes with similar security assumptions [209]. The scheme was compared against the HIME(R) scheme [484] and the Rabin-SAEP scheme [103], the security of each of which can be reduced to the problem of factoring a modulus n of the form $n = pq$ or $n = p^rq$ for small r. Whilst all of the schemes had comparable security reductions, EPOC-2 was found to have a comparatively large key size and a slow running time.

6.4.5 PSEC-KEM

PSEC-KEM is a tweaked version of PSEC-2 and was submitted by the NTT Corporation of Japan. It consists loosely of the following algorithms. A complete specification is available in [486].

6.4.5.1 The design

Key Generation. Since PSEC-KEM is defined over an elliptic curve, a suitable curve E will have to be generated before the key generation algorithm is executed. The curve should have a point P that generates a secure cyclic subgroup of E with prime order p. We assume that p is of size λ where λ is the security parameter.

The key generation algorithm is a probabilistic algorithm that takes the elliptic curve E, the point P and the order p of P as input. It runs as follows.

1. Generate an integer s uniformly at random from $\{0, \dots, p-1\}$.
2. Set $W := sP$.
3. Set $pk := (E, P, W, p, \lambda)$ and $sk := (s, pk)$.
4. Output the key-pair (pk, sk).

Encapsulation Algorithm. The key encapsulation algorithm is a probabilistic algorithm that takes as input the public-key pk. In order for the scheme to work the two communicating parties must have agreed on the use of a common key derivation function $KDF(\cdot)$. It runs as follows.

1. Generate a suitably sized random bit string r (of size comparable to p, say).
2. Set $H := KDF(0_{32}||r)$, where 0_{32} is the 32-bit representation of the integer 0.
3. Parse H as $t||K$ where t is a $(\lambda + 128)$-bit integer and K is a suitably sized symmetric key.
4. Set $\alpha := t \bmod p$.
5. Set $Q := \alpha W$.
6. Set $C_1 := \alpha P$.
7. Set $C_2 := r \oplus KDF(1_{32}||C_1||Q)$, where 1_{32} is the 32-bit representation of the integer 1.
8. Set $C := (C_1, C_2)$.
9. Output the encapsulated key-pair (K, C).

The PSEC-KEM encapsulation algorithm is also shown pictorially in Fig. 6.21.

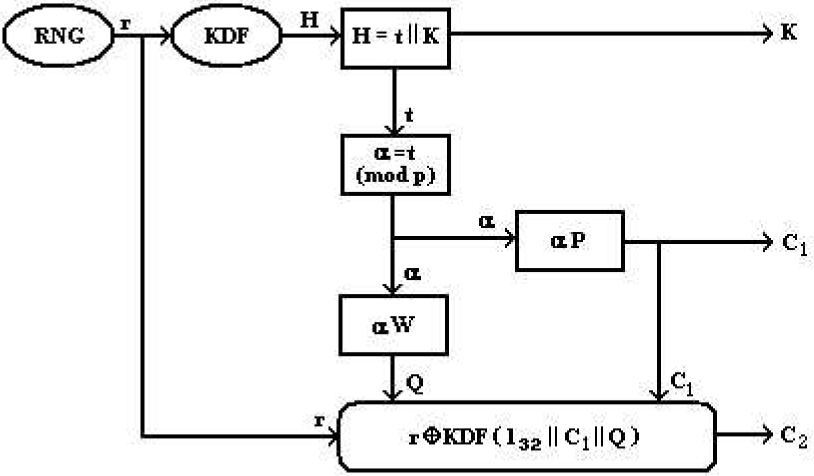


Fig. 6.21. The PSEC-KEM encapsulation algorithm.

Decapsulation Algorithm. The decapsulation algorithm is a deterministic algorithm that takes as input a key encapsulation C and the secret-key sk. It also uses the pre-agreed key derivation function $KDF(\cdot)$. It runs as follows.

1. Parse C as (C_1, C_2).
2. Set $Q := sC_1$.

3. Set $r := C_2 \oplus KDF(1_{32}||C_1||Q)$, where 1_{32} is the 32-bit representation of the integer 1.
4. Set $H := KDF(0_{32}||r)$, where 0_{32} is the 32-bit representation of the integer 0.
5. Parse H as $t||K$, where t is an $(\lambda + 128)$-bit integer and K is a suitably sized symmetric key.
6. Set $\alpha := t \bmod p$.
7. Check $C_1 = \alpha P$. If not, output `Invalid Ciphertext` and halt.
8. Output K.

6.4.5.2 Security analysis

In many ways the PSEC family of asymmetric encryption schemes are counterparts to the EPOC schemes, in the sense that they both use the same constructions to turn a base problem into useable asymmetric encryption schemes. However, when the schemes were tweaked at the start of NESSIE phase II, PSEC-2 was rephrased as a KEM-DEM based cryptosystem (see Sect. 6.3), whereas EPOC-2 (see Sect. 6.4.4) was not. The following security analysis is a summary of the work found mainly in [574, 575]. Some arguments as to why PSEC-2 was chosen to be studied in NESSIE phase II over the other PSEC candidates (see Sect. 6.5.3 and Sect. 6.5.4) were given in [576].

The security proof for PSEC-KEM can be found in [584]. It shows that, in the random oracle model, the problem of breaking the system can be reduced to the problem of solving a computational Diffie-Hellman problem on the elliptic curve E. Formally, if there exists a (t, ϵ, q_D) IND-CCA2 attacker for PSEC-KEM then there exists an algorithm running in time t' that outputs a list of L elements that contains the solution to the CDH problem with probability ϵ' and

$$\begin{aligned} t' &\approx t, && (6.25)\\ \epsilon' &\approx \epsilon - \frac{q_{K_0} + 2q_D}{p} - \frac{q_{K_0} + q_D}{2^\lambda}, && (6.26)\\ L &\leq q_D + q_{K_1}, && (6.27) \end{aligned}$$

where

- the encryption algorithm generates a random integer of length λ in step 1,
- the attacker makes at most q_{K_i} queries to the random oracle representing the key-derivation function where the initial 32-bits are a representation of the integer i.

Of course we may now use the probability amplification techniques given in Sect. 6.2.3.2 to give a (t'', ϵ'') solver for the CDH problem on E with

$$\begin{aligned} \epsilon'' &\approx 1 - \frac{1}{2^k}, && (6.28)\\ t'' &\approx 2k \lceil 1/\epsilon' \rceil t' + 2kL \lceil 1/\epsilon' \rceil T, && (6.29) \end{aligned}$$

where T is the time taken to compute a group element of the form

$$x_1^{-1}y_1^{-1}\{P' - ax_1y_2P - bx_2y_1P + x_2y_2P\},$$

for random integers x_1, y_1, x_2, y_2 and a random elliptic curve point P'.

Since the reduction shown for PSEC-KEM appears to be close to optimal in a group for which a DDH oracle does not exist, we do not expect the security reduction to be significantly improved in the IND-CPA model.

PSEC-KEM is an authenticated KEM. The PSEC-KEM algorithm can be viewed as using a version of ECIES-KEM (see Sect. 6.4.3) as a mask for the key. It also produces data that is used to perform a consistency check.

It is unclear whether PSEC-KEM is vulnerable to the small subgroup attacks of Lee and Lim (see Sect. 6.2.6) because of the Diffie-Hellman check in the decapsulation algorithm.

Along with most of the other NESSIE phase II candidates, PSEC-KEM is vulnerable to a fault attack and, along with the other schemes that are based on elliptic curves, it does seem to be vulnerable to power analysis (see Sect. A.1.2.3). It does not appear to be vulnerable to either an error message attack or a Hamming weight attack [197].

6.4.6 RSA-KEM

The RSA based key encapsulation mechanism is one of the simplest key encapsulation mechanisms. It uses the RSA trapdoor permutation as a security mechanism (see Sect. 6.3.2). Although RSA-KEM was never formally submitted to the NESSIE project, it is included in the evaluation process as a *de facto* standard for a KEM-DEM based cryptosystem and because it has been proposed for inclusion in the ISO/IEC standard 18033-2 [584]. It consists loosely of the following algorithms. A complete specification can be found in [325, 584].

6.4.6.1 The design

Key Generation. The key generation algorithm for RSA-KEM is very similar to that of RSA-OAEP (see Sect. 6.5.5) as both concentrate on producing a valid RSA key. The RSA-KEM key generation algorithm is a probabilistic algorithm that takes the security parameter λ as input and runs as follows.

1. Generate a random public exponent e, an odd integer greater than 1.
2. Randomly generate a prime p of length λ such that $gcd(p-1, e) = 1$.
3. Randomly generate a prime q of length λ such that $gcd(q-1, e) = 1$.
4. Set $n := pq$.
5. Set d to be the unique integer in $\mathbb{Z}/\lambda(n)\mathbb{Z}$ such that $ed \equiv 1 \bmod \lambda(n)$, where $\lambda(n) = l.c.m.(p-1, q-1)$.
6. Set $pk := (n, e, \lambda)$ and $sk := (d, pk)$.
7. Output the key-pair (pk, sk).

Note that there have been many other methods proposed to generate RSA key-pairs. The most notable of these involves choosing a suitable exponent e in step 1 rather than randomly generating it.

Encapsulation Algorithm. The encapsulation algorithm is a probabilistic algorithm that takes as input the public key *pk*. It also uses a common, public key derivation function $KDF(\cdot)$ that is available to all parties. It runs as follows.

1. Generate a random integer $r \in \{0, \ldots, n-1\}$.
2. Set $C := r^e \bmod n$.
3. Set $K := KDF(r)$.
4. Output the encapsulated key-pair (K, C).

Decapsulation Algorithm. The decapsulation algorithm is a deterministic algorithm that takes an encapsulation C and the secret-key *sk* as input. It also uses the pre-agreed key derivation function $KDF(\cdot)$ and runs as follows.

1. Set $r := C^d \bmod n$.
2. Set $K := KDF(r)$.
3. Output K.

6.4.6.2 Security analysis

The following is a summary of the security analysis for RSA-KEM. The results can either can be found in [275] or derived from the general results on KEM-DEM cryptosystems in [193, 195].

The security of RSA-KEM is based on the security of the RSA cryptosystem [543] in the OW-CPA model. We use the general result about security mechanisms of [193] to show that, in the random oracle model, the security of RSA-KEM can be reduced to the RSA problem. Formally, if there exists a (t, ϵ, q_D) attacker for RSA-KEM in the IND-CCA2 sense then there exists a (t', ϵ') solver for the RSA problem with

$$\begin{aligned} \epsilon' &\approx \epsilon\,, && (6.30) \\ t' &\approx t + (q_K + q_D)T\,, && (6.31) \end{aligned}$$

where

– q_K is the number of queries the attacker makes to the random oracle,
– and T is the time taken to calculate x^e for some x.

Since the reduction shown for RSA-KEM appears to be close to optimal, we do not expect the security reduction to be significantly improved in the IND-CPA model.

In the case where the key generation algorithm does not randomly select the public exponent e but instead selects some specific exponent, the security proofs still hold but the security of the scheme reduces to the e-th root problem rather than the RSA problem (see Sect. 6.2.3.1).

The RSA cryptosystem exhibits some homomorphic properties, most noticeably that if $C_1 = m_1^e \bmod n$ and $C_2 = m_2^e \bmod n$ then $C_1 C_2 \bmod n$ is the encryption of $m_1 m_2 \bmod n$. Whilst this is highly desirable in certain applications, it does allow unknown messages to be manipulated in quite specific ways. In RSA-KEM we rely on the nature of the key derivation function $KDF(\cdot)$ to destroy any relations between keys that result from relations in encapsulations. Hence the properties of the key derivation function are more critical in RSA-KEM than they might be in, say, ACE-KEM (see Sect. 6.4.1). This homomorphic property is also useful when implementing side-channel attacks. It has been shown that RSA-KEM is vulnerable to a fault attack that recovers the secret key [349], a chosen modulus attack [207, 349] and a Hamming weight attack [197, 349]. See Sect. 6.2.7 for a discussion of the relevance of these attacks.

There is also an attack against the RSA cryptosystem that can be applied to RSA-KEM. It has been shown that if the secret exponent d is less than $n^{0.292}$ then it can be recovered from the modulus n and public exponent e [106]. It has been conjectured that it will be possible to recover d from n and e providing $d < n^{0.5}$.

RSA-KEM was selected as a suitable *de facto* standard for hybrid RSA based cryptosystems after it was favourably evaluated against RSA-REACT [499] in a paper by Granboulan [275].

6.5 Asymmetric encryption primitives not selected for Phase II

The following algorithms were submitted to NESSIE but not selected for further study in the second phase of the project:

- EPOC-1
- EPOC-3
- PSEC-1
- PSEC-3
- RSA-OAEP

Again we choose to specify the algorithms in a general mathematical way rather than as a detailed specification. In particular we assume that variables are stored in binary form even if they are integers, elliptic curve points, etc. We also assume that hash functions, mask generating functions, key derivation functions and symmetric encryption schemes all take inputs and produce outputs of a "correct" length for the asymmetric scheme. References are given to complete specifications for the algorithms.

6.5.1 EPOC-1

EPOC-1 was the first of the EPOC series submitted by the NTT Corporation. It is based on the Okamoto-Uchiyama problem [501], which is provably secure in the IND-CPA sense in the random oracle model. The techniques of [241] are then used to transform this into an asymmetric encryption scheme that is IND-CCA2 secure in the random oracle model. The scheme consists loosely of the following algorithms. A complete specification can be found in [239].

6.5.1.1 The design

Key Generation. The key generation algorithm is a probabilistic algorithm that takes a security parameter λ as input and runs as follows.

1. Randomly generate two λ-bit primes p and q. Set $n := p^2 q$.
2. Randomly generate an element $g \in (\mathbb{Z}/n\mathbb{Z})^*$ such that $g_p := g^{p-1} \bmod p^2$ has order p in $(\mathbb{Z}/p^2\mathbb{Z})^*$.
3. Randomly generate an element $h_0 \in (\mathbb{Z}/n\mathbb{Z})^*$ and set $h := h_0^n \bmod n$.
4. Set $w := \frac{g_p - 1}{p} \bmod p$.
5. Choose positive integers $mLen$ and $rLen$ such that $mLen + rLen \leq \lambda - 1$.
6. Set $pk := (n, g, h, mLen, rLen, \lambda)$ and $sk := (p, q, w, pk)$.
7. Output the key-pair (pk, sk).

Encryption Algorithm. The encryption algorithm takes as input the public-key pk and a message m of length $mLen$. It also uses a pre-agreed hash function $Hash(\cdot)$. It runs as follows.

1. Generate a random string R of length $rLen$ and compute $r := Hash(m||R)$.
2. Set $C := g^{m||R} h^r \bmod n$.
3. Output the ciphertext C.

Decryption Algorithm. The decryption algorithm takes as input a ciphertext C and the secret-key sk. It also uses the pre-agreed hash function $Hash(\cdot)$ and runs as follows.

1. Set $C_p := C^{p-1} \bmod p^2$.
2. Set $C'_p := \frac{C_p - 1}{p} \bmod p$.
3. Set $X := \frac{C'_p}{w} \bmod p$.
4. Check that $0 \leq X \leq 2^{mLen+rLen}$. If not, output `Invalid Ciphertext` and halt.
5. Check that $C = g^X h^{Hash(X)}$. If not, output `Invalid Ciphertext` and halt.
6. Set m to be the first $mLen$ bits of X.
7. Output the message m.

6.5.1.2 Security analysis

The following is a summary of the security analysis of EPOC-1 ; for more details see [621]. Some of the arguments as to why EPOC-2 (see Sect. 6.4.4) was selected to be studied in NESSIE phase II over EPOC-1 were given in [576].

It has been shown that, in the random oracle model, EPOC-1 is secure in the IND-CCA2 sense [239]. The security of the scheme reduces to the problem of solving the p-subgroup membership problem on the group $(\mathbb{Z}/n\mathbb{Z})^*$ (see Sect. 6.2.3). Formally, if (t, ϵ, q_D) is an IND-CCA2 attacker for the EPOC-1 scheme then there exists a (t', ϵ') solver for the p-subgroup problem with

$$\epsilon' \approx (\epsilon - q_H 2^{-(rLen-1)})(1 - 2^{-2\lambda})^{q_D} \, , \tag{6.32}$$

$$t' \approx t + q_H(T + c\lambda) \, , \tag{6.33}$$

where

– q_H is the number of queries the attacker makes to the random oracle,
– T is the time taken to calculate $g^m h^r \bmod n$,
– and c is a constant.

EPOC-1 was designed for key distribution, which is outside the scope of the NESSIE project. The scheme was not selected for NESSIE phase II because it had a worse security reduction than EPOC-2 for similar performance costs. Furthermore, the submitters withdrew their support for EPOC-1 in favour of EPOC-2 (see Sect. 6.4.4).

6.5.2 EPOC-3

EPOC-3 was submitted by the NTT Corporation and, like the other members of the EPOC series, uses the Okamoto-Uchiyama cryptosystem [501]. This time the submitters use the REACT transform [499] to improve the security of the basic scheme. The EPOC-3 scheme consists loosely of the following three algorithms. A complete specification is given in [239].

6.5.2.1 The design

Key Generation. The key generation algorithm is a probabilistic algorithm that takes a security parameter λ as input and runs as follows.

1. Randomly generate two λ-bit primes p and q. Set $n := p^2 q$.
2. Randomly generate an element $g \in (\mathbb{Z}/n\mathbb{Z})^*$ such that $g_p := g^{p-1} \bmod p^2$ has order p in $(\mathbb{Z}/p^2\mathbb{Z})^*$.
3. Randomly generate an element $h_0 \in (\mathbb{Z}/n\mathbb{Z})^*$ and set $h := h_0^n \bmod n$.
4. Set $w := \frac{g_p - 1}{p} \bmod p$.
5. Set $pk := (n, g, h, \lambda)$ and $sk := (p, q, w, pk)$.
6. Output the key-pair (pk, sk).

Encryption Algorithm. The encryption algorithm takes as input the public-key pk and a message m. It also relies on a set of pre-agreed system parameters. In order for the scheme to work the two communicating parties must have agreed upon a public choice of a hash function $Hash(\cdot)$, a key derivation function $KDF(\cdot)$ and a symmetric encryption scheme ($Sym.Encrypt$, $Sym.Decrypt$). The encryption algorithm runs as follows.

1. Generate two independent and uniform random bit strings r and R of length $\lambda - 1$.
2. Derive a suitable symmetric key $K := KDF(R)$.
3. Set $C_1 := g^R h^r \bmod n$.
4. Set $C_2 := Sym.Encrypt(m, K)$, i.e. the encryption of the message m in the symmetric scheme under the key K.
5. Set $C_3 := Hash(C_1 || C_2 || R || m)$.
6. Output the ciphertext $C = (C_1, C_2, C_3)$.

Decryption Algorithm. The decryption algorithm uses the same set of system parameters as the encryption algorithm: a hash function $Hash(\cdot)$, a key derivation function $KDF(\cdot)$ and a symmetric encryption scheme (*Sym.Encrypt*, *Sym.Decrypt*). It takes as input a ciphertext C and the secret key sk. It runs as follows.

1. Parse C as an appropriately sized triple (C_1, C_2, C_3).
2. Set $C_1' := C_1^p \bmod p^2$.
3. Set $C_1'' := \frac{C_1'-1}{p} \bmod p$.
4. Set $R := \frac{C_1''}{w} \bmod p$.
5. Derive a suitable symmetric key $K := KDF(R)$.
6. Decrypt the ciphertext C_2 using the symmetric encryption scheme and the key K, i.e. set $m := Sym.Decrypt(C_2, K)$.
7. Check if $R < 2^{\lambda-1}$. If not, output `Invalid Ciphertext` and halt.
8. Check that $C_3 = Hash(C_1||C_2||R||m)$. If not, output `Invalid Ciphertext` and halt.
9. Output the message m.

6.5.2.2 Security analysis

The following is a summary of the security analysis of EPOC-3; for more details see [621]. Some of the arguments as to why EPOC-2 (see Sect. 6.4.4) was selected to be studied in NESSIE phase II over EPOC-3 were given in [576].

It has been shown in [239] that EPOC-3 is IND-CCA2 secure in the random oracle model, and reduces to the problem of solving the gap-factoring problem for the modulus n. As with EPOC-2 (see Sect. 6.4.4) we will assume that the symmetric encryption system used is a Vernam cipher. Formally, if there exists a (t, ϵ, q_D) attacker for EPOC-3 then there exists a (t', ϵ') solver for the gap-factoring problem (see Sect. 6.2.3) with

$$\epsilon' \approx \frac{1}{2}\epsilon - \frac{q_D}{2^\lambda}, \tag{6.34}$$

$$t' \approx t + c(q_H + q_K), \tag{6.35}$$

where

– c is a constant,
– q_H is the number of queries the attacker makes to the hash function random oracle,
– and q_K is the number of queries the attacker makes to the key derivation function random oracle.

The reduction of EPOC-3 is not as efficient as that of EPOC-2, and the underlying assumption is not as strong. Furthermore, the submitters withdrew their support from the scheme in favour of EPOC-2.

6.5.3 PSEC-1

PSEC-1 is the first of the PSEC family of cryptosystems that were submitted to NESSIE by the NTT Corporation. It consists loosely of the following algorithms. A complete specification can be found in [238].

6.5.3.1 The design

Key Generation. PSEC-1 is defined over an elliptic curve, so, before any of these algorithms are executed, it is necessary to have constructed a suitably secure elliptic curve E and chosen a point $P \in E$ with prime order p. We assume that the length of p is equal to the security parameter λ and that the elliptic curve is defined over a finite field F_q, where q is a prime power of length $qLen$. The key generation algorithm is a probabilistic algorithm that takes $(E, P, p, qLen, \lambda)$ as input. It runs as follows.

1. Generate a random integer $s \in (\mathbb{Z}/p\mathbb{Z})^*$.
2. Set $W := sP$.
3. Choose positive integers $mLen$ and $rLen$ such that $mLen + rLen = qLen$.
4. Set $pk := (E, P, p, qLen, W, mLen, rLen, \lambda)$ and $pk := (s, pk)$.
5. Output the key-pair (pk, sk).

Encryption Algorithm. The encryption algorithm is a probabilistic algorithm that takes as input a message m of length $mLen$ and the public-key pk. Before the encryption algorithm can be run the communicating parties must have agreed upon the use of some common, public hash function $Hash(\cdot)$. The encryption algorithm runs as follows.

1. Generate a random bit string r of length $rLen$.
2. Set $\alpha := Hash(m||r)$.
3. Set $C_1 := \alpha P$.
4. Set x to be the x-coordinate of αW.
5. Set $C_2 := (m||r) \oplus x$.
6. Output the ciphertext $C = (C_1, C_2)$.

Decryption Algorithm. The decryption algorithm is a deterministic algorithm that takes as input a ciphertext C and the private-key sk. It also uses the agreed hash function $Hash(\cdot)$. It runs as follows.

1. Parse the ciphertext C into (C_1, C_2).
2. Set x to be the x-coordinate of sC_1.
3. Set m and r to be the appropriately sized bit strings that satisfy $(m||r) = C_2 \oplus x$.
4. Set $\alpha := Hash(m||r)$.
5. Check that $C_1 = \alpha P$. If not, output `Invalid Ciphertext` and halt.
6. Output m.

6.5.3.2 Security analysis

The following is a summary of the security analysis of PSEC-1; for more details see [574]. Some of the arguments that justify the selection of PSEC-2 for further study in NESSIE phase II over PSEC-1 can be found in [576].

The security claims of [238] refer to a security proof in [241]. Here it is loosely shown that PSEC-1 is IND-CCA2 secure in the random oracle model provided that the decisional Diffie-Hellman problem (see Sect. 6.2.3) is intractable on the elliptic curve group generated by P. Formally, if there exists a (t, ϵ, q_D) attacker for PSEC-1 then there exists a (t', ϵ') solver for the decisional Diffie-Hellman problem on the elliptic curve group generated by P, with

$$\epsilon' \approx (\epsilon - q_H 2^{-rLen+1})(1-2p)^{q_D} , \tag{6.36}$$

$$t' \approx t + q_H(T + c\lambda) , \tag{6.37}$$

where

- the attacker makes at most q_H queries to the random oracle simulating the hash function,
- T is the time taken to calculate sQ on the elliptic curve,
- and c is a constant.

PSEC-1 was designed for key distribution, which is outside the scope of the NESSIE project. Hence PSEC-1 has a very limited message space. It shares a lot of properties with PSEC-KEM (see Sect. 6.4.5) but reduces to a weaker security assumption. Furthermore, the submitters withdrew their support from PSEC-1 in favour of PSEC-KEM.

6.5.4 PSEC-3

The last of the PSEC algorithms submitted by NTT Corporation, PSEC-3, is a hybrid encryption scheme based on the hardness of the gap Diffie-Hellman problem on certain elliptic curve groups. It consists loosely of the following algorithms. A complete specification can be found in [238].

6.5.4.1 The design

Key Generation. The key generation algorithm for PSEC-3 is similar to that of PSEC-KEM (see Sect. 6.4.5). Since PSEC-3 is based on the Diffie-Hellman problem on elliptic curves, it is necessary to have generated a suitable curve E and chosen a point $P \in E$ with prime order p. We will assume that the length of p is equal to the security parameter λ and that E is defined over a finite field $\mathbb{F}_q$, where q is a prime power of length $qLen$. The key generation algorithm is a probabilistic algorithm that takes $(E, P, p, qLen, \lambda)$ as input. It runs as follows.

1. Generate a random integer $s \in (\mathbb{Z}/p\mathbb{Z})^*$.
2. Set $W := sP$.
3. Set $pk := (E, P, p, qLen, W, \lambda)$ and $sk := (s, pk)$.
4. Output the key-pair (pk, sk).

Encryption Algorithm. The encryption algorithm is a probabilistic algorithm that takes a message m and the public-key pk as input. It is necessary for the two communicating parties to have agreed on the use of some common functions: a hash function $Hash(\cdot)$, a key derivation function $KDF(\cdot)$ and a symmetric encryption scheme ($Sym.Encrypt$, $Sym.Decrypt$). It runs as follows.

1. Generate a random integer $r \in (\mathbb{Z}/p\mathbb{Z})^*$.
2. Set $C_1 := rP$.
3. Set x to be equal to the x-coordinate of rW.
4. Generate a random bit string u of length $qLen$.
5. Set $C_2 = u \oplus x$.
6. Derive a suitable symmetric key $K := KDF(u)$.
7. Encrypt the message m using the agreed symmetric encryption scheme under the key K, i.e. set $C_3 = Sym.Encrypt(m, K)$.
8. Set $C_4 = Hash(C_1||C_2||C_3||u||m)$.
9. Output the ciphertext $C = (C_1, C_2, C_3, C_4)$.

Decryption Algorithm. The decryption algorithm is a deterministic algorithm that takes a ciphertext C and the secret-key sk as input. It also uses the pre-agreed hash function $Hash(\cdot)$, key derivation function $KDF(\cdot)$ and symmetric encryption scheme ($Sym.Encrypt$, $Sym.Decrypt$). It runs as follows.

1. Parse the ciphertext C as (C_1, C_2, C_3, C_4).
2. Set x to be the x-coordinate of sC_1.
3. Set $u := C_2 \oplus x$.
4. Derive a suitable symmetric key $K := KDF(u)$.
5. Decrypt the symmetric ciphertext C_3 using the key K, i.e. set $m := Sym.Decrypt(C_3, K)$.
6. Check that $C_4 = Hash(C_1||C_2||C_3||u||m)$. If not, output `Invalid Ciphertext` and halt.
7. Output m.

6.5.4.2 Security analysis

The following is a summary of the security analysis for PSEC-3; for more details see [574]. Some justification for the selection of PSEC-KEM (see Sect. 6.4.5) for further study over PSEC-3 is given in [576].

PSEC-3 is an amalgamation of two techniques used to enhance the security of a key distribution scheme similar to PSEC-1 (see Sect. 6.5.3). The scheme uses standard techniques to transform the weak key distribution scheme into a weak hybrid encryption scheme. It then uses the techniques of [241] to improve the security of the scheme.

The security analysis of PSEC-3 assumes that the implementation uses a Vernam cipher as its symmetric component. The specifications [238] claim that PSEC-3 is IND-CCA2 secure in the random oracle model and reduces to solving the gap Diffie-Hellman problem on the elliptic curve group generated by P. A formal proof of security was never given.

PSEC-KEM has an efficient reduction to a better assumption than PSEC-3, and PSEC-3 does not appear to be significantly faster than a hybrid scheme using PSEC-KEM. PSEC-3 is also very similar in structure to ECIES (see Sect. 6.4.2). Furthermore, the submitters withdrew their support for PSEC-3 in favour of PSEC-KEM.

6.5.5 RSA-OAEP

The RSA-OAEP algorithm is a well-established asymmetric encryption scheme that uses the OAEP padding scheme developed by Bellare and Rogaway [53]. It was submitted to NESSIE by RSA Laboratories and consists loosely of the following algorithms. A complete specification can be found in [325].

6.5.5.1 The design

Key Generation. The key generation algorithm for RSA-OAEP is similar to that of RSA-KEM (see Sect. 6.4.6). It takes the security parameter λ as input and runs as follows.

1. Generate a public exponent e, an odd integer greater than 1.
2. Randomly generate a prime p of length λ such that $gcd(p, e) = 1$.
3. Randomly generate a prime q of length λ such that $gcd(q, e) = 1$.
4. Set $n := pq$.
5. Set d to be the unique integer in $(\mathbb{Z}/\lambda(n)\mathbb{Z})^*$ such that $ed \equiv 1 \bmod \lambda(n)$, where $\lambda(n) = l.c.m.(p-1, q-1)$.
6. Choose positive integers $mLen$ and $rLen$ such that the length of n in bits is equal to $mLen + rLen + 32$.
7. Set $pk := (n, e, mLen, rLen, \lambda)$ and $sk := (d, pk)$.
8. Output the key-pair (pk, sk).

Encryption Algorithm. The encryption algorithm is a probabilistic algorithm that takes a message m of length $mLen$ and the public key pk as input. In order to use the scheme all parties must have agreed on the use of a common, public mask generating function $MGF(\cdot)$. Note that here we use the mask generating function to produce outputs of different lengths. We trust that the size of the output required from the mask generating function will be clear from the context.

The encryption algorithm runs as follows.

1. Generate a random bit string R of length $rLen$.
2. Set $X := MGF(R) \oplus (1_{16}||m)$, where 1_{16} is the 16-bit representation of the integer 1.
3. Set $Y := R \oplus MGF(X)$.
4. Set $Z := 0_{16}||Y||X$ where 0_{16} is the 16-bit representation of the integer 0.
5. Set $C = Z^e \bmod n$.
6. Output the ciphertext C.

Decryption Algorithm. The decryption algorithm is a deterministic algorithm that takes a ciphertext C and the secret-key *sk* as input. It requires access to the same mask generating function that is used in the encryption process, and runs as follows.

1. Set $Z := C^d \bmod n$.
2. Check that the leftmost 16 bits of Z are equal to 0. If not, output `Invalid Ciphertext` and halt.
3. Parse Z as $0_{16}||Y||X$, where Y has length $rLen$, X has length $mLen + 16$ and 0_{16} is the 16-bit representation of the integer 0.
4. Set $R := Y \oplus MGF(X)$.
5. Set $W := X \oplus MGF(R)$.
6. Check that the leftmost 16 bits of W are equal to the 16-bit representation of the integer 1. If not, output `Invalid Ciphertext` and halt.
7. Parse W as $1_{16}||m$, where m has length $mLen$ and 1_{16} is the 16-bit representation of the integer 1.
8. Output m.

6.5.5.2 Security analysis

The following is a summary of the security analysis of the RSA-OAEP asymmetric encryption scheme; for more details see [417].

The RSA cryptosystem [543] is well known not to be message-indistinguishable in any normal attack model; however it is thought to be OW-CPA secure in the standard model. Indeed its security has been so well studied that it is considered to be a trusted cryptographic problem in its own right and so needs no more justification as a cryptosystem. The OAEP padding method was introduced in [53] and provided with a proof that it transforms a scheme that is OW-CPA secure into a scheme that is IND-CCA2 secure in the random oracle model. Unfortunately this proof was shown to have a flaw [583]. The proof was corrected for the RSA cryptosystem in [243].

This proof reduces the security of the cryptosystem to the RSA problem. Formally, if (t, ϵ, q_D) is an attacker for RSA-OAEP in the IND-CCA2 model then there exists a (t', ϵ') solver for the RSA problem with

$$\epsilon' \geq \frac{\epsilon^2}{4} - \epsilon \cdot \left(\frac{2q_D q_M + q_D + q_M}{2^{rLen}} + \frac{2q_D}{2^{16}} + \frac{32}{4^{\lambda - rLen}} \right), \qquad (6.38)$$

$$t' \geq 2t + 3q_M^2 + O(\lambda^3), \qquad (6.39)$$

where

– q_M is the number of queries the attacker makes to the random oracle representing the mask generating function.

Note that this security bound is not tight.

There is an attack against the RSA cryptosystem that can be applied to RSA-OAEP. It has been shown that if the secret exponent d is less than $n^{0.292}$ then it can be recovered from the modulus n and public exponent e [106]. It has been conjectured that it will be possible to recover d from n and e providing $d < n^{0.5}$.

Although side-channel attacks were not considered during phase I, the wealth of literature on RSA based cryptosystems means that they are easy to find. It is vulnerable to an error-message attack [414] and to a Hamming weight attack [349]. Each of these attacks recover a hidden message.

At the end of phase I NESSIE invited RSA Laboratories to submit a tweaked version of the RSA cryptosystem with a padding method that gave rise to a more efficient security reduction (such as RSA-SAEP or RSA-SAEP+ [103]). RSA Laboratories responded by saying:

> The only known RSA-based encryption scheme with a better security reduction than RSA-OAEP is Victor Shoup's adaption RSA-OAEP+. However, the security reduction for RSA-OAEP+ is not tight enough either in practice. Since neither of the schemes are provably secure for concrete parameters, replacing one with the other does not appear meaningful.
>
> – J. Jönsson, RSA Laboratories

Hence RSA-OAEP was not selected for further study in NESSIE phase II because it offers no advantages over a hybrid scheme that uses RSA-KEM (see Sect. 6.4.6).

7. Digital signature schemes

7.1 Introduction

What is a digital signature? A signature is used by a signer to authenticate a document, in such a way that anyone can check the validity of the signature. A digital signature scheme does not mimic exactly the classical handwritten signatures, because the signature is different for each message. If this were not the case, a signature could be extracted from a document and copied to another document.

The digital information that allows the signer to generate valid digital signatures is the private key, and the digital information that allows the verifier to check the validity of the signature is the public key.

Components of a digital signature scheme. A signature scheme is described by the following four algorithms, with the security parameter k:

- a parameter generation $\mathsf{Generate} : (k, \rho) \mapsto \mathsf{param}$,
- a key generation $\mathsf{KeyGen} : (\mathsf{param}, \rho') \mapsto (\mathsf{pk}, \mathsf{sk})$,
- a signature generation $\mathsf{Sign} : (\mathsf{param}, \mathsf{sk}, m, r) \mapsto \sigma$,
- a signature verification $\mathsf{Ver} : (\mathsf{param}, \mathsf{pk}, \sigma, r') \mapsto m$ or reject.

All these algorithms are deterministic. pk is the public key, sk is the private key, m is a message and σ a signed message. The inputs ρ, ρ', r and r' (if non empty) contain the optional randomisation for the algorithms. Usually these are fixed length bit strings.

The digital signature scheme is sound if the following condition holds.

- For all k and m and for all ρ, ρ', r and r', let $\mathsf{param} = \mathsf{Generate}^{\rho}(k)$ and $(\mathsf{pk}, \mathsf{sk}) = \mathsf{KeyGen}^{\rho'}(\mathsf{param})$, and let $\sigma = \mathsf{Sign}^{r}_{\mathsf{param},\mathsf{sk}}(m)$. Then $\mathsf{Ver}^{r'}_{\mathsf{param},\mathsf{pk}}(\sigma) = m$.
  It may be accepted that the scheme is sound either with probability 1 or for all but a negligible proportion of ρ, ρ', r and r'.

The digital signature scheme is secure if it is hard to generate a valid σ without knowing sk. When studying the security of a digital signature scheme, the exact description of the algorithms $\mathsf{Generate}$, KeyGen and Sign is not needed. Only the probability distribution of their output makes a difference (see also the

[0] Coordinator for this chapter: ENS — Louis Granboulan

notion of *equivalent signature schemes* in Sect. 7.2.3). This is why we will begin the description of signature schemes by the description of their most specific component: Ver.

Appendix or message recovery. Signature schemes with appendix have the property that $\sigma = m\|s$ and s is called the appendix. Signature schemes with partial message recovery have the property that $\sigma = \hat{m}\|s$ and that the whole message is $m = \hat{m}\|\bar{m}$, where $\bar{m}$ is the recovered part of the message. They typically have a lower bound for the size of the whole message, which is also the number of message bits recovered. This lower bound can be overcome by storing the length of the actual recovered part in $\bar{m}$, e.g. by padding $\bar{m}$ with a 1 followed by a string of 0s. With this padding, one bit of message expansion is added. All the signature schemes submitted to NESSIE are signature schemes with appendix.

Stateful schemes. In a stateful digital signature scheme, the signing algorithm Sign is allowed to keep an internal state that tracks the number of signatures generated and optionally their values.

All the signature schemes submitted to NESSIE are stateless, but there exist interesting examples of stateful schemes [169, 212, 273].

Public key and identity. It is important that the verifier is convinced that the public key used to verify a signature corresponds to the alleged signer. Generally, it is not the goal of the signature scheme to decide how the public key infrastructure is organised, but any application that uses a signature scheme needs to decide how public keys are linked to meaningful identities.

Some digital signature schemes are identity-based, which means that the signer can choose his public key to be equal to his identity, and that a trusted authority generates the corresponding secret key. No identity-based signature scheme was submitted to NESSIE.

Description of the scheme, public parameters and public keys. Many descriptions of signature schemes only include the precise description for the algorithms Sign and Ver. However, to study the performance and security of the scheme one needs to know what is part of the scheme, what is a parameter that can be tuned to different applications, and what is in the keys specific to each user.

In many cases, parameters and keys are not clearly separated, and parameter generation and key generation are merged in a single algorithm KeyGen' : $(k, \rho\|\rho') \mapsto (\mathsf{param}\|\mathsf{pk}, \mathsf{param}\|\mathsf{sk})$, but separating the two is useful in practice.

The public parameters of a scheme usually include some information on the key size, but may include other additional information like the description of an elliptic curve subgroup. The choice of a hash function for the scheme may be fixed, but is usually left as a parameter, or may even be left up to the user (see also the discussion on hash identifiers [334]).

Parameters and key validation. Many descriptions of signature schemes only include the algorithms Sign and Ver. They certainly are the most specific components of a scheme, and their performance and security are usually studied assuming that the parameters and keys are uniformly randomly distributed. However, the way parameters and keys are actually generated can be the source of weaknesses and it is important that the verifier and the signer can trust the parameters and the keys. This document will not go deeply into this topic, but this issue will be highlighted when necessary.

Short to long term security. Some applications may need long term security (50 to 80 years) but current knowledge does not allow us to make predictions about cryptanalytic advances for such a long time. Some applications only need short term validity of a signature (1 day to 1 month). Medium term security is 5 to 10 years and is the main target of this document.

All signature schemes should incorporate in their public parameters a deadline for validity. If the validity needs to be extended beyond this deadline, re-signing with more recent public parameters is mandatory.

7.2 Security requirements

7.2.1 Security model

7.2.1.1 Existential unforgeability under adaptive chosen message attack

Unforgeability. An *existential forger under adaptive chosen message attack* is a (randomised) algorithm that inputs a public key and tries to produce a valid signature, the forgery. The forger is able to make queries to a black box that generates valid signatures and the forgery must be new (see below). A (t, ε, q_S)-forger succeeds in time t with probability ε and is allowed to make q_S signing queries. A signature scheme with no (t, ε, q_S)-forger is said to be (t, ε, q_S)-secure.

The original definition of an existential forger [273] required that the forgery is a valid signed message for a message that was not the input of a signature query. Our definition only requires that the forgery is a valid signed message that was not the answer to a query. This means that another valid appendix for the same message is a valid forgery. This requirement is called "super-security" by Goldreich [265, Volume 2, Sect. 6.5.2], "strong unforgeability" by Bellare and Namprempre (introduced for MACs [50]) or "non-malleability" by Stern *et al.* [597].

We will aim at non-malleable existential unforgeability under adaptive chosen message attack. Non-malleability may not be really important [14] but some applications may need it.

Some weaker security requirements. The adaptive chosen message signing oracle can be replaced by a less powerful oracle.

- Single-occurrence chosen message attacks (SO-CMA) [597]. The forger is not allowed to make multiple signing queries on the same message. This attack model appears during the study of non-deterministic signature schemes, especially ESIGN. See Sect. 7.3.1.2 and 7.4.2.

- Random message attacks (RAND). The forger has access to a list of signed messages that correspond to random messages. This attack model depends on a probability distribution on the message space. Schemes that are secure in this attack model can be used as building blocks for schemes that are secure against adaptive chosen message attacks. See Sect. 7.3.3.3.
  Goldwasser, Micali and Rivest [273] defined *known-message attacks* where the forger has access to a list of signed messages, but they don't say how these messages are chosen, only that the forger did not choose them.

Security level. The security level of a scheme is k bits if there exists no (t, ε, q_S)-forger with $\log_2(t/\varepsilon) < k$.

This value k depends on the time unit used for t. A natural time unit is the running time of the verification algorithm.

Non-repudiation of origin. In a similar way to the two flavours of unforgeability, two flavours of non-repudiation of origin exist.

Basic non-repudiation of origin means that any third party can be convinced that a valid signed document corresponds to a message that has been deliberately signed by the secret key holder. It makes it impossible to repudiate a message. Existential unforgeability under adaptive chosen message attack implies (basic) non-repudiation.

Strong non-repudiation of origin means that any third party can be convinced that a valid signed document has been deliberately signed by the secret key holder. It makes it impossible to repudiate a signed message. Non-malleable (strong) existential unforgeability under adaptive chosen message attack implies strong non-repudiation.

> REMARK: The verification algorithm needs to get the whole signed message σ to compute its answer. Therefore, for a signature scheme with appendix, it is meaningless to consider properties where the appendix is studied separately from the message.
> For example, the notion of duplicate signatures is meaningless. It was defined [597] as an attack against an appropriate definition of non-repudiation. Duplicate signatures are values m, m', s such that $\mathsf{Ver}^{r'}_{\mathsf{param},\mathsf{pk}}(m\|s) = m$ and $\mathsf{Ver}^{r'}_{\mathsf{param},\mathsf{pk}}(m'\|s) = m'$. This property also appeared in [123] where it was seen as a potential weakness.
> While the existence of duplicate signatures may be surprising, it is not a weakness of a digital signature scheme. If the scheme is existentially unforgeable, it proves that both signed messages were produced by a secret key holder.
> Moreover, one can notice that the secret key can easily be deduced from the duplicate signatures described in [123,597]. Therefore a user that shows duplicate signatures in these schemes is a user that publishes his secret key.

It is important that the secret key is not compromised, because any holder of the secret key can sign documents. Therefore, non-repudiation of origin is not a proof of security against viruses or Trojan horses. Forward secrecy and related properties [4, 15, 49, 313, 376, 411] deal with this problem. Such properties are not in the scope of the NESSIE evaluation and we make the hypothesis that the secret key is never compromised.

7.2.1.2 Other security models

While existential unforgeability under adaptive chosen message attack is the *de facto* standard notion of security for signature schemes, it does not cover all possible attacks. This security notion only considers a unique randomly generated param and a unique $(\mathsf{pk}, \mathsf{sk})$ pair. Non-uniqueness or non-random generation lead to realistic attack models that are not covered by existential unforgeability under adaptive chosen message attack.

Authentication of origin. An attacker against this property of a digital signature scheme is an algorithm that inputs a signed message σ valid for a public key pk and outputs another public key pk' such that σ is valid for pk'.

This property is not implied by existential unforgeability, but it is useful to mimic some properties of hand-written signatures, e.g. in applications where some write-only and time-stamped database is used to receive signed messages and then to solve disputes for anteriority.

The attack against the property of authentication of origin is named *duplicate-signature key selection* [94] or *key substitution* [440] or *key-collision* [546].

Multi-key/multi-user setting. These settings consider the case where a fixed param and many pk_i are used, and the adversary has access to signature oracles for all those keys.

The multi-user setting for digital signature schemes first appeared in [249,440] and its security requirements are related to those of the multi-user setting for asymmetric encryption schemes [43].

By definition, a (t, ε, n, q_S)-mu-forger has access to the signature oracles corresponding to n public keys pk_i, is allowed to make at most q_S queries to the oracles, runs in time t and outputs a forgery for some pk_i with probability ε. A scheme for which the existence of a (t, ε, n, q_S)-mu-forger implies the existence of a (t, ε, q_S)-forger is *secure in the multi-user setting.* This also implies authentication of origin.

Using similar techniques to those of Bellare *et al.* for multi-user security with asymmetric encryption, Galbraith *et al.* [249] showed how random self-reducibility of Schnorr-like signature schemes proves multi-user security.

A similar requirement appeared in [274] under the name *multi-key.* Here the best simultaneous attack on all the keys should be an independent attack on each key.

By definition, a (t, ε, n, q_S)-mk-forger has access to the signature oracles corresponding to n public keys pk_i, is allowed to make at most q_S queries to each oracle, runs in time nt and outputs a list of forgeries for each pk_i such that each forgery is valid with probability ε. A scheme for which the existence of a (t, ε, n, q_S)-mk-forger implies the existence of a (t, ε, q_S)-forger is *secure in the multi-key setting.*

This is the case if all the verification algorithms with distinct $(\mathsf{param}, \mathsf{pk})$ values are information-theoretically independent. However, most digital signature schemes share a common hash function for all public keys. Finding a collision in the hash function is a more efficient simultaneous attack on multiple keys than on a single key. Therefore it is good practice to have a key-dependent hash function, for example by including param and pk in the input of a common hash function.

Parameter manipulation. If param is not chosen at random, new attacks may appear. For example, param can be chosen with a trapdoor enabling the generation of valid signatures without knowing the secret key [96] or such that a collision is introduced in the hash function [444, 612].

One technique that protects against parameter manipulation is parameter validation, e.g. the publication of (k, ρ) together with param, but this might not be sufficient if the algorithm that generates the parameters from the seed leaves some freedom, as is the case for the ECDSA standardised technique [615]. Ideally, one should provide a security proof for $\mathsf{Generate}$. Another heuristic protection has been proposed [340, 444] to protect against an attacker that studies the properties of a hash function used in the scheme to select weak or trapdoor parameters: one can include the values $\mathsf{param}, \mathsf{pk}$ in the input to this hash function.

Side-channel attacks. A hidden assumption of our security model is that the attacker may have access to the input and output of the signing algorithm, but the attacker should not be able to get any information about intermediate values that appear during the computation of a signature.

An adversary that has access to this type of information is said to be able to mount a side-channel attack (see Appendix A for more details).

Key recovery vs. forgery. If one access to a forger allows an attacker to compute the secret key, then additional forgeries can be made without the forger and the message for these forgeries can be chosen by the attacker. For example, if a side-channel attack can succeed in producing a forgery, then a few accesses to side-channel information can allow an attacker to make an unlimited number of chosen-text-forgeries. Therefore, the existence of a reduction from forgery to key recovery can be seen as a weakness of the system.

7.2.2 Intractability assumptions

7.2.2.1 Mathematical problems

A *mathematical problem* is described by the set of instances of size l, a probability distribution on this set, and a set of possible solutions. For each instance, some of the possible solutions are valid and the others invalid.

Computational and decisional problems. If the set of possible solutions is $\{\mathsf{yes}, \mathsf{no}\}$ with one valid and one invalid for each instance, then it is called a decisional problem; else it is called a computational (or search) problem.

A *solution-checker* is an efficient algorithm that inputs an instance and a solution and tells if the solution is valid. The problem of the existence of a solution-checker is the decisional problem associated with the problem.

Concrete intractability. A *solver* for the problem is an efficient algorithm that inputs an instance and outputs a valid solution. A mathematical problem is (t', ε')-intractable for size l if there exists no solver running in time t' that succeeds with probability better than ε' for a random instance of size l. The problem has intractability k' bits if there exists no (t', ε')-solver with $\log_2(t'/\varepsilon') < k'$. The value k' depends on the time unit.

Asymptotic intractability. A problem is intractable if for any polynomial p the problem with size l has intractability $p(l)$ bits for large enough l. With the terminology of complexity theory an intractable problem (usually only defined for decisional problems) is a problem that is not a member of P.

NP-hard problems are proven to be at least as difficult to solve as all NP problems (problems that have a polynomial-time solution-checker). They may be good candidates for intractable problems. However NP-hardness only gives a bound for the worst case and not for an average instance of the problem, and hard instances may be difficult to generate.

7.2.2.2 Trusted cryptographic problems

No mathematical problem is provably intractable, but the following problems are good candidates and their intractability is assumed when proving the security of some cryptographic schemes.

Factorisation-based problems.

- **The integer factorisation problem.** An instance is a composite integer n of l bits. A solution is a non-trivial factor of n.
  The probability distribution of the instances is unclear. Usually, the integer n is constructed as pq or p^2q where p and q are randomly generated primes of similar sizes, because these are the most difficult instances for the current best factorisation algorithms. [1]
  The corresponding decisional problem is easy because the algorithm that computes the gcd of the instance and the solution is a solution-checker.
- **The RSA problem.** An instance is (n, e, y) where n is a composite integer of l bits, e is an integer coprime to $\phi(n)$ and $y \in (\mathbb{Z}/n\mathbb{Z})^\times$, the set of invertible elements modulo n. A solution is some $x \in (\mathbb{Z}/n\mathbb{Z})^\times$ such that $y = x^e$.
  The probability distribution of the instances is as unclear as for the factorisation problem. There exist two techniques for generating random instances: choosing n, then e and then y, or choosing e, then n and then y. The second technique is usually preferred because it allows one to fix the value of e in advance and use the e-th root problem below.
  The only known method of solving this problem is to solve the integer factorisation problem for n. The assumption that the RSA problem is hard is known as the RSA assumption.
  The corresponding decisional problem is easy because the algorithm that computes $x^e \overset{?}{=} y$ is a solution-checker.

[1] There is no simple method to detect if n is of one of these special forms. In general, the best known technique to detect if n is of one of these forms is to factor it.
Many algorithms exist for the generation of random primes and give different probability distributions. One efficient algorithm has been submitted to NESSIE [331].

- **The e-th root problem.** An instance with parameter e is (n, y) where n is a composite integer of l bits such that $\phi(n)$ is coprime to e and $y \in (\mathbb{Z}/n\mathbb{Z})^\times$. A solution is some $x \in (\mathbb{Z}/n\mathbb{Z})^\times$ such that $y = x^e$.
  This problem looks very similar to the RSA problem, but having a fixed value for e leads to some theoretical results, e.g. if e is a smooth integer [110]. One can also notice that if e is not coprime to $\phi(n)$, e.g. $e = 2$, then the e-th root problem (where y is a e-th power) is proven to be equivalent to the factorisation problem.
  The only known method of solving the e-th root problem is to solve the integer factorisation problem for n.
  The corresponding decisional problem is easy because the algorithm that computes $x^e \overset{?}{=} y$ is a solution-checker.
- **The flexible RSA problem.** An instance is (n, y) where n is a composite integer of l bits and $y \in (\mathbb{Z}/n\mathbb{Z})^\times$. A solution is some $x \in (\mathbb{Z}/n\mathbb{Z})^\times$ and $e > 1$ such that $y = x^e$.
  The only known method of solving this problem is to solve the integer factorisation problem for n. The assumption that this problem is hard is known as the *strong RSA assumption*.
  The corresponding decisional problem is easy because the algorithm that computes $x^e \overset{?}{=} y$ is a solution-checker.
- **The approximate e-th root problem (AER).** An instance with parameters e and d is (n, y) where n is a composite integer of l bits such that $\phi(n)$ is coprime to e and $y \in (\mathbb{Z}/n\mathbb{Z})^\times$. Let $\alpha_d(x) = \left\lfloor \frac{x \bmod n}{n^{(d-1)/d}} \right\rfloor$ the $n^{\frac{1}{d}}$-approximation of x. A solution is some $x \in (\mathbb{Z}/n\mathbb{Z})^\times$ such that $\alpha_d(y) = \alpha_d(x^e)$.
  For $d = 3$, this problem has been solved for $e < 4$. The only known method of solving this problem for $d = 3$ and $e \geq 4$ is to solve the integer factorisation problem for n.
  The corresponding decisional problem is easy because the algorithm that computes $\alpha_d(x^e) \overset{?}{=} \alpha_d(y)$ is a solution-checker.
- **The claw-free approximate e-th root problem (Claw-AER).** Parameters and instances are defined as for AER but a solution is a pair $x, z \in (\mathbb{Z}/n\mathbb{Z})^\times$ such that $\alpha_d(yz^e) = \alpha_d(x^e)$. This problem appeared recently [274] and its intractability is not well known.
- **The second-preimage approximate e-th root problem (2nd-AER).** Parameters and instances are defined as for AER but a solution is a value $x \in (\mathbb{Z}/n\mathbb{Z})^\times$ such that $x \neq y$ and $\alpha_d(x^e) = \alpha_d(y^e)$. This problem is related to the non-malleability of ESIGN and its intractability is not well known.

Discrete logarithm-based problems. All problems are parameterised by a cyclic group $\langle G \rangle$ and its generator G. The order of $\langle G \rangle$ is a number q with l bits. Group operations should be easy to compute, but elements of this group may be indistinguishable from elements of a larger group.

$\langle G \rangle$ is usually a subgroup of the multiplicative group of a finite field of prime order p or a subgroup of the group of points on an elliptic curve over a finite field.

- **The discrete logarithm problem.** An instance is H, a random element in $\langle G \rangle$. The solution is $x \in \mathbb{Z}/q\mathbb{Z}$ such that $G^x = H$.

This value is unique and is the discrete logarithm $\log H$.
The corresponding decisional problem is easy because the algorithm that computes $G^x \overset{?}{=} H$ is a solution-checker.

- **The computational Diffie-Hellman problem.** An instance is (H_1, H_2), both random elements of $\langle G \rangle$. The solution is H such that $\log H = \log H_1 \cdot \log H_2$.
  The only known method for solving this problem is to solve the discrete logarithm problem.
- **The decisional Diffie-Hellman problem.** An instance is (H, H_1, H_2), elements of $\langle G \rangle$. The solution is yes if $\log H = \log H_1 \cdot \log H_2$.
  The probability distribution of instances is given by the following algorithm: take independent random $x_1, x_2, x_3 \in \mathbb{Z}/q\mathbb{Z}$ and a random bit b, let $H_1 = G^{x_1}$, $H_2 = G^{x_2}$ and, depending on b, either $H = G^{x_3}$ or $H = G^{x_1 x_2}$.
  The only known generic method for solving this problem is to solve the computational Diffie-Hellman problem, hence to solve the discrete logarithm problem. However, there exist groups where an efficient algorithm solves the decisional Diffie-Hellman problem without solving the computational Diffie-Hellman problem (they are called GDH groups and are elliptic curves where the Tate/Weil pairing [235, 329] has special properties).
- **The gap Diffie-Hellman problem.** Solve the computational Diffie-Hellman problem with access to an oracle that answers the decisional Diffie-Hellman problem. In practice this problem appears in GDH groups.

Multivariate algebra-based problems.

- **The MQ problem.** The MQ problem is to find a solution to a given set of multivariate quadratic equations over a finite field, and is NP-hard in general.
- **The HFE problem and variants.** The HFE problem is a special case of the MQ problem where the set of equations is not random but constructed so that there is a trapdoor to their solution, and the HFEv$^-$ problem is an extension of the HFE problem which is harder to solve.
  The QUARTZ, FLASH and SFLASH problems are special cases of HFEv$^-$.

7.2.2.3 How to estimate concrete intractability

Best known solvers. There are some algorithms for solving the above problems. The notation $L_q[\alpha, c] = O(\exp((c + o(1))(\ln q)^\alpha (\ln \ln q)^{1-\alpha}))$ is used for the asymptotic complexity of some of them.

- **Integer factorisation.** The fastest known algorithms for factorising large integers are the Number Field Sieve [393] and the Elliptic Curve Method [397]. The asymptotic time taken by the number field sieve to factor an integer n is approximately $L_n[\frac{1}{3}, c_{\text{NFS}}]$ where c_{NFS} is a constant depending on the variant of number field sieve. [2] The asymptotic time taken by the elliptic curve method

[2] The General Number Field Sieve works for any integer n and has $c_{\text{NFS}} = (\frac{64}{9})^{1/3} \simeq 1.923$. The Special Number Field Sieve works for $n = k \cdot a^b \pm c$ and has $c_{\text{NFS}} = (\frac{32}{9})^{1/3} \simeq 1.526$.

to factor an integer whose smallest factor is p is $L_p[\frac{1}{2}, \sqrt{2}]$. Both algorithms are subexponential in the size of their input.
An improvement of the elliptic curve method exists for $n = p^2q$ [213, 517] and a special algorithm for $n = p^rq$ with large r [108].

- **Discrete logarithm over $\mathbb{F}_p$.** The index-calculus method [144,221,492] is the fastest known method of solving the discrete logarithm problem over $\mathbb{F}_p$. It is closely related to the number field sieve factoring algorithm and has expected asymptotic running time of $L_p[\frac{1}{3}, c_{\text{NFS}}]$, which is subexponential.
- **Elliptic curve discrete logarithm.** The fastest general methods of attack for solving the elliptic curve discrete logarithm problem are the Pollard ρ and the Pollard λ methods [524]. For a group with q elements, the Pollard ρ runs in time $\sqrt{\pi q/2}$, and the Pollard λ runs in time $2\sqrt{q}$ but can be faster in some special cases. Both can be efficiently parallelised [607] and have been slightly improved [250, 624]. No subexponential algorithm has been found for solving the elliptic curve discrete logarithm problem.
There exist subexponential attacks for specific elliptic curves: supersingular and similar curves [236, 282, 439, 550] and anomalous curves [554, 564, 587].
- **Generic group discrete logarithm.** Nechaev [474] and Shoup [581] proved that the best algorithm to solve the discrete logarithm in a generic group runs in time $\mathcal{O}(\sqrt{q})$. However, all known concrete groups can easily be distinguished from a generic group (they have automorphisms that are easy to compute from the binary representation of the elements).
- **MQ and related problems.** For the MQ and HFE problems the situation is somewhat more complicated and not as well studied. The MQ problem is usually solved by looking for a Gröbner basis — a method that can handle generic multivariate equations. Faugère [225] showed that the instances of HFE generated for QUARTZ are easier than generic instances and Courtois, Daum and Felke [157] studied the implications of this result.
The XL and FXL algorithms are designed for systems where the equations are quadratic and there is an attack of Shamir and Kipnis [566] on the HFE problem.

Quantum computers. Today large quantum computers don't exist, and they may never exist, but their theoretical aspects have been studied and many researchers are trying to build one. The largest quantum computer that has been built handled 7 q-bits. If larger quantum computers can be realised, their impact on cryptology is important, because some algorithms have been designed that can efficiently solve the integer factorisation or discrete logarithm problem with a quantum computer [580]. No algorithm is known that solves MQ and related problems faster with a quantum computer than with current computers.

Estimations for the difficulty of problems. While it is impossible to be sure that a cryptographic problem will remain intractable (because a fast polynomial algorithm might be discovered, that solves all NP problems), it is necessary to estimate the lifespan of a public key. Articles giving such estimates have been written by many researchers [396, 491, 586, 623] and have led to various conclusions. See also the discussion in [573].

The simplest presentation of their results can be given in terms of equivalent symmetric key size for a given security. The NESSIE call [475] asked for a security equivalent to an 80-bit symmetric key, by asking that the best attacks require the equivalent of 2^{80} Triple-DES operations.

- **Current practice.** For normal security most products use DES (56-bit symmetric key), 512-bit RSA moduli, 112-bit elliptic curve keys and 160/512-bit DSA. Higher security is obtained with Triple-DES (112 bits) or AES (128 bits), 1024-bit RSA and 160-bit elliptic curve keys.
- **Estimates by Certicom.** The Standards for Efficient Cryptography Group [136, appendix B.2] gives the following estimates for comparable key sizes.

Equivalent symmetric key size	56	80	112	128	192	256
RSA modulus length	512	1024	2048	3072	7680	15360
Elliptic curve key size	112	160	224	256	384	512

- **Estimate of Silverman.** A cost-based analysis [586] gives the following table.

Equivalent symmetric key size	56	64	80	96	112	128
RSA modulus length	430	530	760	1020	1340	1620

  With these estimates, the computing power for the factorisation of 1020-bit integers should not be available during the next 20 years, and a 768-bit integer should be factorised by public effort around 2019.
- **Estimates for the computing power available.** Blaze *et al.* [95] estimated in 1996 that a minimum of 75 bits was necessary to have security for commercial use, and that 90 bits were needed to protect data for the next 20 years.
- **Estimates by Lenstra and Verheul.** Their article [396] is the most complete study of this topic. They take many factors into account:
  - Trusted key length for secure block ciphers.
  - Increase of computing power and memory available on constant-cost computers.
  - Increase of the budget of attackers.
  - Cryptanalytic advances.

  Their results can be summarised with the following approximate table.

Equivalent symmetric key size	56	64	72	80	90	100
Elliptic curve key size	105	120	135	160	185	220
RSA modulus length	417	682	1024	1500	2236	3100
RSA cost-based equiv.	288	480	768	1150	1792	2600

  They also find equivalent dates for different key sizes if one makes the hypothesis that DES could be trusted until 1982.

Equivalent symmetric key size	56	64	72	80	90	100
Last year with trust	1982	1992	2002	2012	2025	2040

- **Bernstein's circuit for integer factorisation.** Recently, Bernstein [55] proposed a new hardware design for the linear algebra step of the Number Field Sieve that might reduce the cost of factorisation. Dedicated hardware for the sieving is also studied. The proposed measure of the efficiency of such techniques is "construction cost × run time", which is quite different from the classical count of the number of operations. This approach has been heavily discussed within the community of researchers in cryptography and algorithmic number theory [394, 557], but currently no real consensus emerges. It is likely that dedicated hardware will improve the cost of factorisation, but several estimations give different values, all being asymptotic. However, one can remark that Bernstein's techniques don't reduce the number of operations.

- **Some records.** Here is a table of some historical factorisation records.

Year	1978	1982	1986	1992	1999	2002
Bit size	150	170	289	429	512	525

  Exhaustive key search for 64 bits was achieved in 2002 with two years of computation on the spare time of thousands of computers [203]. Factorisation of a 525-bit number was achieved in 2002 with two months of computation on a few dozens of computers [31].
- **A conclusion.** To comply with the NESSIE requirement of an equivalence with a symmetric key of 80 bits, the size given in the above papers for an elliptic curve key is 160 bits, but the sizes for an RSA modulus range from 760 to 1500 bits. All submitted schemes use the intermediate value of 1024 bits.

  However, the NESSIE call was phrased in terms of number of operations, because it was felt that cost-based analysis introduces an additional parameter that is not very well understood. Moreover, the difference between attack techniques against symmetric and asymmetric schemes is already taken into account by the fact that the minimal computational cost of an attack against NESSIE-recommended symmetric primitives is 2^{128} operations while the minimal computational cost of an attack against NESSIE-recommended asymmetric primitives is 2^{80} operations.

  Therefore, we use the following table, based on the hypothesis that the factorisation of 512-bit numbers with NFS needs a workfactor of 2^{56} and that 190-bit factors are found by ECM with a workfactor of 2^{56}.

Equivalent symmetric key size	56	64	80	112	128	160
Elliptic curve key size	112	128	160	224	256	320
Modulus length (pq)	512	768	1536	4096	6000	10000
Modulus length (p^2q)	570	800	1536	4096	6000	10000

7.2.3 Proven security

A proof of security is the description of a (randomised) algorithm called a *reduction algorithm*. This algorithm is a (t', ε')-solver for some mathematical problem and interacts with a (t, ε, q_S)-forger for the signature scheme. For a random instance of the problem the reduction algorithm sends a random-looking public key to the forger and uses the forgery to solve the problem. Therefore $\varepsilon' \leq \varepsilon$ and $t' \geq t$.

Usually $q_S \ll t$, and we will require $k = \log_2(t/\varepsilon) = 80$ (the attacker's computing power is estimated to 2^{80} triple-DES operations) and $\log_2 q_S = 30$ (the attacker cannot require more than a billion signed messages).

Simulation of signature queries. The black box that is used by the forger to get the q_S signatures depends on the public key. Therefore it is provided by the reduction algorithm, which acts as an oracle. For each signature query made by the forger, the reduction algorithm should answer with a valid signed message, and this answer should be indistinguishable from an answer made by the signing algorithm with the secret key. Therefore the reduction algorithm is also named a *simulator*.

The reduction does not always succeed, partly because its simulation of an equivalent signature scheme may not be perfect, and partly because the forgery may be useless.

Uniformity of proofs. Uniform reduction means that the reduction algorithm does not depend on the description of the forger. Non-uniform reduction means that for any possible forger, there exists a reduction algorithm which is a solver. An example of non-uniform reduction can be found in [523] where the reduction algorithm depends on the success probability and number of queries of the forger.

Efficiency of proofs. We have a *tight* proof of security if $t'/\varepsilon' \simeq t/\varepsilon$.

We have a *not so tight* proof of security if $t'/\varepsilon' \simeq q_S t/\varepsilon$.

We have a *loose* proof of security if $t'/\varepsilon' \gg q_S t/\varepsilon$, for example $t'/\varepsilon' \simeq t^2/\varepsilon$.

For example, for the security proof to give 80 bits of security, factorisation-based schemes should use a 1536-bit modulus if they have a tight proof of security, a 4096-bit modulus with a not so tight proof and a 10000-bit modulus with a loose proof.

Equivalent signature schemes. Two signature schemes are equivalent when the following conditions are satisfied:

- The possible values of param are the same.
- The distributions of the pk generated are indistinguishable.
- Both verification algorithms are the same.
- The output of the two Sign algorithms for fixed m and random r are indistinguishable. When this last condition is omitted, it is a weak equivalence.

Any (t, ε, q_S)-forger for a signature scheme is also a (t, ε, q_S)-forger for all equivalent signature schemes. If $q_S \leq 1$, weak equivalence is sufficient.

A security proof for a signature scheme shows that if the mathematical problem is intractable, then there exists no forger for any scheme equivalent to it.

7.2.4 Proofs in an idealised world

Definition. A proof in an idealised world involves a restriction of the power of the attacker: some components of the verification algorithm cannot be computed by the forger alone, help from the simulator is needed. By definition all our algorithms are deterministic, hence the components are deterministic functions of their inputs.

Basic oracle property. An idealised component has the basic oracle property if the simulator knows the values of all inputs and outputs of this component. Usually the basic oracle property is introduced in the proof by requesting that for each access to this component, the attacker must send to the simulator a query containing the input. Then the simulator computes the output of the component and sends it to the attacker. The simulator behaves like an oracle for this function, hence the name.

Usually the number of queries to the oracle is bounded by q_O, and because the actual computation of the idealised components takes time, a scheme with k bits of security and with the appropriate time unit always has $q_O \leq 2^k$.

Randomness. Instead of computing the idealised component, the simulator may give random answers, provided that they agree with the properties of the component.

The simulator needs to remember all the inputs that are queried and the answers that are made, and make the appropriate consistency checks before answering. In other words, the reduction algorithm constructs the oracle input/output table in answer to queries.

Such an idealised proof makes the assumption that the component has no other useful property than the ones used for the consistency checks.

Programmability. Instead of choosing the answer at random from the set of consistent answers, the simulator is allowed to generate the answer with any technique that is indistinguishable from a random choice.

Example of idealised components and some terminology.

- **Random oracle model.** The idealised component is a hash function. The only consistency check is that two identical inputs get the same answer, hence the name [52]. This is the simplest possible consistency check and it was the first idealised model used for security proofs [229]. The alternative **generic hash model** terminology has been proposed [126].
  The random oracle model is the most widely used model for security proofs in an idealised world [52–54, 149].
- **Generic group model.** The idealised component is the group where some computations take place. The consistency check has to make sure that algebraic properties of the group operation are respected.
  The generic group model appeared in [474, 581] and has been used to prove the security of some specific schemes [124, 126].
- **Random permutation model.** The idealised component is a permutation, and oracle queries can be made for the permutation or for the inverse permutation. Consistency checks make sure that it is 1-to-1.
  This model is used to prove the security of some specific schemes [276].
- **Ideal cipher model.** The idealised component is a block cipher. This is a simultaneous use of multiple idealised permutations, one for each value of the key.
  This model is used to prove the security of some specific schemes [276, 323].

Can we trust security proofs in an idealised world? Proofs in these models cannot generically be translated into the real world [135, 194, 230], but it is widely believed that a proof in an idealised model gives some confidence in the design of a cryptographic primitive.

The impossibility result of Canetti, Goldreich and Halevi [135] shows that there exist systems that are secure in an idealised model, but insecure when the idealised component is replaced by *any* concrete component. Such a weakness is very unlikely to be present in a simple cryptographic design, but cannot be ruled out. The only solution is to find a proof in the real world.

Another problem with proofs in an idealised world is that they don't specify which concrete attacks against the idealised component should be prevented. The availability of different proofs that idealise distinct components gives a partial answer to this problem.

7.2.5 Assessment process

The digital signature submissions were assessed with reference to the submitted security proof. The underlying intractability assumption was reviewed.

Variants of the scheme were studied to verify whether the designers have made the optimal choices or not. When interesting variants were found, we interacted with the submitters to find justifications for keeping the submitted design unchanged. Compatibility with existing *de facto* standards was not received as a good argument for not improving a scheme, because the goal of NESSIE is not to recommend the *de facto* standards, but to recommend a portfolio of secure and efficient primitives.

7.3 Overview of the common designs

We describe three types of design commonly used to build digital signature schemes.

- The idea of using a trapdoor one-way function to obtain digital signatures dates back to 1978 [543]. This paradigm is also named hash-then-invert or hash-then-sign and NESSIE submissions RSA-PSS, ESIGN, Quartz, Flash and Sflash fall into this category.
- The discrete logarithm problem was the first basis for security in public key cryptography [201] and the first digital signature scheme based on this problem was introduced in 1985 [219]. A wide family of other schemes based on the discrete logarithm problem has been developed and the NESSIE submission ECDSA is one of them.
- The above schemes don't have a security proof without an idealised model. There exist many schemes that have a security proof in the "real world" and the NESSIE submission ACE-Sign is one of those.

This section contains much technical information about the design criteria for digital signature schemes, and the key ideas of the security proofs that support them. Much of the material in this section appeared in various places, but no previous overview of common designs for digital signature schemes covers all the schemes submitted to NESSIE.

7.3.1 Schemes based on trapdoor one-way functions

7.3.1.1 Properties of one-way functions

Family of one-way functions. A family of one-way functions is a collection of functions $\{\mathsf{f}_i : \mathcal{S}_i \to \mathcal{H}_i | i \in \mathcal{I}\}$ over some index set $\mathcal{I} = \bigoplus_l \mathcal{I}_l$ (disjoint union) with the following properties.

OW1 There is an efficient randomised algorithm Gen which takes as input a length parameter l and outputs an index $i \in \mathcal{I}_l$.

OW2 There is an efficient randomised algorithm which on input i outputs $x \in \mathcal{S}_i$. The resulting probability distribution is denoted $x \leftarrow \mathcal{S}_i$.

OW3 Each f_i is efficiently computable.
We remark that there is an efficient randomised algorithm that outputs some $y \in \mathcal{H}_i$: this algorithm generates $x \leftarrow \mathcal{S}_i$ and finds $y = \mathsf{f}_i(x)$. The resulting probability distribution is written $y \leftarrow \mathcal{H}_i$.

OW4 Preimage-resistance: for random $i \leftarrow \mathsf{Gen}(l), y \leftarrow \mathcal{H}_i$, the problem of finding a value $x \in \mathcal{S}_i$ such that $\mathsf{f}_i(x) = y$ is intractable.

OW5 Second-preimage-resistance: for random $i \leftarrow \mathsf{Gen}(l), x \leftarrow \mathcal{S}_i$, the problem of finding a value $z \in \mathcal{S}_i$, $z \neq x$, such that $\mathsf{f}_i(x) = \mathsf{f}_i(z)$ is intractable.

Property OW5 will imply non-malleability of the signature scheme. It is always true if f_i is injective.
Note that if the preimage-resistance has intractability k bits, then the set $\mathcal{H}_i$ has at least 2^k elements, because exhaustive sampling in $\mathcal{S}_i$ is always possible.

Family of claw-free functions. A family of claw-free functions is a collection of functions $\{\mathsf{f}_i : \mathcal{S}_i \to \mathcal{H}_i, \mathsf{g}_i : \mathcal{T}_i \to \mathcal{H}_i | i \in \mathcal{I}\}$ over some index set $\mathcal{I} = \bigoplus_l \mathcal{I}_l$ with the following properties.

CF1 There is an efficient randomised algorithm Gen which takes as input a length parameter l and outputs an index $i \in \mathcal{I}_l$.

CF2 There are efficient randomised algorithms which on input i output $x \in \mathcal{S}_i$ or $z \in \mathcal{T}_i$.

CF3 Each f_i and g_i are efficiently computable.

CF4 Claw-freeness: for a random $i \leftarrow \mathsf{Gen}(l)$, the problem of finding two values $x \in \mathcal{S}_i$ and $z \in \mathcal{T}_i$ such that $\mathsf{f}_i(x) = \mathsf{g}_i(z)$ is intractable.

We remark that if (f, g) is claw-free, both f and g are preimage-resistant.
Note that if the claw-freeness has intractability k bits then the set $\mathcal{H}_i$ has at least 2^{2k} elements, or exhaustive sampling in $\mathcal{S}_i$ and $\mathcal{T}_i$ would lead to a collision with the birthday paradox.

Uniformity. In the above general definitions, the distributions $i \leftarrow \mathsf{Gen}(l)$, $x \leftarrow \mathcal{S}_i$ and $y \leftarrow \mathcal{H}_i$ don't need to be uniform.
A family f is said to be uniform if the following additional property holds.

UN The distribution $y \leftarrow \mathcal{H}_i$ is indistinguishable from the uniform distribution in $\mathcal{H}_i$.

Trapdoor invertibility. The family f is invertible with a trapdoor if the following additional properties hold.

TR1 The algorithm Gen also outputs a trapdoor trap_i.

TR2 There is an efficient (randomised) algorithm that on input $i, \mathsf{trap}_i, y \in \mathcal{H}_i$ returns a value $x = \mathsf{f}_i^{-1}(y)$ such that $\mathsf{f}_i(x) = y$.

TR3 The distribution of x generated by $y \leftarrow \mathcal{H}_i$ and $x = \mathsf{f}_i^{-1}(y)$ is indistinguishable from the distribution $x \leftarrow \mathcal{S}_i$.

Note that in many examples of such families of functions we have $\mathcal{S}_i = \mathcal{T}_i = \mathcal{H}_i$ and f_i and g_i are permutations, but these more general definitions are necessary to cover all submissions to NESSIE.

A generalisation to verifiable simulatable functions. While the usual definition of the hash-then-invert paradigm makes the hypothesis that the f_i are efficiently computable, this requirement can be relaxed for the FDH and PFDH constructions, as described below. This is used e.g. in Sect. 7.3.2.11.

A family of verifiable simulatable one-way functions is similar to a family of one-way functions, but with the properties OW2 and OW3 replaced by

OW2$_\mathsf{S}$ There exists an efficient randomised algorithm S_i^f which on input i outputs a pair $(x, y) \in \mathcal{S}_i \times \mathcal{H}_i$ such that $y = \mathsf{f}_i(x)$. The corresponding probability distributions are written $x \leftarrow \mathcal{S}_i$ and $y \leftarrow \mathcal{H}_i$. This algorithm simulates $x \leftarrow \mathcal{S}_i,\ y = \mathsf{f}_i(x)$.

OW3$_\mathsf{T}$ Each test function $\mathsf{T}_i^\mathsf{f} : \mathcal{S}_i \times \mathcal{H}_i \to 0/1$ defined by $\mathsf{T}_i^\mathsf{f}(x, y) = \big(y \overset{?}{=} \mathsf{f}_i(x)\big)$ is efficiently computable.

A family of verifiable simulatable claw-free functions is similar to a family of claw-free functions, but with the properties CF2 and CF3 replaced by

CF2$_\mathsf{S}$ There exist two efficient randomised algorithms S_i^f and S_i^g which on input i output a pair $(x, y) \in \mathcal{S}_i \times \mathcal{H}_i$ such that $y = \mathsf{f}_i(x)$ or a pair $(z, y) \in \mathcal{T}_i \times \mathcal{H}_i$ such that $y = \mathsf{g}_i(z)$ and such that the two corresponding distributions $y \leftarrow \mathcal{H}_i$ are indistinguishable.

CF3$_\mathsf{T}$ The test functions $\mathsf{T}_i^\mathsf{f} : \mathcal{S}_i \times \mathcal{H}_i \to 0/1$ and $\mathsf{T}_i^\mathsf{g} : \mathcal{T}_i \times \mathcal{H}_i \to 0/1$ defined by $\mathsf{T}_i^\mathsf{f}(x, y) = \big(y \overset{?}{=} \mathsf{f}_i(x)\big)$ and $\mathsf{T}_i^\mathsf{g}(z, y) = \big(y \overset{?}{=} \mathsf{g}_i(z)\big)$ are efficiently computable.

7.3.1.2 FDH: Full Domain Hash

Definition. This is the most natural scheme. It was formally introduced in 1988 [46] and was provided with a security proof in the random oracle model in 1993 [52].

The f-FDH digital signature scheme with appendix is defined as follows. For any $i \in \mathcal{I}$ the components are a hash function H_i with output in $\mathcal{H}_i$ and a trapdoor invertible verifiable function $\mathsf{f}_i : \mathcal{S}_i \to \mathcal{H}_i$ with test T_i^f. The appendix is an element of $\mathcal{S}_i$ and H_i is idealised as a (programmable) random oracle.

- The key generation algorithm runs Gen and sets $\mathsf{pk} = i$ and $\mathsf{sk} = \mathsf{trap}_i$.
- The verification algorithm on $m\|s$, where m is the message and $s \in \mathcal{S}_i$ the appendix, checks if $\mathsf{T}_i^\mathsf{f}(s, \mathsf{H}_i(m)) \overset{?}{=} 1$.
- The signature generation algorithm computes $h = \mathsf{H}_i(m)$ and uses the trapdoor to compute $s = \mathsf{f}_i^{-1}(h)$. The signed message is (m, s).

Note that in the multi-key setting the hash functions H_i should be independent. However, in most actual published FDH schemes H_i depends only on the length parameter.

Theorem 7.1 (Necessary conditions). *The following conditions are necessary to the existential unforgeability of the* f*-FDH signature scheme.*

1. *Preimage-resistance of* f*.*
2. *Second-preimage-resistance of* f *is necessary for non-malleability.*
3. *Collision-resistance of* H*, which implies second-preimage-resistance.*
4. *Preimage-resistance of* H*.*

Proof. We describe an attacker if one of the conditions does not hold.

1. The attacker computes $h = \mathsf{H}_i(m)$ and finds a preimage $s \in \mathsf{f}_i^{-1}(\{h\})$.
2. The attacker queries for a valid signed message (m, s) and computes another $s' \in \mathsf{f}_i^{-1}(\{\mathsf{H}_i(m)\})$.
3. With a collision $\mathsf{H}_i(m) = \mathsf{H}_i(m')$ with $m \neq m'$, the attacker queries for a valid signed message (m, s). Then (m', s) is a new valid signed message.
4. The attacker computes a pair $(s, h) = \mathsf{S}_i^{\mathsf{f}}$ and finds a preimage $m \in \mathsf{H}_i^{-1}(\{h\})$. Then (m, s) is a valid signed message.

□

Theorem 7.2. (Security result if f is verifiable simulatable one-way). *Let* f *be uniform one-way with preimage-resistance of* $k + \log_2 q_H$ *bits and second-preimage-resistance of* k *bits. If either* f_i^{-1} *is deterministic or the forger is* SO-CMA*, then in the random oracle model with* q_H *hash queries and* q_S *signing queries* f*-FDH has a security level of* k *bits.*

Proof. This result dates back to Bellare and Rogaway [52]. It is a special case of the security result for PFDH (see next section). □

Generic attack. Theorem 7.2 is the best possible generic security result for a FDH scheme, because it applies to a trapdoor invertible bijection of a set of 2^{2k} elements with preimage resistance of $2k$ bits. Such a trapdoor invertible bijection might exist, and the FDH scheme based on this function can be broken with 2^k hash queries and no signature query, by looking for a collision between random $\mathsf{H}_i(m)$ and random $\mathsf{f}_i(s)$.

Theorem 7.3. (Security result if f comes from a verifiable simulatable claw-free pair). *Let* (f, g) *be uniform claw-free with intractability of* $k + \log_2 q_S$ *bits with* f *having second-preimage-resistance of* k *bits. If either* f_i^{-1} *is deterministic or the forger is* SO-CMA*, then in the random oracle model with* q_H *hash queries and* q_S *signing queries* f*-FDH has a security level of* k *bits.*

Proof. This result applied to RSA dates back to Coron [149]. Its generalisation to claw-free pairs is due to Dodis and Reyzin [206]. It is a special case of the security result for PFDH (see next section). □

Generic attack. Theorem 7.3 is the best possible security result for a generic FDH scheme, because it applies to a trapdoor invertible bijection of a set of 2^{2k} elements with claw-freeness of k bits. Such a trapdoor invertible bijection might exist, and the FDH scheme based on this function can be broken with 2^k hash queries and no signature query, by looking for a collision between random $\mathsf{H}_i(m)$ and random $\mathsf{f}_i(s)$.

7.3.1.3 FDH with a non-deterministic f_i^{-1}

Definition. For non-deterministic f_i^{-1} the previous results only prove the security of the FDH design if the forger is not allowed to make multiple queries for a single message. However, it is possible to tweak the design to make it deterministic and have proven security. The following FDH-D technique is used in the FLASH and QUARTZ families [161, 514] and in ESIGN-D [274].

Let $\mathsf{f}_i^{-1}(r, h)$ denote the randomised algorithm that computes a preimage of h with random seed $r \leftarrow \mathcal{R}_h$. Let prf be a family of pseudo-random functions such that the output distribution of $\mathsf{prf}_{\Delta}(h)$ for uniform random Δ is indistinguishable from the distribution $r \leftarrow \mathcal{R}_h$.

– The key generation algorithm chooses a random k-bit index Δ, runs Gen and sets $\mathsf{pk} = i$ and $\mathsf{sk} = (\mathsf{trap}_i, \Delta)$.
– The verification algorithm on $m\|s$, where m is the message and $s \in \mathcal{S}_i$ the appendix, checks if $\mathsf{T}_i^{\mathsf{f}}(s, \mathsf{H}_i(m)) \overset{?}{=} 1$.
– The signature generation computes $h = \mathsf{H}_i(m)$ and $r = \mathsf{prf}_{\Delta}(h)$ and uses the trapdoor to compute $s = \mathsf{f}_i^{-1}(r, h)$. The signed message is (m, s).

A remark. For some schemes (QUARTZ or ESIGN-D) the distribution $r \leftarrow \mathcal{R}_h$ is defined as follows. A value r is chosen uniformly from a set $\mathcal{R}$. If it is incompatible with h then it is discarded and another r is chosen, until a compatible value is found. Then prf_{Δ} is a function that generates an (infinite) sequence of uniform values $r \in \mathcal{R}$ and takes the first that is compatible with h.

7.3.1.4 PFDH: Probabilistic Full Domain Hash

Definition. This scheme was defined by Coron [151] but the underlying ideas date back to Bellare and Rogaway [54].

For any $i \in \mathcal{I}$ the components are a hash function H_i with output in $\mathcal{H}_i$, a trapdoor invertible verifiable function $\mathsf{f}_i : \mathcal{S}_i \to \mathcal{H}_i$ with test $\mathsf{T}_i^{\mathsf{f}}$ and a set $\mathcal{R}_i$ with uniform sampling in 2^{k_i} elements. The appendix is an element of $\mathcal{S}_i \times \mathcal{R}_i$ and H_i is idealised as a (programmable) random oracle.

– The key generation algorithm runs Gen and sets $\mathsf{pk} = i$ and $\mathsf{sk} = \mathsf{trap}_i$.
– The verification algorithm on $m\|r\|s$, where m is the message, $r \in \mathcal{R}_i$ and $s \in \mathcal{S}_i$, checks if $\mathsf{T}_i^{\mathsf{f}}(s, \mathsf{H}_i(m\|r)) \overset{?}{=} 1$.
– The signature generation algorithm generates $r \leftarrow \mathcal{R}_i$, computes $h = \mathsf{H}_i(m\|r)$ and uses the trapdoor to compute $s = \mathsf{f}_i^{-1}(h)$. The signed message is (m, r, s).

An alternative description of PFDH is that it is an FDH signature scheme on messages of the form $m\|r$, with a restriction on the attacker. The attacker is not able to decide the r-part of his queries to the signature oracle. Therefore if $\mathcal{R}_i$ is sufficiently large, the attacker does not learn anything useful from the signing queries.

Theorem 7.4. (Security result if f is verifiable simulatable one-way). *Let f be uniform one-way with preimage-resistance of $k + \log_2 q_H$ bits and second-preimage-resistance of k bits. If either f_i^{-1} is deterministic or $q_S < \sqrt{2^{k_i}}$, then*

in the random oracle model with q_H hash queries and q_S signing queries f*-PFDH has a security level of k bits.*
Note that when based on one-wayness, PFDH does not have better security than FDH.

Proof. This result is a generalisation of the proof by Bellare and Rogaway for FDH [52]. Dodis and Reyzin [206] give an argument showing that no better proof can be found in a black box model.

Assuming the existence of a $(t, \varepsilon, q_S, q_H)$-forger for f-PFDH, we show how to construct an algorithm (the simulator) that runs in time $t' \simeq 2t$ and either breaks the preimage-resistance of f with probability $\varepsilon' \simeq \varepsilon/(q_H + q_S)$ or breaks the second-preimage-resistance of f with probability $\varepsilon'' \simeq \varepsilon$. [3]

The simulator receives a challenge $(i, y \in \mathcal{H}_i)$. Then it chooses a random element $j_0 \in \{1...q_H + q_S\}$ and runs the forger on $\mathsf{pk} = i$.

For a hash query $m_j \| r_j$, if the answer was already defined then it is returned. Else, if it is the j_0-th query then the answer is the challenge y, and otherwise the simulator picks a random $(x_j, y_j) \leftarrow \mathsf{S}_i^{\mathsf{f}}$ and sets $\mathsf{H}_i(m_j \| r_j) = y_j$. This simulates the hash function because y_j has uniform distribution (property UN).

For a signing query m_j the simulator generates $r_j \leftarrow \mathcal{R}_i$ and internally simulates a hash query for $m_j \| r_j$. It returns the corresponding $r_j \| x_j$. This fails if the j_0-th query is a signing query, which happens with probability $\frac{q_S}{q_H + q_S}$.

It is also necessary to show that the simulator makes a good simulation of the signing algorithm. Property TR3 implies that the simulation is valid for signing queries of distinct values of m. A problem may only arise if multiple signature queries for the same message are made [597]. If two answers have the same r_j, then the simulation also answers the same x_j. If f_i^{-1} is deterministic, this is exactly the right behaviour. Else we should avoid such a collision, which we do if $q_S < \sqrt{2^{k_i}}$ or if the forger is $\mathsf{SO\text{-}CMA}$.

The forgery is some $m \| r \| s$. If the signing oracle never received m as a query and returned an answer $r_j \| x_j$ with $r_j = r$, then $\mathsf{H}_i(m \| r)$ is unset unless it was a hash query. In that case, if the forgery corresponds to the j_0-th query (which happens with probability $\frac{1}{q_H + q_S}$), then it gives a preimage of y. This contradicts property OW4 of one-way functions.

Else the signing oracle received m as a query and returned an answer $r \| x_j$ with $x_j \neq s$. This contradicts the second-preimage resistance, property OW5 of one-way functions.

The running time of the simulator is the running time t of the forger plus the time corresponding to $q_H + q_S$ executions of the verification algorithm, which is bounded by t. □

[3] Dodis and Reyzin propose a proof where $\frac{\varepsilon'}{\varepsilon} \simeq \frac{1}{q_H + 1}$, but their proof is flawed in the following way. (The reader is invited to look at the sketch proof of (b) in Sect. 3 of [206]). If the forger makes the corresponding hash query after each signing query, then with probability $\frac{q_S}{q_H}$ their simulator will be unable to answer the selected hash query. If $q_H - q_S \ll q_H$, the success of the simulator is $\frac{\varepsilon'}{\varepsilon} \simeq \frac{1}{q_H^2}$.

Theorem 7.5. (Security result if f comes from a verifiable simulatable claw-free pair). *Let* (f,g) *be uniform claw-free with intractability of* $k+\max(\log_2 q_S - k_i, 0)$ *bits, with* f *having second-preimage-resistance of* k *bits. If either* f_i^{-1} *is deterministic or* $q_S < \sqrt{2^{k_i}}$*, then in the random oracle model with* q_H *hash queries and* q_S *signing queries* f*-PFDH has a security level of* k *bits. When based on claw-freeness, PFDH does have better security than FDH.*

Proof. This result dates back to Bellare and Rogaway [52] and Coron [151] showed that this is optimal (under reasonable assumptions). The generalisation to claw-free pairs is due to Dodis and Reyzin [206].

Assuming the existence of a (t,ε,q_S,q_H)-forger for f-PFDH, we construct a simulator that either breaks the claw-freeness of (f,g) or breaks the second-preimage-resistance of f.

The simulator receives a challenge i. Then it generates a list L of q_S random elements of $\mathcal{R}_i$ and runs the forger on $\mathsf{pk}=i$. Some elements may have multiple occurrences in L.

For a hash query $m_j\|r_j$, if the answer was already defined then it is returned. Else, if $r_j \in L$ the simulator picks a random $(x_j,y_j) \leftarrow \mathsf{S}_i^{\mathsf{f}}$ and sets $\mathsf{H}_i(m_j\|r_j) = y_j$. Else the simulator picks a random $(z_j,y_j) \leftarrow \mathsf{S}_i^{\mathsf{g}}$ and sets $\mathsf{H}_i(m_j\|r_j) = y_j$. This simulates the hash function because y_j has a uniform distribution (property UN).

For a signing query m_j the simulator takes an element $r_j \in L$ and internally simulates a hash query for $m_j\|r_j$. Then it deletes r_j from L. Since r_j was in L the hash query has generated an x_j and the simulator can return the appendix $r_j\|x_j$.

For the same reason as in the previous proof, this is a good simulation of the signing algorithm if f_i^{-1} is deterministic, if $q_S < \sqrt{2^{k_i}}$ or if the forger is SO-CMA.

If the forgery $m\|r\|s$ does not contradict the second-preimage resistance, then it corresponds to a hash query, which either had an f answer or a g answer. If it corresponds to a g answer, then it contradicts the claw-freeness because $\mathsf{f}(s) = \mathsf{g}(z_j)$. When the list L contains q elements, a g answer happens with probability $(1-2^{-k_i})^q$. The number of elements of L decreases regularly during the simulation, so the probability that the forgery corresponds to a g answer is $\varepsilon'/\varepsilon = \frac{1}{q_S}\sum_{q=0\ldots q_S}(1-2^{-k_i})^q$.
If $q_S \ll 2^{k_i}$ then $\varepsilon'/\varepsilon \simeq (1-2^{-k_i}q_S) \simeq 1$ and therefore the security loss is $-\log_2(\varepsilon'/\varepsilon) \simeq 0$.
If $q_S \gg 2^{k_i}$ then $\varepsilon'/\varepsilon \simeq \frac{1}{q_S}\frac{1}{2^{-k_i}}$ and therefore the security loss is $-\log_2(\varepsilon'/\varepsilon) \simeq \log_2 q_S - k_i$. □

7.3.1.5 PSS and PSSR: partial message recovery

Definition. PSS means Probabilistic Signature Scheme, and PSSR means Probabilistic Signature Scheme with message Recovery.

The PSS and PSSR schemes are due to Bellare and Rogaway [54]. They include some message recovery in (P)FDH. The key idea is that the output of H can be smaller than $\mathcal{H}_i$, provided that it is collision-intractable and that the input of f_i^{-1} is random.

For any $i \in \mathcal{I}$, the recovered part of the message is a_i bits long and it is an element of the set $\mathcal{A}_i = \{0,1\}^{a_i}$. The components of the scheme are a hash function H_i with output in $\mathcal{B}_i$, a hash function $\mathsf{G}_i : \mathcal{B}_i \to \mathcal{A}_i$ and a trapdoor invertible function $\mathsf{f}_i : \mathcal{S}_i \to \mathcal{H}_i$ such that $\mathcal{H}_i = \mathcal{A}_i \times \mathcal{B}_i$. The functions H_i and G_i are idealised as (programmable) random hash oracles. For k bits of security the set $\mathcal{B}_i$ should have at least 2^{2k} elements.

- The key generation algorithm runs Gen and sets $\mathsf{pk} = i$ and $\mathsf{sk} = \mathsf{trap}_i$.
- The verification of $\hat{m}\|s$ computes $a\|b = \mathsf{f}_i(s)$, $\bar{m}\|r = a \oplus \mathsf{G}_i(b)$ and checks $\mathsf{H}_i(\hat{m}\|\bar{m}\|r) \overset{?}{=} b$. The message is $m = \hat{m}\|\bar{m}$.
- The signature generation for the message $m = \hat{m}\|\bar{m}$ computes $h = \mathsf{H}_i(\hat{m}\|\bar{m}\|r)$ and $a = (\bar{m}\|r) \oplus \mathsf{G}_i(h)$ and uses the trapdoor to compute $s = \mathsf{f}_i^{-1}(a\|h)$. The signed message is $\hat{m}\|s$.

PSS is the special case where this scheme is applied to PFDH with $\hat{m} = m$ and $\bar{m} = 0...0$ or another constant.

Theorem 7.6. (Security result). *PSS(R) has the same security as (P)FDH.*

Proof. This result is due to Coron [151] and is an improvement on the original result of Bellare and Rogaway [54]. For a complete proof, the reader should look at those papers.

The main difference from (P)FDH is that the unique hash function $m \mapsto \mathsf{H}_i(m)$ is replaced by $m \mapsto a\|h$ where $h = \mathsf{H}_i(m)$ and $a = \bar{m} \oplus \mathsf{G}_i(h)$. If there is no collision in h, then all oracle queries to H_i or to G_i make a commitment to some value for m. This is the essential reason why the security of PSS(R) is the same as the security of (P)FDH if the number of hash-oracle queries is at most the square root of the number of elements of $\mathcal{F}_i$.

Note that this theorem also applies to the variant with $h\|a$ instead of $a\|h$, which just uses the different definition $\mathcal{H}_i = \mathcal{B}_i \times \mathcal{A}_i$ and is also named PSS. □

7.3.1.6 OPSSR: maximal message recovery

Definition of Basic OPSSR. OPSSR means Optimal Padding for Signature Schemes with message Recovery.

The Basic OPSSR scheme is due to Granboulan [276] and is designed to have maximal message recovery. It is based on the fact that the important property of H_i is not one-wayness but collision-intractability. Therefore a (random) permutation can be used instead of a hash function.

For any $i \in \mathcal{I}$, the set of possible messages is $\mathcal{M}_i$ and $\mathcal{V}_i$ is a set of 2^k elements. The components of the scheme are a bijection $\mathsf{P}_i : \mathcal{H}_i \to \mathcal{M}_i \times \mathcal{V}_i$ and a trapdoor invertible function $\mathsf{f}_i : \mathcal{S}_i \to \mathcal{H}_i$. The function P_i is idealised as a (programmable) random permutation oracle.

- The key generation algorithm runs Gen and selects an arbitrary public value $v_i \in \mathcal{V}_i$, e.g. $v_i = 0^k$. It sets $\mathsf{pk} = (i, v_i)$ and $\mathsf{sk} = (\mathsf{trap}_i, v_i)$.
- The verification algorithm on s computes $m\|v = \mathsf{P}_i(\mathsf{f}_i(s))$ and checks if $v \overset{?}{=} v_i$.
- The signature generation algorithm uses the trapdoor to compute the signed message $s = \mathsf{f}_i^{-1}(\mathsf{P}_i^{-1}(m\|v_i))$.

A probabilistic variant of OPSSR can be defined as in PFDH, by replacing m with $m\|r$.

Definition of OPSSR. The full OPSSR scheme can do partial message recovery, which allows the signer to sign messages of arbitrary length with a fixed public key. It is a variant of Basic OPSSR where one replaces the bijection P_i by a keyed family E_i with keyspace equal to the output of a collision-intractable hash function $\hat{\mathsf{H}}_i$. It is defined by $\mathsf{P}_i(\hat{m}\|a) = \hat{m}\|\mathsf{E}_i[\hat{\mathsf{H}}_i(\hat{m})](a)$. The family E_i is idealised as a (programmable) ideal cipher.

- The key generation algorithm is the same as for Basic OPSSR.
- The verification algorithm on $\hat{m}\|s$ computes $m\|v = \hat{m}\|\mathsf{E}_i[\hat{\mathsf{H}}_i(\hat{m})](\mathsf{f}_i(s))$ and checks if $v \overset{?}{=} v_i$.
- The signing algorithm uses the trapdoor to compute the signed message $\hat{m}\|s$ where $s = \mathsf{f}_i^{-1}(\mathsf{E}_i^{-1}[\hat{\mathsf{H}}_i(\hat{m})](\bar{m}\|v_i))$.

Comparison with previous paddings. FDH is a special case of Basic OPSSR based on the involution $\mathsf{P}_i(m\|v) = m\|(v_i \oplus v \oplus \mathsf{H}_i(m))$ and on the family $\{\mathsf{f}_i^{\mathcal{M}} : \mathcal{M} \times \mathcal{S}_i \to \mathcal{M} \times \mathcal{H}_i\}$ defined by $\mathsf{f}_i^{\mathcal{M}}(m\|x) = m\|\mathsf{f}_i(x)$. This family $\mathsf{f}^{\mathcal{M}}$ has the same properties as f, but the proof given below does not apply to FDH because this function P_i is trivially not a random permutation.

PSS(R) is another special case of Basic OPSSR based on $\mathsf{P}_i(\hat{m}\|a\|b) = \hat{m}\|\bar{m}\|(v_i \oplus b \oplus \mathsf{H}_i(\hat{m}\|\bar{m}))$, where $\bar{m} = a \oplus \mathsf{G}_i(b)$, and its inverse $\mathsf{P}_i^{-1}(\hat{m}\|\bar{m}\|v) = \hat{m}\|(\bar{m} \oplus \mathsf{G}_i(b))\|b$, where $b = v_i \oplus v \oplus \mathsf{H}_i(\hat{m}\|\bar{m})$, and on the family $\mathsf{f}^{\hat{\mathcal{M}}}$. But the proof given below does not apply to PSS(R) because this function P_i is trivially not a random permutation.

The main advantage of OPSSR compared to PSSR is that the message expansion can be reduced to k bits (the size of v) instead of $2k$ bits (the size of $\mathcal{B}_i$).

Theorem 7.7. (Security result). *OPSSR has the same security as (P)FDH, where the random permutation model replaces the random oracle model.*

Proof. This result is due to Granboulan [276]. For a complete proof, the reader should look at this paper.

As with PSS(R), the key idea is that all oracle queries to P_i or P_i^{-1} make a commitment to some value for m. □

7.3.1.7 CPC : generalised Chained Patarin Construction

Definition. This technique is used in Quartz [161] and is also named the generalised Feistel-Patarin construction [155].

The parameter $r \geq 1$ is the number of rounds. The components are r hash functions $(\mathsf{H}_{i,j})_{j=1\ldots r}$ with output in $\mathcal{S}_i$ and a trapdoor function $\mathsf{f}_i : \mathcal{S}_i \times \mathcal{T}_i \to \mathcal{S}_i$. The $\mathsf{H}_{i,j}$ are idealised as programmable random oracles, the set $\mathcal{S}_i$ has a group operation $\oplus$ and the appendix is an element of $\mathcal{S}_i \times \underbrace{\mathcal{T}_i \times ... \times \mathcal{T}_i}_{r \text{ times}}$,

- The key generation algorithm runs Gen and sets $\mathsf{pk} = i$ and $\mathsf{sk} = \mathsf{trap}_i$.

- The verification algorithm on $m\|s\|t_1\|...\|t_r$ sets $s_r = s$, computes the sequence $s_{j-1} = \mathsf{f}_i(s_j\|t_j) \oplus \mathsf{H}_j(m)$ for $j = r...1$ and checks if $s_0 \overset{?}{=} 0$.
- The signature generation algorithm sets $s_0 = 0$, computes the sequence $s_j\|t_j = \mathsf{f}_i^{-1}(s_{j-1} \oplus \mathsf{H}_j(m))$ for $j = 1...r$ and sets $s = s_r$.

If f_i^{-1} is not deterministic the same technique as for FDH-D can be used and is named CPC-D.

Theorem 7.8. (Security result). *While the proven security of CPC is the same as for FDH, the generic attack against FDH does not work against CPC. Therefore the actual security of CPC may be better than FDH.*

In fact, Courtois proved that if $q_S = 0$ then the generic attack against CPC is the best possible.

Proof. See Courtois' paper [155]. □

Generic attack. Let S be the number of elements of $\mathcal{S}_i$ and let us compute $S^{r/(r+1)}$ random values $(s\|t, \mathsf{f}_i(s\|t))$. These values allow us to invert f_i with probability $S^{-1/(r+1)}$.

The generic attack then generates $S^{r/(r+1)}$ random messages m and computes the corresponding $(\mathsf{H}_{i,j}(m))_{j=1...r}$. For each m the probability that a forgery can be made by using the precomputed partial table for f_i^{-1} is $S^{-r/(r+1)}$, so the average number of forgeries made by this attack is 1.

7.3.2 Schemes based on the Discrete Logarithm Problem

7.3.2.1 Introduction

We describe how most DL-based signature schemes [5, 319, 340, 445, 447, 455, 464, 489, 490, 519, 559, 560] can be built by mixing four components, which we call a group, a hash function, a projection and a category. We describe some possible values of each component, and give some security proofs.

All the security proofs for DL-based signature schemes work in an idealised model. Therefore it may happen that the scheme is not secure when a concrete component is chosen to replace an idealised component, and it may even happen that any choice of a concrete component makes an insecure scheme. We will show how the security can be proven in three independent ways: when the group is idealised, or when the hash function is idealised, or when the projection is idealised. A scheme for which a security proof exists in all three models is secure in the real world unless all three concrete components are weak choices.

The toolbox. The signature scheme is built on

- A **DL-group** $\langle G\rangle$ of order q (usually a prime number) where computing discrete logarithms is hard.
  The set of possible private keys is $\mathcal{V} \subset \mathbb{Z}/q\mathbb{Z}$. If the private key of the signer is $v \in \mathcal{V}$, the corresponding public key is the element $V = G^v$. Depending on the category, we may need $\mathcal{V} = \mathbb{Z}/q\mathbb{Z}$ or $\mathcal{V} = (\mathbb{Z}/q\mathbb{Z})^\times$.
  In this section all groups will be denoted multiplicatively, even in the case of elliptic curves.

- A **projection**. This is a function $\mathsf{p} : \langle G \rangle \to \mathcal{R}$, where $\mathcal{R}$ can be an arbitrary set.
- A **hash function** that makes a digest of the message. It is a function $\mathsf{H} : \mathcal{M} \times \mathcal{R} \to \mathcal{H}$, where $\mathcal{M}$ is the set of possible messages.
- A **category** that defines the formulae for signature and verification.
  The category defines two functions ϕ and $\psi : \mathcal{H} \times \mathcal{R} \times \mathcal{S} \to \mathbb{Z}/q\mathbb{Z}$ and a function $\sigma : \mathcal{I} \to \mathcal{S}$, where $\mathcal{I} \subset \mathcal{H} \times \mathcal{R} \times \mathcal{V} \times \mathcal{K}$, $\mathcal{S} = \mathbb{Z}/q\mathbb{Z}$ or $(\mathbb{Z}/q\mathbb{Z})^\times$ and $\mathcal{K} = \mathbb{Z}/q\mathbb{Z}$ or $(\mathbb{Z}/q\mathbb{Z})^\times$.

Description. The digital signature scheme works as follows:

- *Verification.* The verification of $(m, r, s) \in \mathcal{M} \times \mathcal{R} \times \mathcal{S}$ computes $h = \mathsf{H}(m, r)$, $\alpha = \phi(h, r, s)$, $\beta = \psi(h, r, s)$, $R = G^\alpha V^\beta$ and checks if $r \overset{?}{=} \mathsf{p}(R)$.
- *Signature.* To sign the message m one takes a random $k \in \mathcal{K}$ and computes $R = G^k$, $r = \mathsf{p}(R)$ and $h = \mathsf{H}(m, r)$, until $(h, r, v, k) \in \mathcal{I}$ and $s = \sigma(h, r, v, k)$. The signed message is (m, r, s).
- *Parameters and keys.* For most DL-based schemes, the description of $\langle G \rangle$ is a public parameter and the public key is V. Schemes where both $\langle G \rangle$ and V are in the public key might be more secure in the multi-key setting.
- *Partial message recovery.* For some schemes the p function is designed to allow partial message recovery. The verification $r \overset{?}{=} \mathsf{p}(R)$ also extracts the recovered message $\bar{m}$.

7.3.2.2 DL-groups

DL-based signature schemes do computations in a cyclic group $\langle G \rangle$ of known order q and known generator G. Multiplying or taking inverses of elements of the group should be easy, but elements of $\langle G \rangle$ might be indistinguishable from elements of a larger set $\mathbb{G}$. [4]

Usually $\langle G \rangle$ is a cyclic subgroup of the multiplicative group $(\mathbb{Z}/p\mathbb{Z})^\times$ of integers modulo p, or an elliptic curve subgroup, and q is a prime number.

The exponentiation is a bijection from $\mathbb{Z}/q\mathbb{Z}$ to $\langle G \rangle$ defined by $k \mapsto G^k$, and the discrete logarithm is the inverse of that bijection. By definition, computing the discrete logarithm in a DL-group is intractable.

$(\mathbb{Z}/q\mathbb{Z})^\times$ is the set of invertible elements of $\mathbb{Z}/q\mathbb{Z}$, and for any $k \in (\mathbb{Z}/q\mathbb{Z})^\times$ the group element G^k is a generator of $\langle G \rangle$. One must know the factorisation of q to compute inverses in $(\mathbb{Z}/q\mathbb{Z})^\times$.

Let $\#q$ be an integer smaller than $\log_2 q$ and let $[\mathbb{Z}/q\mathbb{Z}]_\#$ be the subset of $\mathbb{Z}/q\mathbb{Z}$ that contains the integers smaller than $2^{\#q}$. Then $[\mathbb{Z}/q\mathbb{Z}]_\#$ form a group under the operation $\oplus$ corresponding to the XOR of bit strings of length $\#q$.

[4] Shoup [584, Sect. 13] defines the more complete notion of "abstract group" $(\mathcal{H}, \mathcal{G}, \mathbf{g}, \mu, \nu, \mathcal{E}, \mathcal{D}, \mathcal{E}', \mathcal{D}')$. His $\mathcal{G}$ is a cyclic group generated by $\mathbf{g}$: it is $\langle G \rangle$ with our notation. His $\mathcal{H}$ is a group that contains $\mathcal{G}$: it is $\mathbb{G}$ with our notation. His μ is our q and his ν is the index of $\mathcal{G}$ in $\mathcal{H}$. His $\mathcal{E}$ and $\mathcal{D}$ define bijective encodings of the elements of $\mathcal{H}$ to octet strings. His $\mathcal{E}'$ is a partial encoding, and corresponds to what we call a "projection".

7.3.2.3 Hash function

The function $\mathsf{H} : \mathcal{M} \times \mathcal{R} \to \mathcal{H}$ should be easy to compute, should have uniform random output for random m and may have some of the following properties.

- H is *Type I* if $\forall m \in \mathcal{M}$ and $r, r' \in \mathcal{R}$, $\mathsf{H}(m,r) = \mathsf{H}(m,r')$. This common value is called $\mathsf{H}(m)$.
  H is *Type II* if it is not *Type I*.

- H with Type I is *collision-resistant* if it is hard to find distinct inputs $m \neq m'$ such that $\mathsf{H}(m) = \mathsf{H}(m')$.
- H with Type II is *collision-resistant* if it is hard to find distinct inputs $(m,r) \neq (m',r')$ such that $\mathsf{H}(m,r) = \mathsf{H}(m',r')$.

- H with $\mathcal{H} \subset [\mathbb{Z}/q\mathbb{Z}]_{\#}$ and $\mathcal{R} \subset [\mathbb{Z}/q\mathbb{Z}]_{\#}$ is *xor-collision-resistant* if given random r, r' it is hard to find m, m' such that $\mathsf{H}(m,r) \oplus r = \mathsf{H}(m',r') \oplus r'$. For a type I hash, this is equivalent to preimage-resistance.
- H with $\mathcal{H} \subset \mathbb{Z}/q\mathbb{Z}$ and $\mathcal{R} \subset \mathbb{Z}/q\mathbb{Z}$ is *add-collision-resistant* if given random r, r' it is hard to find m, m' such that $\mathsf{H}(m,r) + r = \mathsf{H}(m',r') + r'$. For a type I hash, this is equivalent to preimage-resistance.
- H with $\mathcal{H} \subset \mathbb{Z}/q\mathbb{Z}$ and $\mathcal{R} \subset (\mathbb{Z}/q\mathbb{Z})^{\times}$ is *div-collision-resistant* if given random r, r' it is hard to find m, m' such that $\mathsf{H}(m,r)/r = \mathsf{H}(m',r')/r'$. For a type I hash, this is equivalent to preimage-resistance.

- H is *suitable as a random oracle* if the knowledge of any number of input-output pairs cannot help to build an algorithm that will compute another input-output pair without doing the computation of H.

Usually one takes a cryptographic hash function such as those studied in Chapter 4. It is likely that these functions have uniform output and have all the variants of collision-resistance mentioned above. However, strictly speaking, none is suitable as a random oracle, because of their extensibility property [127].

7.3.2.4 Projections

The function $\mathsf{p} : \langle G \rangle \to \mathcal{R}$ is easy to compute by the signer, and the verifier should be able to test if $r \overset{?}{=} \mathsf{p}(R)$ for $R \in \langle G \rangle$ and $r \in \mathcal{R}$. Note that if elements of $\langle G \rangle$ are indistinguishable from elements of $\mathbb{G}$, then p is defined on the complete group $\mathbb{G}$. The projection may have some of the following properties.

- p is ε-*almost uniform* if $\forall r \in \mathcal{R}$, $\Pr_{R \in \langle G \rangle}[\mathsf{p}(R) = r] \geq \varepsilon$. This is similar to the *entropy-smoothing* property defined by Shoup [584].
- p is ε-*almost invertible* if there exists an efficient algorithm to compute the function $\mathsf{p}^{-1} : \mathcal{R} \to \mathcal{P}(\langle G \rangle)$ such that
  - $\forall R \in \mathsf{p}^{-1}(r), \mathsf{p}(R) = r$
  - At least a proportion ε of the sets $\mathsf{p}^{-1}(r)$ is non empty.
  - Elements randomly taken from random sets $\mathsf{p}^{-1}(r)$ are indistinguishable from elements randomly taken from $\langle G \rangle$.
- p is $\ell + 1$-*collision-resistant* for $\ell \geq 1$ if it is hard to find distinct $R_0, ..., R_\ell$ such that $\mathsf{p}(R_0) = ... = \mathsf{p}(R_\ell)$.

- p is *suitable as a random oracle* if the knowledge of any number of input-output pairs cannot help to build an algorithm that will compute another input-output pair without doing the computation of p or p^{-1}. Note that an almost invertible function can be suitable as a random oracle (see also Sect. 7.3.2.7).

Examples of projections.

- **Identity projection.** For $\mathbb{G} \subset \mathcal{R}$ it is $\mathsf{p}(R) = R$.
  This function is almost uniform and collision-resistant. It is not almost-invertible if membership of $\langle G \rangle$ is hard to check. It is not suitable as a random oracle.
- **DSA projection.** For $\mathbb{G} = (\mathbb{Z}/p\mathbb{Z})^{\times}$ and $\mathcal{R} = \mathbb{Z}/q\mathbb{Z}$ this is $\mathsf{p}(R) = R \bmod q$.
  This function is almost uniform and probably $\log q$-collision-resistant [123]. It is not almost-invertible if membership of $\langle G \rangle$ is hard to check. It is not suitable as a random oracle.
- **EC projections.** If $\mathbb{G}$ is an elliptic curve defined over some finite field $\mathbb{F}$, let (R_x, R_y) be the coordinates of a point R and $\mathsf{i}_{\mathbb{F}}$ a mapping from $\mathbb{F}$ to the set of integers.
  - **ECxq projection.** For $\mathcal{R} = \mathbb{Z}/q\mathbb{Z}$ it is $\mathsf{p}(R) = \mathsf{i}_{\mathbb{F}}(R_x) \bmod q$.
  - **ECx2 projection.** For $\mathcal{R} = [\mathbb{Z}/q\mathbb{Z}]_{\#}$ it is $\mathsf{p}(R) = \mathsf{i}_{\mathbb{F}}(R_x) \bmod 2^{\#q}$.
  - **ECaddq projection.** For $\mathcal{R} = \mathbb{Z}/q\mathbb{Z}$ it is $\mathsf{p}(R) = \mathsf{i}_{\mathbb{F}}(R_x + R_y) \bmod q$.

  These functions are almost uniform and almost invertible. They are probably $\log q$-collision-resistant. They are not suitable as a random oracle.
- **KCDSA projection.** p is a hash function with output in $\mathcal{R}$, e.g. based on SHA-1 (see Sect. 4.4.2).
  This function is uniform and collision-resistant. It is almost (because of the extensibility property) suitable as a random oracle. It is not almost-invertible.
- **Permuted projection.** Any projection $\mathsf{p}' : \langle G \rangle \to \mathcal{R}$ can be composed with a random permutation $\mathsf{P} : \langle G \rangle \to \langle G \rangle$ to obtain $\mathsf{p} = \mathsf{p}' \circ \mathsf{P}$.
  The projection p inherits the properties of p', but is also suitable as a random oracle.

Projections with partial message recovery. Let $\mathsf{F} : \langle G \rangle \times \bar{\mathcal{M}} \to \mathcal{R}$ and $\mathsf{F}^{-1} : \langle G \rangle \times \mathcal{R} \to \bar{\mathcal{M}} \cup \{\mathit{fail}\}$ such that $\forall R \in \langle G \rangle$, $\mathsf{F}(R, \bar{m}) = r \Leftrightarrow \mathsf{F}^{-1}(R, r) = \bar{m}$. Then the function $\mathsf{p}(R) = \mathsf{F}(R, \bar{m})$ is a projection that allows partial message recovery. The verification $r \overset{?}{=} \mathsf{p}(R)$ is false if, and only if, $\mathsf{F}^{-1}(R, r)$ returns *fail*.

- **PVSSR projection.** This is the composition of an arbitrary encryption function E over $\mathcal{R}$, with a key selected from $\langle G \rangle$, and a redundancy function $\rho : \bar{\mathcal{M}} \to \mathcal{R}$. The definition is $\mathsf{F}(R, \bar{m}) = \mathsf{E}_R \circ \rho(\bar{m})$ and $\mathsf{F}^{-1}(R, r) = \rho^{-1} \circ \mathsf{E}_R^{-1}(r)$.
  The redundancy function ρ should have the following properties:
  - it is collision resistant,
  - the inverse $\rho^{-1} : \mathcal{R} \to \bar{\mathcal{M}} \cup \{\mathit{fail}\}$ is easy to compute.
  - a random element $\tilde{m} \in \mathcal{R}$ is very unlikely to be the image of some $\bar{m} \in \bar{\mathcal{M}}$,

  If E is a secure cipher, then the projection is uniform, collision-resistant and suitable as a random oracle, but is not almost invertible.

- **Group projection.** This is a special case of the PVSSR projection, based on any other projection $\mathsf{p}' : \langle G \rangle \to \mathcal{G}$ such that $\mathcal{G}$ is a group with an action on $\mathcal{R}$. This defines a one-time-pad encryption scheme and the projection is $\mathsf{p}(R) = \mathsf{p}'(R) \cdot \rho(m)$.
  This projection has the same properties as p'.
- **NS projection.** This is the Group projection where $\mathcal{G} = \mathcal{R} = \mathbb{Z}/q\mathbb{Z}$ with additive action. It is defined by the equation $\mathsf{p}(R) = \mathsf{p}'(R) + \rho(m) \bmod q$.
  This projection has the same properties as p'.
- **NR projection.** This is the Group projection where $\mathbb{G} = \mathcal{G} = (\mathbb{Z}/p\mathbb{Z})^\times$ has a multiplicative action on $\mathcal{R} = \mathbb{Z}/p\mathbb{Z}$, and with a tweak of the Identity projection $\mathsf{p}'(R) = R^{-1} \bmod p$. The NR projection is defined by the equation $\rho(m) = R \cdot \mathsf{p}(R) \bmod p$.

7.3.2.5 Categories

Definition and properties. The category is described by the sets $\mathcal{V}$, $\mathcal{K}$ and $\mathcal{S}$, subsets of $\mathbb{Z}/q\mathbb{Z}$, the sets $\mathcal{H}$ and $\mathcal{R}$, the two functions ϕ and $\psi : \mathcal{H} \times \mathcal{R} \times \mathcal{S} \to \mathbb{Z}/q\mathbb{Z}$ and a set $\mathcal{I} \subset \mathcal{H} \times \mathcal{R} \times \mathcal{V} \times \mathcal{K}$.

A category should meet some of the following properties.

- **Other functions.** Let $\mathcal{A}$ be the set of possible outputs for ϕ and $\mathcal{B}$ the set of possible outputs for ψ. Five additional functions σ, λ_h, λ_s, λ_r, μ_h can be defined, and for each of these functions there exists an efficient algorithm that computes the result.
  - $\sigma : \mathcal{I} \to \mathcal{S}$.
  - $\lambda_h : \mathcal{A} \times \mathcal{B} \times \mathcal{R} \to \mathcal{H}$
  - $\lambda_s : \mathcal{A} \times \mathcal{B} \times \mathcal{R} \to \mathcal{S}$
  - $\lambda_r : \mathcal{A} \times \mathcal{B} \times \mathcal{H} \to \mathcal{R}$
  - $\mu_h : \mathcal{S} \times \mathcal{R} \times \mathcal{V} \times \mathcal{K} \to \mathcal{H}$
- **Main properties.** These properties are mandatory for all DL-based schemes.
  ($\mathfrak{m}1$) For all $(h, r, v, k) \in \mathcal{I}$, the value $s = \sigma(h, r, v, k)$ is such that if $\alpha = \phi(h, r, s)$ and $\beta = \psi(h, r, s)$ then $k = \alpha + v \cdot \beta$.
  ($\mathfrak{m}2$) For all $v \in \mathcal{V}$ and $h \in \mathcal{H}$, $\Pr_{r \in \mathcal{R},\, k \in \mathcal{K}}[(h, r, v, k) \in \mathcal{I}] \geq \varepsilon_{\mathfrak{m}}$.

  Property ($\mathfrak{m}1$) implies that all signatures generated by the signing algorithm are valid. Property ($\mathfrak{m}2$) implies that the expected number of random values for k needed to generate a signature is less than $\frac{1}{\varepsilon_{\mathfrak{m}}}$.
- **Other properties.**
  ($\mathfrak{o}1$) For all $(h, r, s) \in \mathcal{H} \times \mathcal{R} \times \mathcal{S}$ the equation $\lambda_h(\phi(h, r, s), \psi(h, r, s), r) = h$ holds.
  ($\mathfrak{o}2$) For all $(h, r, s) \in \mathcal{H} \times \mathcal{R} \times \mathcal{S}$ the equation $\lambda_s(\phi(h, r, s), \psi(h, r, s), r) = s$ holds.
  ($\mathfrak{o}3$) The function $s \mapsto \mu_h(s, r, v, k)$ is the inverse of $h \mapsto \sigma(h, r, v, k)$.
- **Additional properties for security with idealised p.**
  ($\mathfrak{p}1$) For fixed (h, r, v) and uniform k such that $(h, r, v, k) \in \mathcal{I}$ the value $\sigma(h, r, v, k)$ is uniform in $\mathcal{S}$.

Note that this property together with the hypothesis that the function $k \mapsto \mathsf{p}(G^k)$ is (almost-)uniform and one-way implies that the set of possible valid appendices (r, s) for a message m is uniformly distributed.

($\mathfrak{p}2$) For fixed $h \in \mathcal{H}$ and $v \in \mathcal{V}$ and uniformly random $s \in \mathcal{S}$ and $r \in \mathcal{R}$, the value $k = \phi(h, r, s) + v \cdot \psi(h, r, s)$ is uniformly random in $\mathcal{K}$.
Note that failure may happen if $\mathcal{I} \neq \mathcal{H} \times \mathcal{R} \times \mathcal{V} \times \mathcal{K}$ and if $k \notin \mathcal{K}$. It is accepted that the probability of this failure is negligible.

($\mathfrak{p}3$) Given random r and r', it is hard to find some (α, β) and messages m and m' such that $\lambda_h(\alpha, \beta, r) = \mathsf{H}(m, r)$ and $\lambda_h(\alpha, \beta, r') = \mathsf{H}(m', r')$.
Note that for a type I hash, this property is usually equivalent to the preimage-resistance of H.

- **Additional properties for security with idealised H.**
  ($\mathfrak{h}1$) If $h = \lambda_h(\alpha, \beta, r)$ and $s = \lambda_s(\alpha, \beta, r)$, then $\alpha = \phi(h, r, s)$ and $\beta = \psi(h, r, s)$.
  ($\mathfrak{h}2$) $\Pr_{\alpha \in \mathcal{A}, \beta \in \mathcal{B}}[\lambda_h(\alpha, \beta, \mathsf{p}(G^\alpha V^\beta)) \in \mathcal{H} \text{ and } \lambda_s(\alpha, \beta, \mathsf{p}(G^\alpha V^\beta)) \in \mathcal{S}] \geq \varepsilon_{\mathfrak{h}}$.
- **Additional properties for security with idealised $\langle G \rangle$.**
  ($\mathfrak{g}1$) For all (h, r, s) the equation $\lambda_r(\phi(h, r, s), \psi(h, r, s), h) = r$ holds.
  ($\mathfrak{g}2$) For any (h, h', r, s), if $\lambda_r(\phi(h, r, s), \psi(h, r, s), h') = r$ then $h' = h$.

Simple categories. These are the categories where $\mathcal{H} \subset \mathbb{Z}/q\mathbb{Z}$ and $\mathcal{R} \subset \mathbb{Z}/q\mathbb{Z}$ and where each of ϕ and ψ only does one operation in $\mathbb{Z}/q\mathbb{Z}$. These are less general than Meta-ElGamal [298] or TEGTSS [123] schemes, but cover all actual published schemes.

Examples. These are taken from the literature. Properties ($\mathfrak{m}1$), ($\mathfrak{m}2$), ($\mathfrak{o}1$), ($\mathfrak{o}2$), ($\mathfrak{o}3$), ($\mathfrak{p}1$), ($\mathfrak{p}2$), ($\mathfrak{h}1$) and ($\mathfrak{h}2$) hold for all these examples.

- **ElGamal category.** Let $\mathcal{H} \subset \mathbb{Z}/q\mathbb{Z}$, $\mathcal{R} = \mathcal{V} = \mathcal{K} = \mathcal{S} = \mathcal{B} = (\mathbb{Z}/q\mathbb{Z})^\times$ and $\mathcal{A} = \mathbb{Z}/q\mathbb{Z}$. Because $\mathcal{I} = \{(h, r, v, k) | h + v \cdot r \in (\mathbb{Z}/q\mathbb{Z})^\times\}$ property ($\mathfrak{p}2$) can fail with negligible probability. Property ($\mathfrak{p}3$) is equivalent to div-collision-resistance of H. Properties ($\mathfrak{g}1$) and ($\mathfrak{g}2$) hold with the restrictions $\mathcal{H} \subset (\mathbb{Z}/q\mathbb{Z})^\times$ and $\mathcal{A} = (\mathbb{Z}/q\mathbb{Z})^\times$. Properties ($\mathfrak{m}2$) and ($\mathfrak{h}2$) hold with $\varepsilon_{\mathfrak{m}} = \frac{\varphi(q)}{q}$ and $\varepsilon_{\mathfrak{h}} = \frac{|\mathcal{H}|}{q}$.

$$\begin{aligned} \phi(h, r, s) &= h/s & \lambda_h(\alpha, \beta, r) &= \alpha\beta^{-1} \cdot r \\ \psi(h, r, s) &= r/s & \lambda_s(\alpha, \beta, r) &= \beta^{-1} \cdot r \\ \sigma(h, r, v, k) &= (h + v \cdot r)/k & \lambda_r(\alpha, \beta, h) &= \alpha^{-1}\beta \cdot h \\ \mu_h(s, r, v, k) &= k \cdot s - v \cdot r & & \end{aligned}$$

- **Inverse ElGamal category.** Let $\mathcal{H} \subset \mathbb{Z}/q\mathbb{Z}$, $\mathcal{R} = \mathcal{V} = \mathcal{K} = \mathcal{S} = \mathcal{B} = (\mathbb{Z}/q\mathbb{Z})^\times$ and $\mathcal{A} = \mathbb{Z}/q\mathbb{Z}$. Because $\mathcal{I} = \{(h, r, v, k) | h + v \cdot r \in (\mathbb{Z}/q\mathbb{Z})^\times\}$ property ($\mathfrak{p}2$) can fail with negligible probability. Property ($\mathfrak{p}3$) is equivalent to div-collision-resistance of H. Properties ($\mathfrak{g}1$) and ($\mathfrak{g}2$) hold with the restrictions $\mathcal{H} \subset (\mathbb{Z}/q\mathbb{Z})^\times$ and $\mathcal{A} = (\mathbb{Z}/q\mathbb{Z})^\times$. Properties ($\mathfrak{m}2$) and ($\mathfrak{h}2$) hold with $\varepsilon_{\mathfrak{m}} = \frac{\varphi(q)}{q}$ and $\varepsilon_{\mathfrak{h}} = \frac{|\mathcal{H}|}{q}$.

$$\begin{array}{ll}
\phi(h,r,s) = h \cdot s & \lambda_h(\alpha,\beta,r) = \alpha\beta^{-1} \cdot r \\
\psi(h,r,s) = r \cdot s & \lambda_s(\alpha,\beta,r) = r^{-1} \cdot \beta \\
\sigma(h,r,v,k) = k/(h + v \cdot r) & \lambda_r(\alpha,\beta,h) = \alpha^{-1}\beta \cdot h \\
\mu_h(s,r,v,k) = k/s - v \cdot r &
\end{array}$$

- **GOST category.** Let $\mathcal{H} \subset (\mathbb{Z}/q\mathbb{Z})^\times$, $\mathcal{V} = \mathcal{K} = \mathcal{S} = \mathcal{A} = \mathbb{Z}/q\mathbb{Z}$ and $\mathcal{R} = \mathcal{B} = (\mathbb{Z}/q\mathbb{Z})^\times$. Property $(\mathfrak{o}3)$ needs the restriction $\mathcal{K} = (\mathbb{Z}/q\mathbb{Z})^\times$. Property $(\mathfrak{p}3)$ is equivalent to div-collision-resistance of H. Properties $(\mathfrak{g}1)$ and $(\mathfrak{g}2)$ hold. Properties $(\mathfrak{m}2)$ and $(\mathfrak{h}2)$ hold with $\varepsilon_\mathfrak{m} = 1$ and $\varepsilon_\mathfrak{h} = \frac{|\mathcal{H}|}{q}$.

$$\begin{array}{ll}
\phi(h,r,s) = s/h & \lambda_h(\alpha,\beta,r) = \beta^{-1} \cdot r \\
\psi(h,r,s) = r/h & \lambda_s(\alpha,\beta,r) = \alpha\beta^{-1} \cdot r \\
\sigma(h,r,v,k) = k \cdot h - v \cdot r & \lambda_r(\alpha,\beta,h) = \beta \cdot h \\
\mu_h(s,r,v,k) = (s + v \cdot r)/k &
\end{array}$$

- **GDSA category.** Let $\mathcal{H} \subset \mathbb{Z}/q\mathbb{Z}$, $\mathcal{K} = \mathcal{S} = \mathcal{A} = \mathcal{B} = \mathbb{Z}/q\mathbb{Z}$ and $\mathcal{R} = \mathcal{V} = (\mathbb{Z}/q\mathbb{Z})^\times$. Property $(\mathfrak{p}3)$ is equivalent to div-collision-resistance of H. Properties $(\mathfrak{g}1)$ and $(\mathfrak{g}2)$ hold with the restrictions $\mathcal{H} \subset (\mathbb{Z}/q\mathbb{Z})^\times$ and $\mathcal{A} = (\mathbb{Z}/q\mathbb{Z})^\times$. Properties $(\mathfrak{m}2)$ and $(\mathfrak{h}2)$ hold with $\varepsilon_\mathfrak{m} = 1$ and $\varepsilon_\mathfrak{h} = \frac{|\mathcal{H}|}{q}$.

$$\begin{array}{ll}
\phi(h,r,s) = h/r & \lambda_h(\alpha,\beta,r) = \alpha \cdot r \\
\psi(h,r,s) = s/r & \lambda_s(\alpha,\beta,r) = \beta \cdot r \\
\sigma(h,r,v,k) = (k \cdot r - h)/v & \lambda_r(\alpha,\beta,h) = \alpha^{-1} \cdot h \\
\mu_h(s,r,v,k) = k \cdot r - v \cdot s &
\end{array}$$

- **KCDSAadd category.** Let $\mathcal{H} \subset \mathbb{Z}/q\mathbb{Z}$, $\mathcal{R} = \mathcal{K} = \mathcal{S} = \mathcal{A} = \mathcal{B} = \mathbb{Z}/q\mathbb{Z}$ and $\mathcal{V} = (\mathbb{Z}/q\mathbb{Z})^\times$. Property $(\mathfrak{p}3)$ is equivalent to add-collision-resistance of H. Properties $(\mathfrak{g}1)$ and $(\mathfrak{g}2)$ hold. Properties $(\mathfrak{m}2)$ and $(\mathfrak{h}2)$ hold with $\varepsilon_\mathfrak{m} = 1$ and $\varepsilon_\mathfrak{h} = \frac{|\mathcal{H}|}{q}$.

$$\begin{array}{ll}
\phi(h,r,s) = h + r & \lambda_h(\alpha,\beta,r) = \alpha - r \\
\psi(h,r,s) = s & \lambda_s(\alpha,\beta,r) = \beta \\
\sigma(h,r,v,k) = (k - (h + r))/v & \lambda_r(\alpha,\beta,h) = \alpha - h \\
\mu_h(s,r,v,k) = (k - v \cdot s) - r &
\end{array}$$

- **KCDSAxor category.** Let $\mathcal{H} = \mathcal{R} = \mathcal{A} = [\mathbb{Z}/q\mathbb{Z}]_\#$, $\mathcal{K} = \mathcal{S} = \mathcal{B} = \mathbb{Z}/q\mathbb{Z}$ and $\mathcal{V} = (\mathbb{Z}/q\mathbb{Z})^\times$. Property $(\mathfrak{p}3)$ is equivalent to xor-collision-resistance of H. Properties $(\mathfrak{g}1)$ and $(\mathfrak{g}2)$ hold. Properties $(\mathfrak{m}2)$ and $(\mathfrak{h}2)$ hold with $\varepsilon_\mathfrak{m} = 1$ and $\varepsilon_\mathfrak{h} = 1$.

$$\begin{array}{ll}
\phi(h,r,s) = h \oplus r & \lambda_h(\alpha,\beta,r) = \alpha \oplus r \\
\psi(h,r,s) = s & \lambda_s(\alpha,\beta,r) = \beta \\
\sigma(h,r,v,k) = (k - (h \oplus r))/v & \lambda_r(\alpha,\beta,h) = \alpha \oplus h \\
\mu_h(s,r,v,k) = (k - v \cdot s) \oplus r &
\end{array}$$

- **Schnorr category.** Let $\mathcal{H} \subset \mathbb{Z}/q\mathbb{Z}$, $\mathcal{V} = \mathcal{K} = \mathcal{S} = \mathcal{A} = \mathbb{Z}/q\mathbb{Z}$ and $\mathcal{B} = \mathcal{H}$. The variable r is not used and is taken from an arbitrary set $\mathcal{R}$. Property

$(\mathfrak{p}3)$ is implied by the collision-resistance of H. Properties $(\mathfrak{g}1)$ and $(\mathfrak{g}2)$ do not hold because λ_r cannot be defined. Property $(\mathfrak{o}3)$ needs the restriction $\mathcal{V} = (\mathbb{Z}/q\mathbb{Z})^\times$. Properties $(\mathfrak{m}2)$ and $(\mathfrak{h}2)$ hold with $\varepsilon_\mathfrak{m} = 1$ and $\varepsilon_\mathfrak{h} = 1$.

$$\begin{array}{ll}
\phi(h,r,s) = s & \lambda_h(\alpha,\beta,r) = \beta \\
\psi(h,r,s) = h & \lambda_s(\alpha,\beta,r) = \alpha \\
\sigma(h,r,v,k) = k - v \cdot h & \\
\mu_h(s,r,v,k) = (k-s)/v &
\end{array}$$

- **Swapped-Schnorr category.** Let $\mathcal{H} \subset \mathbb{Z}/q\mathbb{Z}$, $\mathcal{K} = \mathcal{S} = \mathcal{B} = \mathbb{Z}/q\mathbb{Z}$, $\mathcal{V} = (\mathbb{Z}/q\mathbb{Z})^\times$ and $\mathcal{A} = \mathcal{H}$. The variable r is not used and is taken from an arbitrary set $\mathcal{R}$. Property $(\mathfrak{p}3)$ is implied by the collision-resistance of H. Properties $(\mathfrak{g}1)$ and $(\mathfrak{g}2)$ do not hold because λ_r cannot be defined. Properties $(\mathfrak{m}2)$ and $(\mathfrak{h}2)$ hold with $\varepsilon_\mathfrak{m} = 1$ and $\varepsilon_\mathfrak{h} = 1$.

$$\begin{array}{ll}
\phi(h,r,s) = h & \lambda_h(\alpha,\beta,r) = \alpha \\
\psi(h,r,s) = s & \lambda_s(\alpha,\beta,r) = \beta \\
\sigma(h,r,v,k) = (h-k)/v & \\
\mu_h(s,r,v,k) = v \cdot s + k &
\end{array}$$

7.3.2.6 Examples of published signature schemes

- The ElGamal scheme [219] is defined on the multiplicative group $(\mathbb{Z}/p\mathbb{Z})^\times$, with a slight variant of the ElGamal category (where $-r$ replaces r), the identity projection and a type I hash.
- The DSA scheme [464] is defined on a prime order subgroup of the multiplicative group $(\mathbb{Z}/p\mathbb{Z})^\times$, with the ElGamal category, the DSA projection and a type I hash.
- The ECDSA scheme [319] is defined on a prime order elliptic curve subgroup with the ElGamal category, the ECxq projection and a type I hash.
- The GOST 34.10 scheme [445] is defined on a prime order multiplicative subgroup of $\mathbb{Z}/p\mathbb{Z}$, with a slight variant of the GOST category (where $-r$ replaces r), the DSA projection and a type I hash.
- The KCDSA scheme [340] is defined on a prime order multiplicative subgroup of $\mathbb{Z}/p\mathbb{Z}$ or on a prime order elliptic curve subgroup, with the KCDSAxor category, the KCDSA projection and a type I hash, where some certification data is hashed together with the message.
- The ECGDSA scheme [13] is defined on a prime order elliptic curve subgroup, with the GDSA category, the ECxq projection and a type I hash.
- The DSA-II scheme [123] is defined on a prime order multiplicative subgroup of $\mathbb{Z}/p\mathbb{Z}$, with the ElGamal category, the KCDSA projection and a type II hash.
- The ECDSA-II scheme [412] is defined on a prime order elliptic curve subgroup, with the ElGamal category, the ECxq projection and a type II hash.
- The ECDSA-III scheme [412] is defined on a prime order elliptic curve subgroup, with the ElGamal category, the ECaddq projection and a type II hash.
- The Schnorr scheme [559] is defined on a prime order multiplicative subgroup of $\mathbb{Z}/p\mathbb{Z}$, with a slight variant of the Schnorr category (where $-h$ replaces h), the identity projection and a type II hash.

- The GPS-sign scheme [263] is defined on a subgroup of $\mathbb{Z}/n\mathbb{Z}$ with a variant of the Schnorr category (where $\mathcal{S} = \mathbb{Z}$), the identity projection and a type II hash. The composite modulus n is viewed as part of the public key and not as a scheme parameter.
- The Nyberg-Rueppel scheme [489,490] is a scheme with total message recovery, and hence no variable m. It is defined on a prime order multiplicative subgroup of $\mathbb{Z}/p\mathbb{Z}$, with the Schnorr category, the NR projection and the type II hash defined by $\mathsf{H}(r) = r \bmod q$.
- The PVSSR scheme (Pintsov-Vanstone Signature Scheme with message Recovery [519]) is defined on a prime order elliptic curve subgroup with a slight variant of the Schnorr category (where $-h$ replaces h), the PVSSR projection and a type II hash.
- The Naccache-Stern scheme [455] is defined on a prime order elliptic curve subgroup with the ElGamal category, the NS projection based on ECxq projection and a type I hash.
- The Abe-Okamoto scheme [5] is a scheme with total message recovery, and hence no variable m. It is defined on a prime order elliptic curve subgroup with the Schnorr category, the xor variant of the NS projection based on ECx2 projection and the type II hash $\mathsf{H}(r)$.

7.3.2.7 Initial results to be used in the security proofs

The random oracle model for almost invertible functions. The random oracle model builds an oracle for a one-way function f, that answers queries for $\mathsf{f}(x)$ with some uniformly distributed value y. Suitable functions have uniform output, are collision-resistant, etc.

If the function f is almost invertible, then the random oracle model should also allow queries for $\mathsf{f}^{-1}(x)$. Many results that were proven for the original random oracle model are also valid for this model.

The forking lemma. This lemma is found e.g. in [523] and is a tool for proofs of security in the random oracle model. The lemma holds when the scheme has the following property: each forgery can be linked to a unique "critical" query to the random oracle. The critical query is an input x such that knowing $x \overset{\mathsf{f}}{\mapsto} y$ is necessary to check if the forgery is valid.

The forking lemma also holds in the random oracle model for an almost invertible function, if the critical query hypothesis holds.

Let q_S be the number of signature queries, q_H the number of oracle queries, n_H the number of possible outputs for the random oracle and ε the probability that the forger outputs a valid forgery.

Lemma 7.1. (Forking lemma). *There exist constants c_0 and c_1 such that if $\varepsilon \geq c_0/n_H$ then after an expected number of $c_1 \cdot q_H/\varepsilon$ replays of the simulation with different choices for the random oracle, one can obtain (with some probability ε') another forgery with the same critical query but having another uniform random answer.*

In [523, *Lemma 8*] we have $c_0 = 7q_H$, $c_1 = 2(7 + \frac{1}{q_H})$ and $\varepsilon' = 3/25$ and also [523, *Theorem 10*] $c_0 = 7q_H$, $c_1 = 84480$ and $\varepsilon' \simeq 1$.

The improved forking lemma. This lemma is an extension of the forking lemma that is found in [123] and is used together with $\ell + 1$-collision-resistant functions.

Lemma 7.2. (Improved forking lemma). *There exist constants c_0 and c_1 such that if $\varepsilon \geq c_0/n_H$ then after an expected number of $c_1 \cdot q_H/\varepsilon$ replays of the simulation with different choices for the random oracle, one can obtain (with some probability ε') ℓ other forgeries with the same critical query but having other uniform random answers.*

In [123, *Lemma 10*] we have $c_0 = 4$, $c_1 = 24\ell \log(2\ell) + \frac{1}{q_H}$ and $\varepsilon' = 1/96$.

Unique representation.

Lemma 7.3. (Unique representation). *If the discrete logarithm of $V \in \langle G \rangle$ is hard to compute and if two representations $R = G^{\alpha}V^{\beta}$ and $R = G^{\alpha'}V^{\beta'}$ can be computed then $\alpha = \alpha'$ and $\beta = \beta'$.*

Proof. This is proven by $(\alpha - \alpha') = (\beta' - \beta) \cdot \log V$. □

7.3.2.8 Security proof with idealised p

This proof is based on one of the results from [123]. In this proof, H may be a Type I or Type II hash function, and p may be almost invertible.

Theorem 7.9. *A DL-based signature scheme is existentially unforgeable (and non-malleable) under adaptive chosen message attacks if the discrete logarithm is hard, if H is collision-resistant, if p is a random oracle and if the category has properties ($\mathfrak{o}1$), ($\mathfrak{o}2$), ($\mathfrak{p}2$), ($\mathfrak{p}1$), and ($\mathfrak{p}3$). The security reduction is loose.*

Proof. To answer a signature query for m, the simulator generates a random r and a random s, and computes $h = \mathsf{H}(m, r)$ and $R = G^{\phi(h,r,s)}V^{\psi(h,r,s)}$. With property ($\mathfrak{p}2$), the value R is uniformly distributed and with property ($\mathfrak{p}1$) the value s has the same distribution as for the signing algorithm. The simulator sets the oracle table $\mathsf{p}(R) := r$. The signed message is (m, r, s).

p-oracle queries that were not defined by a signature query are answered with a random value. If p is almost invertible, then p^{-1}-oracle queries are answered with some $R = G^{\alpha'}V^{\beta'}$ for random α' and β'. The probability that the oracle table cannot be set is the probability of a collision in R, which is low if $(q_H + q_S)^2 \leq q$.

When the forger outputs its forgery (m, r, s), the critical query is the value $R = G^{\alpha}V^{\beta}$ where $\alpha = \phi(h, r, s)$, $\beta = \psi(h, r, s)$ and $h = \mathsf{H}(m, r)$.

Let us suppose that the critical query was part of a signature query for some m' that answered $(m', r', s') \neq (m, r, s)$. We define $h' = \mathsf{H}(m', r')$, $\alpha' = \phi(h', r', s')$ and $\beta = \psi(h', r', s')$. Validity of the signature means that $R = G^{\alpha'}V^{\beta'}$, and the unique representation of R implies $\alpha' = \alpha$ and $\beta' = \beta$. We also have $r' = r = \mathsf{p}(R)$. Property ($\mathfrak{o}1$) implies $h = \lambda_h(\alpha, \beta, r) = h'$ and property ($\mathfrak{o}2$) implies $s = \lambda_s(\alpha, \beta, r) = s'$. Therefore $(m', r', s') \neq (m, r, s)$ implies $m' \neq m$. Since $\mathsf{H}(m', r') = \mathsf{H}(m, r)$ we have found a collision in H.

Let us suppose that the critical query was a p-oracle query for R. The forking lemma allows us to find another forgery $(m', r', s') \neq (m, r, s)$ with the same critical R but a different oracle for p. The unique representation of R implies $\alpha' = \alpha$ and $\beta' = \beta$. Therefore the simulator can find α, β, m and m' such that $\lambda_h(\alpha, \beta, r) = \mathsf{H}(m, r)$ and $\lambda_h(\alpha, \beta, r') = \mathsf{H}(m', r')$ for random r and r', which is intractable if ($\mathfrak{p}3$) holds.

Let us suppose that the critical query was a p^{-1}-oracle query that returned $R = G^{\alpha'} V^{\beta'}$. The unique representation of R implies $\alpha' = \alpha$ and $\beta' = \beta$, which is very unlikely because α' and β' were kept secret. □

7.3.2.9 Security proof with idealised H of type II

This proof is based on one of the results from [123]. In this proof, H is a type II hash function.

Theorem 7.10. *A DL-based signature scheme is existentially unforgeable under adaptive chosen message attacks if the discrete logarithm is hard, if* H *is a random oracle with large output set, if* p *is almost uniform and* $\ell + 1$*-collision-resistant and if the category has properties* ($\mathfrak{o}1$)*,* ($\mathfrak{o}2$)*,* ($\mathfrak{h}1$)*, and* ($\mathfrak{h}2$)*. Collision-resistance of* p *also implies non-malleability. The security reduction is loose.*

Proof. To answer a signature query, the simulator generates random $\alpha \in \mathcal{A}$ and $\beta \in \mathcal{B}$ and computes $R = G^\alpha V^\beta$, $r = \mathsf{p}(R)$, $h = \lambda_h(\alpha, \beta, r)$ and $s = \lambda_s(\alpha, \beta, r)$, until $h \in \mathcal{H}$ and $s \in \mathcal{S}$. This is equivalent to using the signature generation algorithm with $k = \alpha + v \cdot \beta$, and therefore this simulation has the same output distribution. Property ($\mathfrak{h}2$) says that the expected number of random α, β needed is less than $\frac{1}{\varepsilon_\mathfrak{h}}$. The simulator sets the oracle table $\mathsf{H}(m, r) := h$. The signed message is (m, r, s).

Oracle queries that were not defined by a signature query are answered with a random value. The value of $R = G^\alpha V^\beta$ is uniformly distributed for random α and β. If p is $\frac{1}{n}$-almost uniform then the probability that the oracle table cannot be set is bounded by the probability of a collision in r, which is low if $(q_H + q_S)^2 \leq n$.

When the forger outputs its forgery (m, r, s), the critical query is the H-oracle query of (m, r).

Let us suppose that the critical query was part of a signature query. This signature query returned a valid (m, r, s') with the same oracle. Therefore $h' = h$. If p is collision-resistant, then $R = R'$ and its unique representation implies $\alpha' = \alpha$ and $\beta' = \beta$, so property ($\mathfrak{o}2$) implies $s' = s$.

Let us suppose that the critical query was an oracle query for (m, r). The improved forking lemma allows us to find ℓ other forgeries (m, r, s_i) with the same critical (m, r) but different oracles for H. Since all $\mathsf{p}(R_i) = r$, the $\ell + 1$ collision-resistance of p implies that there exists a pair where $R_i = R_j$. Unique representation implies $\alpha_i = \alpha_j$ and $\beta_i = \beta_j$. Property ($\mathfrak{o}1$) implies a unique possible value $h_i = h_j$, which is unlikely to be the one given by the two different oracles for H, because the output set is large. □

7.3.2.10 Security proof with idealised $\langle G \rangle$

The generic group model was introduced by Shoup [581] and extended by Brown [126] to prove the security of ECDSA.

Theorem 7.11. *A DL-based signature scheme is existentially unforgeable under adaptive chosen message attacks in the generic group model if* H *is uniform, one-way and collision-resistant, if* p *is almost uniform and almost invertible and if the category has properties* $(\mathfrak{g}1)$ *and* $(\mathfrak{g}2)$*. The security reduction is tight.*

Proof. We don't include in this document the proof given in [126] but we show below how it can be adapted to other schemes than ECDSA, using our general framework.

The proof was written for a type I hash function, but it also works for a type II hash. It was written for ElGamal category, but it works for all categories with properties $(\mathfrak{g}1)$ and $(\mathfrak{g}2)$.

- In [Table 1] step 3 of **Hint** is replaced with
  $s_{m+1} = z_1 \cdot \sigma(h_{m+1}, \mathsf{p}(A_{m+1}), z_2 z_1^{-1}, z_{m+1})$.
- In [Table 2] steps 1 and 2 of **Hint** are replaced with
  $C_{(m+1)1} = \phi(h_{m+1}, \mathsf{p}(A_{m+1}), s_{m+1})$ and
  $C_{(m+1)2} = \psi(h_{m+1}, \mathsf{p}(A_{m+1}), s_{m+1})$.
- In [Table 4] step 2.b should use $\mathsf{p}^{-1}(\lambda_r(C_{i1}, C_{i2}, e))$.
- In [Table 7] step 1.b.iii should use $\mathsf{p}^{-1}(\lambda_r(C_{i1}, C_{i2}, \hat{e}_i))$.

Property $(\mathfrak{g}2)$ is used when the proof shows that
$r = ... = \lambda_r(C_{m1}, C_{m2}, \hat{e}_i) = \lambda_r(\phi(\mathsf{H}(m), r, s_m), \psi(\mathsf{H}(m), r, s_m), \hat{e}_i)$
and then deduces that $\hat{e}_i = \mathsf{H}(m)$. □

7.3.2.11 Security proof with idealised H of type I

A proof where the scheme is seen as an FDH scheme. Under some conditions, DL-based schemes can easily fit into the hash-then-invert paradigm.

Theorem 7.12. *A type I DL-based signature scheme is existentially unforgeable under* ***single-occurrence*** *chosen message attacks if* H *is a random oracle,* p *is invertible, the category has properties* $(\mathfrak{h}1)$ *and* $(\mathfrak{o}3)$*, and the following problem is intractable: given* $h \in \mathcal{H}$*, finding* $(r, s) \in \mathcal{R} \times \mathcal{S}$ *such that* $r = \mathsf{p}(G^{\phi(h,r,s)} V^{\cdot \psi(h,r,s)})$.

Proof. The FDH scheme based on the function $\mathsf{f}(r \| s) = \mu_h(s, r, v, \log_G(\mathsf{p}^{-1}(r)))$, is exactly the Type I scheme based on this category and projection p.

- f is not efficiently computable if the discrete logarithm is hard.
- The test function $\mathsf{T}^{\mathsf{f}}(r \| s, h) = \big(r \overset{?}{=} \mathsf{p}(G^{\phi(h,r,s)} V^{\psi(h,r,s)})\big)$ is efficiently computable from the public information.
- The simulation function takes random $\alpha \in \mathcal{A}$, $\beta \in \mathcal{B}$ and computes $\mathsf{S}^{\mathsf{f}}(\alpha, \beta) = (r \| s, h)$ with $r = \mathsf{p}(G^{\alpha} V^{\beta})$, $s = \lambda_s(\alpha, \beta, r)$ and $h = \lambda_h(\alpha, \beta, r)$.
- The randomised inverse $\mathsf{f}^{-1}(k, h) = \mathsf{p}(G^k) \| \sigma(h, \mathsf{p}(G^k), v, k)$ is efficiently computable with the trapdoor.

The function f is verifiable simulatable trapdoor and its preimage-resistance is based on the intractability of the following problem: given $h \in \mathcal{H}$, finding $(r,s) \in \mathcal{R} \times \mathcal{S}$ such that $r = \mathsf{p}(G^{\phi(h,r,s)} V^{\cdot\psi(h,r,s)})$. This problem is provably as hard as the discrete logarithm in $\langle G \rangle$ if p is replaced with a random oracle. This result is weaker than the previous ones, because two components need to be simultaneously idealised.

Non-malleability of the scheme is also equivalent to second-preimage resistance of f. □

Proofs based on "semilogarithm" problems. A result of Brown [125] can be seen as an improvement of Theorem 7.12. It is a proof that the single-occurrence security of ECDSA is equivalent, in the random oracle model for H, to the intractability of an *ad hoc* "semilogarithm" problem. This result applies directly to the ElGamal category and can be generalised to other categories.

Definition. *An instance of the $(\bar{\phi}, \bar{\psi})$-semilogarithm problem in the group $\langle G \rangle$ with projection p is a random element $P \in \langle G \rangle$. A solution is a pair (r,u) such that $r = \mathsf{p}(G^{\bar{\phi}(r,u)} P^{\bar{\psi}(r,u)})$.*

Theorem 7.13. (Intractability of the semilogarithm problem is necessary for the security of ECDSA). *If there exists a solver for the (u, ru)-semilogarithm problem, then one can attack all Type I schemes based on the ElGamal category, e.g. ECDSA.*

Proof. To forge a signature of m under public key V, one computes $h = \mathsf{H}(m)$ and $P = V^{1/h}$, finds (r,u) a semilogarithm of P, and computes $s = h/u$. Then $r = \mathsf{p}(G^{h/s} V^{r/s})$ and (m,r,s) is a valid signed message. □

Theorem 7.14. (Intractability of the semilogarithm problem in the random oracle model is sufficient for the SO-CMA security of ECDSA). *If there exists an existential forger under **single-occurrence** chosen message attacks for a Type I scheme based on the ElGamal category, in the random oracle model for H, then there exists a solver for the (u, ru)-semilogarithm problem. The security reduction is loose.*

Proof. This proof is similar to both the proof of Theorem 7.2 (FDH) and that of Theorem 7.10 (Type II).

To find a semilogarithm of $P \in \langle G \rangle$, one pre-selects a random $h_0 \in \mathcal{H}$ and runs the forger with public key $V = P^{h_0}$.

To answer a signature query, the simulator generates random $\alpha \in \mathcal{A}$ and $\beta \in \mathcal{B}$ and computes $R = G^{\alpha} V^{\beta}$, $r = \mathsf{p}(R)$, $h = \lambda_h(\alpha, \beta, r)$ and $s = \lambda_s(\alpha, \beta, r)$, and sets $\mathsf{H}(m) := h$. The signed message is (m,r,s).

To answer an H-oracle query, the corresponding signature query is made, with the exception of one query which is answered with h_0.

If the forgery (m,r,s) corresponds to this H-oracle query with answer h_0, then $(r, h_0/s)$ is a (u, ru)-semilogarithm of P. □

Similar results can be obtained for some other categories. Condition $(\mathfrak{h}1)$ is necessary, but not sufficient.

- *Inverse ElGamal category.* The security of schemes built with the Inverse ElGamal category is based on the (u, ru)-semilogarithm problem. The proof uses $P = V^{1/h}$ and $s = u/h$.
- *GOST category.* The security of schemes built with the GOST category is based on the (u, r)-semilogarithm problem. The proof uses $P = V^{1/h}$ and $s = u \cdot h$.
- *GDSA category.* It is not clear if a similar security result can be obtained for the GDSA category.
- *KCDSAadd category.* It is not clear if a similar security result can be obtained for the KCDSAadd category.
- *KCDSAxor category.* It is not clear if a similar security result can be obtained for the KCDSAxor category.
- *Schnorr category.* The security of schemes built with the Schnorr category is based on the $(u, 1)$-semilogarithm problem. The proof uses $P = V^h$ and $s = u$. Note that this shows that a Type I scheme based on Schnorr category is insecure, because the $(u, 1)$-semilogarithm problem is easy.
- *Swapped Schnorr category.* It is not clear if a similar security result can be obtained for the Swapped Schnorr category.

7.3.2.12 Comments on the security proofs

Comparison with the results from [123]. The two above results follow closely the proofs from Brickell *et al.* [123], but their interactions with the components of the scheme are more clearly detailed. Property ($\mathfrak{p}3$) was not clearly defined in terms of interaction between the category and H.

Moreover, we showed that the proof with an idealised p also works if p is almost invertible.

Comparison with the results from [126]. Our result is more general but does not go into all the details that are considered by Brown in [126]. For example we don't consider zero-finder-resistance, because our toolbox restricts ElGamal category to $\mathcal{H} \subset (\mathbb{Z}/q\mathbb{Z})^\times$ to meet properties ($\mathfrak{g}1$) and ($\mathfrak{g}2$).

Note that footnote number 13 in [126] explains why collision-resistance together with uniformity implies preimage-resistance, so Theorem 4 of [126] doesn't mention the assumption that the hash function needs to be preimage-resistant.

7.3.2.13 Comments on the toolbox

Type I or Type II. Both approaches have their specific security proof where H is idealised. However, Type II is probably to be preferred, because of the fact that a Type I scheme with Schnorr category is insecure, while a Type II scheme with Schnorr category is probably secure, and the fact that the Type I proof only considers SO-CMA security.

Projections. None of the projections previously proposed in the literature has all the required properties for our three proofs. This is why we described how to build a permuted projection. The permuted EC projections probably have all the required properties, but a random fixed permutation of $\langle G \rangle$ may be difficult to design [92]. If partial message recovery is useful, Group projections are the best candidates.

Categories. No category is clearly better than the other ones.

ElGamal category and GOST category have all the required properties, but they rely on distinct semilogarithm problems which are difficult to compare.

Schnorr and Swapped-Schnorr categories are the simplest choice, but they need a Type II hash and are not proven with idealised $\langle G \rangle$.

The KCDSAadd and KCDSAxor categories have the advantage of using simpler computations.

For Schnorr category, the order q can have intractable factorisation, because no inverse is computed. For the KCDSA categories and Swapped-Schnorr category, computing inverses in $(\mathbb{Z}/q\mathbb{Z})^\times$ is only needed for key generation. If the signer's only private information is v^{-1}, neither the verifier nor the signer needs to know the factorisation of q. Intractable factorisation might be useful for an identity based scheme [432].

Two variants. A variant of this toolbox defines the signed message to be the value $(m, R, s) \in \mathcal{M} \times \langle G \rangle \times \mathcal{S}$ instead of (m, r, s). The signature generation is exactly the same, but the verification computes $r = \mathsf{p}(R)$, $h = \mathsf{H}(m, r)$, $\alpha = \phi(h, r, s)$, $\beta = \psi(h, r, s)$, and checks if $R \overset{?}{=} G^\alpha V^\beta$. This variant can be applied to all the schemes described above.

A second variant only applies to the case where neither ϕ not ψ use their input r, e.g. Schnorr category. The signed message is the value $(m, h, s) \in \mathcal{M} \times \mathcal{H} \times \mathcal{S}$. The verification computes $\alpha = \phi(h, \cdot, s)$, $\beta = \psi(h, \cdot, s)$, $R = G^\alpha V^\beta$, $r = \mathsf{p}(R)$, and checks if $h \overset{?}{=} \mathsf{H}(m, r)$. This variant reduces the signature size if $\mathcal{H}$ is smaller the $\mathcal{R}$.

The above security proofs can be translated easily to both variants.

7.3.3 Schemes with security proven in the "real world"

7.3.3.1 Introduction

Thoughts on the signature oracle. The security proof is the description of a reduction algorithm (aka. a simulator) that interacts with a forger and uses the forgery to solve some intractable problem (see Sect. 7.2.3). One important difficulty is that the simulator needs to be able to answer the signature queries made by the forger. The simulator must know some trapdoor that allows it to generate at least q_S valid signatures for arbitrary messages. Notice that this requirement implies that the public key is generated by the simulator. To reduce the security of the scheme to the intractability of a mathematical problem, it is necessary that the forgery could not have been made by the simulator. This remark was made by Goldwasser, Micali and Rivest [273, Sect. 4].

One-time and fixed-time signature schemes. With a one-time signature scheme, a public key is used for validating one signature only. For fixed-time signature schemes, there is an *a priori* upper bound on the number of messages that can be validated with a given public key.

One-time signature schemes as chameleon hashing. A one-time signature scheme is KS-secure (also called *secure chameleon hashing* [377]) if no KS-attacker exists. Such an attacker is allowed to make *key-then-sign* queries, where the input is a message and the output is a random public key with a valid signed message. The attacker succeeds if it can make another valid signed message for one of those public keys.

The refreshing paradigm. The key idea is that all the messages will be signed by a secure one-time signature scheme, but with a different public key for each message. The public key of the secure scheme is the concatenation of all those one-time public keys. With this simple construction the public key of the whole scheme has length q_S times the length of the public key of the one-time signature scheme.

This technique for constructing signatures is also used in the online/offline approach to improve the performance of digital signature schemes [223], where a public key for a fast one-time scheme is signed by a normal secure and slow signature scheme.

7.3.3.2 Tree constructions based on the refreshing paradigm

This technique can be used to construct secure signature schemes from secure one-time signature schemes (see also [265, Volume 2, Sect. 6.4.2]).

Practical constructions are based on the refreshing paradigm and use an authentication tree to authenticate the one-time public keys with respect to a unique short public key. No such scheme has been submitted to NESSIE.

7.3.3.3 Using a RAND-secure scheme in the refreshing paradigm

Any RAND-secure signature scheme can be used to authenticate the one-time public keys. The key idea is that the RAND scheme is able to securely sign any random one-time public key, and each one-time public key can securely sign one arbitrary message.

Theorem 7.15. *If* $\mathsf{GenerateA}, \mathsf{KeyGenA}, \mathsf{SignA}, \mathsf{VerA}$ *defines a* KS*-secure signature scheme and if* $\mathsf{GenerateB}, \mathsf{KeyGenB}, \mathsf{SignB}, \mathsf{VerB}$ *defines a* RAND*-secure signature scheme, then the following signature scheme is secure under adaptive chosen message attacks.*

- $\mathsf{Generate}$ *runs* $\mathsf{GenerateA}$ *and* $\mathsf{GenerateB}$ *and outputs* $\mathsf{param} = \mathsf{paramA} \| \mathsf{paramB}$.
- KeyGen *runs* $\mathsf{KeyGenB}$ *and outputs* $\mathsf{pk} = \mathsf{pkB}$ *and* $\mathsf{sk} = \mathsf{skB}$.
- Sign *computes* $(h, \hat{h}) = \mathsf{KeyGenA}$. *Then it computes* $s = \mathsf{SignA}_{\hat{h}}(m)$ *and* $t = \mathsf{SignB}_{\mathsf{sk}}(h)$, *and outputs* $\sigma = (t, s)$.
- Ver *computes* $h = \mathsf{VerB}_{\mathsf{pk}}(t)$ *and* $m = \mathsf{VerA}_h(s)$.

Proof. Let us name (t, s) the forgery, $h = \mathsf{VerB}_{\mathsf{pk}}(t)$ the corresponding one-time public key and $m = \mathsf{VerA}_h(s)$ the corresponding message. Let us name (t_j, s_j) and (h_j, m_j) the questions and answers to the j-th query to the signing oracle.

All the information that the forger receives about the scheme A is contained in m_j, h_j, s_j, where the forger chooses m_j but the public key h_j is random. This is exactly a KS-attack.

All the information that the forger receives about the scheme B is contained in h_j, pk, t_j, where the forger does not choose the values h_j, which are random values. This is exactly a RAND-attack.

Two variants of the theorem can be proven: with or without non-malleability. With non-malleability, we define two types of forgeries.

- *KS forgery.* $\exists j_0 : t = t_{j_0}$. Therefore $s \neq s_{j_0}$ and $h = h_{j_0}$. The forger has produced a new valid signed message s for an old public key h of the scheme A, which contradicts its KS-security.
- *RAND forgery.* $\forall j, t \neq t_j$. The forger has produced a new valid signed message t for the scheme B, which contradicts its RAND-security.

Without non-malleability, we define two types of forgeries.

- *KS forgery.* $\exists j_0 : h = h_{j_0}$ and $m \neq m_{j_0}$. The forger has produced a valid signature s for a new message m and an old public key h of the scheme A, which contradicts its KS-security.
- *RAND forgery.* $\forall j, h \neq h_j$. The forger has produced a valid signature t for a new message h for the scheme B, which contradicts its RAND-security.

□

The two main examples are the GHR scheme [256] and the ACE-Sign submission to NESSIE, both of whose security is based on the strong RSA assumption. One can compare these two schemes. While all components of ACE-Sign are efficiently computable, GHR needs an efficient collision-resistant hash function with an additional property named *division-intractability*, e.g. a hash function that outputs prime numbers. On the other hand, GHR has a tight reduction to the strong RSA assumption, while the reduction for ACE-Sign is not so tight.

7.3.3.4 The Cramer-Shoup family of schemes: ACE-Sign and variants

The RAND component. The following RAND-secure scheme is the core of this family of schemes.

- *Domain parameters.* The security parameter l is the output length of a collision-resistant hash function H and determines the length of the RSA composite number and of some prime numbers.
- *Key generation algorithm.* The key generation algorithm chooses two random strong primes p and q, computes $n = pq$ and chooses two random values x and g in QR_n (quadratic residues). The public key is $\mathsf{pk} = (n, x, g)$ and the private key is $\mathsf{sk} = (p, q)$.
- *Verification algorithm.* The verification of a signed message $(m, y, e) \in \mathcal{M} \times QR_n \times [2^l, 2^{l+1}]$ begins with $h = \mathsf{H}(m)$ and checks if $y^e \stackrel{?}{=} xg^h$.
- *Signing algorithm.* To sign the message m one takes a random prime $e \in [2^l, 2^{l+1}]$ and uses the secret key to compute $y = (xg^{\mathsf{H}(m)})^{1/e}$.

Theorem 7.16. *This scheme is RAND-secure under the Strong RSA assumption, with not so tight reduction.*

Proof. Using a (t, ε, q_S)-forger, the following algorithm either solves with probability 1 the flexible RSA problem or solves with probability $1/q_S$ the e-th root problem.

We can define two types of forgeries, depending on their common values with the queries. We let h_j, y_j, e_j denote the values for the j-th query and h, y, e the values for the forgery.

- *FLEX forgery.* $\forall j, e \neq e_j$. The reduction wants to solve the flexible RSA problem for (n, u). It generates q_S random primes $e_j \in [2^l, 2^{l+1}]$ and a random $\alpha \in [1...n^2]$, then deduces the elements of the public key $g = u^{2\prod_j e_j}$ and $x = g^{\alpha}$. It can answer the j-th oracle query by generating a random m_j and computing $y_j = u^{2(\alpha+\mathsf{H}(m_j))\prod_{i\neq j} e_i}$. The forgery will give an e-th root of g, and therefore a non-trivial root of u.
- *GUESS forgery.* $\exists j_0 : e = e_{j_0}$. The reduction wants to find $u^{1/e'} \bmod n$. It chooses a random j_0 in $1...q_S$, sets $e_{j_0} = e'$ and generates $q_S - 1$ other random primes $e_j \in [2^l, 2^{l+1}]$. It generates random values for $\alpha \in [1...n^2]$ and $h' = \mathsf{H}(m')$, then deduces the elements of the public key $g = u^{2\prod_{j\neq j_0} e_j}$ and $x = g^{\alpha e' - h'}$. It can answer the j_0-th oracle query with $m_{j_0} = m'$, $y_{j_0} = g^{\alpha}$ and $e_{j_0} = e'$. It can answer all other oracle queries by generating a random m_j and computing $y_j = u^{2(\alpha e' - h' + \mathsf{H}(m_j))\prod_{i\neq j, j_0} e_i}$. If indeed the forgery is such that $e = e'$, then $(y/y_{j_0})^e = g^{\mathsf{H}(m)-h'}$ and it gives the e-th root of u. This succeeds if j_0 was correctly guessed (probability $1/q_S$).

□

The chameleon hash. The following one-time signature scheme is used in the original scheme.

- *Domain parameters.* The security parameter l is the output length of a collision-resistant hash function H and determines the length of the RSA composite number n and of a prime number e. A random f in QR_n is chosen and $g = f^e$ is computed. The public domain parameters are n, g and e. The private parameter f is used for key generation. [5]
- *Key generation algorithm.* The key generation algorithm chooses a random value z in QR_n and computes $x = z^e$. The public key is $\mathsf{pk} = (x)$ and the private key is $\mathsf{sk} = (f, z)$.
- *Verification algorithm.* The verification of a signed message $(m, y) \in \mathcal{M} \times QR_n$ begins with $h = \mathsf{H}(m)$ and checks if $y^e \overset{?}{=} x g^h$.
- *Signing algorithm.* To sign the message m one uses the secret key to compute $y = z f^{\mathsf{H}(m)}$.

Theorem 7.17. *This scheme is* KS*-secure under the RSA assumption, with tight reduction.*

Proof. The reduction algorithm wants to find $u^{1/e} \bmod n$. It defines $g = u$ and interacts with the forger. In answer to a key-then-sign query for the message m,

[5] An equivalent scheme can be obtained by generating n with known secret factorisation, kept in the private key.

the reduction generates a random y and computes the public key $x = y^e g^{-\mathsf{H}(m)}$. The forgery is a pair (m', y') such that $(y'/y)^e = g^{\mathsf{H}(m')-\mathsf{H}(m)}$, which gives an e-th root of g. □

ACE-Sign. The actual Cramer-Shoup scheme [170] is an improvement on the straightforward construction based on the two components described above.

The security of the construction is the same if the values n and g are common to the RAND-secure scheme and to the one-time scheme. With this improvement, the public key is (n, x, g, e'), the signed message is (m, y, e, x', y') and the verification checks if $x \overset{?}{=} y^e g^{-\mathsf{H}(x')}$ and if $x' \overset{?}{=} (y')^{e'} g^{-\mathsf{H}(m)}$. If we also notice that x' can be omitted from the signed message, the result is ACE-Sign, fully described in Sect. 7.5.1.1.

Another Chameleon hash. The following scheme is also proposed as an alternative in [170]. It is based on the intractability of discrete logarithms.

- *Domain parameters.* The security parameter l is the output length of a collision-resistant hash function H. Computations are made in a group $\langle G \rangle$ of order q. Two random α, α' coprime to q are chosen such that the public parameters $g_1 = G^{\alpha'}$ and $g_2 = g_1^{\alpha}$ are generators of $\langle G \rangle$. The private parameter α is kept for key generation.
- *Key generation algorithm.* The key generation algorithm chooses a random β and computes $x = g_1^{\beta}$. The public key is $\mathsf{pk} = (x)$ and the private key is $\mathsf{sk} = (\alpha, \beta)$.
- *Verification algorithm.* The verification of a signed message $(m, t) \in \mathcal{M} \times \mathbb{Z}/q\mathbb{Z}$ begins with $h = \mathsf{H}(m)$ and checks if $x \overset{?}{=} g_1^t g_2^h$.
- *Signing algorithm.* To sign the message m one uses the secret key to compute $t = \beta - \alpha\mathsf{H}(m) \bmod q$.

Theorem 7.18. *This scheme is KS-secure if the discrete logarithm problem in $\langle G \rangle$ is intractable, with tight reduction.*

Proof. The reduction algorithm wants to find the logarithm of g_2 with respect to g_1. In answer to a key-then-sign query for the message m, the reduction generates a random t and computes the public key $x = g_1^t g_2^{\mathsf{H}(m)}$. The forgery is a pair (m', t') such that $g_2 = g_1^{(t-t')/(\mathsf{H}(m')-\mathsf{H}(m))}$. □

Generalisation to any group. Damgård and Koprowski [184] proposed a generalisation of ACE-Sign to any group where the equivalent of the flexible RSA problem is intractable.

Variants without a chameleon hash. The key idea is that the chameleon hash is only needed for the j_0-th query in the GUESS forgery. In all the other cases any arbitrary message can be signed by the reduction algorithm. Therefore a chameleon hash brings unnecessary flexibility.

Instead of replacing $g^{\mathsf{H}(m)}$ by $g^{\mathsf{H}(\text{chameleon}-\text{hash}(m))}$, one first variant, due to Camenisch and Lysyanskaya [131], replaces $g^{\mathsf{H}(m)}$ by $g_1^t g_2^{\mathsf{H}(m)}$. The signed message is (m, t, e, y) and the verification checks if $y^e \overset{?}{=} x g_1^t g_2^{\mathsf{H}(m)}$. If n is an l'-bit number and H has l-bit output (i.e. security $l/2$ bits) then this variant is secure when t is an $l' + 3l/2$-bit number. Therefore it is less efficient than ACE-Sign.

Theorem 7.19. *The Camenisch-Lysyanskaya scheme is secure under the Strong RSA assumption, with not so tight reduction.*

Proof. The proof is very similar to the proof of Theorem 7.16.

- *FLEX forgery.* $\forall j, e \neq e_j$. The reduction wants to solve the flexible RSA problem for (n, u). It generates q_S random primes $e_j \in [2^l, 2^{l+1}]$ and random $\alpha, \beta \in [1...n^2]$, then deduces the elements of the public key $g_1 = u^{2\prod_j e_j}$, $g_2 = g_1^\beta$ and $x = g_1^\alpha$. It can answer the j-th oracle query for m_j by generating a random $t_j \in [0...2^{l'+3l/2}]$ and computing $y_j = u^{2(\alpha+t_j+\beta\mathsf{H}(m_j))\prod_{i\neq j} e_i}$. The forgery will give an e-th root of g_1, and therefore a non-trivial root of u.
- *GUESS forgery.* $\exists j_0 : e = e_{j_0}$. The reduction wants to find $u^{1/e'} \bmod n$. It chooses a random j_0 in $1...q_S$, sets $e_{j_0} = e'$ and generates $q_S - 1$ other random primes $e_j \in [2^l, 2^{l+1}]$. It generates random $\alpha, \beta \in [1...n^2]$ and $\gamma \in [0...2^{l'+3l/2}]$, then deduces the elements of the public key $g_1 = u^{2\prod_{j\neq j_0} e_j}$, $g_2 = g_1^\beta$ and $x = g_1^{\alpha e'-\gamma}$. It can answer the j-th oracle queries for m_j with $j \neq j_0$ by generating a random $t_j \in [0...2^{l'+3l/2}]$ and computing $y_j = u^{2(\alpha e'-\gamma+t_j+\beta\mathsf{H}(m_j))\prod_{i\neq j,j_0} e_i}$. It can answer the j_0-th oracle query for m_{j_0} with $y_{j_0} = g^\alpha$, $e_{j_0} = e'$ and $t_{j_0} = \gamma - \beta\mathsf{H}(m_{j_0})$. Owing to the condition $\gamma \in [0...2^{l'+3l/2}]$, the value t appears to be uniform in $[0...2^{l'+3l/2}]$. If indeed the forgery is such that $e = e'$, then $(y/y_{j_0})^e = g^{t-t_{j_0}+\beta\mathsf{H}(m)-\beta\mathsf{H}(m_{j_0})}$ and it gives the e-th root of u. This succeeds if j_0 was correctly guessed (probability $1/q_S$).

□

Two other variants, due to Fischlin [231], replace $g^{\mathsf{H}(m)}$ by $g_1^t g_2^{t+\mathsf{H}(m)}$ or by $g_1^t g_2^{t\oplus\mathsf{H}(m)}$. The signed message is (m, t, e, y) and the verification checks e.g. if $y^e \stackrel{?}{=} x g_1^t g_2^{t\oplus\mathsf{H}(m)}$. These variants are secure for l-bit t, so their efficiency is similar to ACE-Sign with the DL-based chameleon hash.

Theorem 7.20. *The Fischlin schemes are secure under the Strong RSA assumption, with not so tight reduction.*

Proof. The proof is very similar to the proofs of Theorem 7.16 and 7.19.

- *FLEX forgery.* $\forall j, e \neq e_j$. The reduction wants to solve the flexible RSA problem for (n, u). It generates q_S random primes $e_j \in [2^l, 2^{l+1}]$ and random $\alpha, \beta \in [1...n^2]$, then deduces the elements of the public key $g_1 = u^{2\prod_j e_j}$, $g_2 = g_1^\beta$ and $x = g_1^\alpha$. It can answer the j-th oracle query for m_j by generating a random $t_j \in [0...2^l]$ and computing $y_j = u^{2(\alpha+t_j+\beta(t_j\oplus\mathsf{H}(m_j)))\prod_{i\neq j} e_i}$. The forgery will give an e-th root of g_1, and therefore a non-trivial root of u.
- *GUESS/1 forgery.* $\exists j_0 : e = e_{j_0}$ and $t_{j_0} \neq t$. The reduction wants to find $u^{1/e'} \bmod n$. It chooses a random j_0 in $1...q_S$, sets $e_{j_0} = e'$ and generates $q_S - 1$ other random primes $e_j \in [2^l, 2^{l+1}]$. It generates random $\alpha, \beta \in [1...n^2]$ and $\gamma \in [0...2^l]$, then deduces the elements of the public key $g_1 = u^{2\prod_{j\neq j_0} e_j}$, $g_2 = g_1^{\beta e'}$ and $x = g_1^{\alpha e'-\gamma}$. It can answer the j-th oracle queries for m_j with $j \neq j_0$ by generating a random $t_j \in [0...2^l]$ and computing

$y_j = u^{2(\alpha e' - \gamma + t_j + \beta e'(t_j \oplus \mathsf{H}(m_j))) \prod_{i \neq j, j_0} e_i}$. It can answer the j_0-th oracle query for m_{j_0} with $y_{j_0} = g_1^{\alpha + \beta(\gamma \oplus \mathsf{H}(m_{j_0}))}$, $e_{j_0} = e'$ and $t_{j_0} = \gamma$. If indeed the forgery is such that $e = e'$ and $t_{j_0} \neq t$, then $(g_1^{\beta(t_{j_0} \oplus \mathsf{H}(m_{j_0})) - \beta(t \oplus \mathsf{H}(m))} \cdot y / y_{j_0})^e = g^{t - t_{j_0}}$ and it gives the e-th root of u.

- *GUESS/2 forgery.* $\exists j_0 : e = e_{j_0}$ and $t_{j_0} \oplus \mathsf{H}(m_{j_0}) \neq t \oplus \mathsf{H}(m)$. Similar to the GUESS/1 forgery, with the public key $g_2 = u^{2 \prod_{j \neq j_0} e_j}$, $g_1 = g_2^{\beta e'}$ and $x = g_2^{\alpha e' - \gamma}$. Also $t_{j_0} = \gamma \oplus \mathsf{H}(m_{j_0})$ and $y_{j_0} = g_2^{\alpha + \beta t_{j_0}}$.

□

7.3.4 Current standards

The US NIST (National Institute of Standards and Technology) issued in 1994 a FIPS (U.S. Government Federal Information Processing Standard) that described DSA (Digital Signature Algorithm) [464]. This standard has been revised twice: FIPS-186-1 [466] and FIPS-186-2 [467] added the ANSI X9.31 and X9.62 standards.

The ANSI (American National Standards Institute) issued in 1999 ANSI X9.31 [20], which is a partial domain hash RSA signature, and ANSI X9.62 [21], which is ECDSA.

The ISO (International Organisation for Standardisation) has published ISO-9796 [301, 302], ISO-14888 [308, 309] and ISO-15946-2 [310].

The IEEE P1363 group has published various standards on public key cryptography [299, 300] that go down to implementation details.

7.4 Digital signature schemes considered during Phase II

The complete description of a signature scheme needs to explain how to convert integers to and from octet strings or bit strings. Usually, this has no influence on the security and will not be mentioned here.

7.4.1 ECDSA

ECDSA was submitted by Certicom [319] and is a signature scheme based on the intractability of the discrete logarithm problem in an elliptic curve subgroup. It was first proposed in 1992 by Scott Vanstone [610] in response to NIST's request for public comments on their first proposal for DSS [464]. It is an ISO [309] standard since 1998, an ANSI [21] standard since 1999 and an IEEE [299] and NIST [467] standard since 2000. Interoperability between these standards is discussed in `http://www.certicom.com/resources/news/news_103000.html`. The version submitted to NESSIE is the ANSI X9.62 ECDSA.

7.4.1.1 The design

ECDSA is a special case of the family of DL-based signature schemes described in Sect. 7.3.2. It is defined on a prime order elliptic curve subgroup with ElGamal category, ECxq projection and type I hash.

- *Domain parameters.* The security parameter is an integer l, e.g. 160. Let E be an elliptic curve over the finite field $\mathbb{F}$ and $\langle G\rangle$ a subgroup of known prime order $q \in [2^l, 2^{l+1}]$ and known generator G. We use the additive notation for this group.
  $\mathbb{F}$ is either a prime field $GF(p)$ or a characteristic 2 field $GF(2^n)$.
  Let H be a hash function with l bits of output (usually SHA-1) and $\mathsf{i}_{\mathbb{F}}$ a mapping from $\mathbb{F}$ to the set of integers modulo q, that does the conversion from $\mathbb{F}$ to $\mathbb{Z}$ as specified in ANSI X9.62 and then a reduction modulo q.
- *Key generation algorithm.* The key generation algorithm chooses a random $v \in (\mathbb{Z}/q\mathbb{Z})^\times$ and sets $\mathsf{pk} = V = v \cdot G$ and $\mathsf{sk} = v$.
- *Verification algorithm.* The verification algorithm on a signed message $(m, r, s) \in \mathcal{M} \times \mathbb{Z}/q\mathbb{Z} \times (\mathbb{Z}/q\mathbb{Z})^\times$ computes $h = \mathsf{H}(m)$ and $R = s^{-1} \cdot (h \cdot G + r \cdot V)$, and checks if $r \overset{?}{=} \mathsf{i}_{\mathbb{F}}(R_x)$.
- *Signing algorithm.* To sign the message m one takes a random invertible $k \in (\mathbb{Z}/q\mathbb{Z})^\times$, and computes $R = k \cdot G$, $r = \mathsf{i}_{\mathbb{F}}(R_x)$, $h = \mathsf{H}(m)$ and $s = k^{-1}(h + vr)$. If s is not invertible, another k is taken. The signed message is (m, r, s).

Parameter generation. The parameters for ECDSA consist mainly of the description of a suitable elliptic curve and of a base point that generates a subgroup with large prime order.

Depending on the variant of ECDSA, the elliptic curve and base point can be chosen from a table of suitable values [137] or can be taken at random subject to the base point having large prime order. Any underlying field can be chosen, but only prime fields or binary fields are usually considered.

The elliptic curve subgroup is not part of the public key, it is a system parameter common to all users. Therefore it is important that no weaknesses can be found in it.

Okeya *et al.* [503] proposed the use of elliptic curves with Montgomery form to protect against timing attacks. This leads to the OK-ECDSA variant, which was not submitted to NESSIE but is studied by CRYPTREC [502].

7.4.1.2 Security analysis (see also Sect. 7.3.2)

Attacks on ECDSA. Necessary conditions for the security of ECDSA are one-wayness and collision resistance of H and intractability of the (u, ru)-semilogarithm in the elliptic curve E, which implies intractability of the discrete logarithm.

Security proofs for ECDSA. This signature scheme has been published for a long time and various security arguments have been provided which show that the two necessary conditions above might be sufficient. However, none of those security arguments can be used to argue that ECDSA is an optimal design for a scheme based on the intractability of the elliptic curve discrete logarithm.

The first published arguments were proofs in the random oracle model that in fact apply to variants of ECDSA, e.g. to KCDSA (proof with idealised p) or to the Schnorr or PVSSR schemes (proof with idealised H). The proof that works in the generic group model applies to ECDSA itself, but it is also an example of a case where a generic proof is explicitly invalidated by some specific properties of the components. [6]

Therefore one big concern about ECDSA is that it is probably not the best choice in the extensive family of DL-based signature schemes.

The hash function of ECDSA. The function H should be a one-way collision resistant hash function with output in a subset of $(\mathbb{Z}/q\mathbb{Z})^\times$, and the security of the scheme may be weakened if this output set has substantially fewer than q elements.

But the specifications of ECDSA say that H is the SHA-1 hash function, whose output is a 160-bit string, reduced mod$q$. Therefore it might not be an element of $(\mathbb{Z}/q\mathbb{Z})^\times$, because it can be zero. An additional property required for SHA-1 is *zero-finder resistance* [126], which means that a preimage of 0 should be hard to find.

Some realistic attacks on DSA and ECDSA rely on the ability to choose the parameters of the scheme after having studied some properties of the hash function [96, 612]. An easy protection is the inclusion in the input of H of some certification data depending on the parameters and the public key. It has been proposed for KCDSA and some other schemes [340, 444].

Other comments: parameter and key validation. Note that to prove that the parameters are not designed to correspond to a weak elliptic curve, ECDSA asks for a certification, which is the seed used for the generation of a random curve. This certification technique can be bypassed in characteristic 2, and should absolutely be improved [615].

Note also that the ECDSA submitted to NESSIE asks that the verification and the signing algorithms make sure that $r \neq 0$, while the description in this document does not make this verification. The rationale for this check on r is to protect against a very specific type of bad parameter. It is felt that a good algorithm for parameter validation would be preferred to ad hoc checks, and that it is harmful to make the verification and signing algorithms more complicated than necessary.

The random nonce. The random value k used in the signing algorithm should be *unpredictible*, otherwise the scheme can be attacked [216, 482, 483]. However, this value can be deterministically generated from the message and a secret value, like it is done for the FDH-D design (cf. Sect. 7.3.1.3).

Side-channel attacks. In elliptic curve cryptography, scalar point multiplication is a crucial operation. As mentioned in Sect. A.1 algorithms performing this

[6] The elliptic curve subgroup can easily be distinguished from an ideal group because it has the trivial automorphism $(x, y) \mapsto (x, -y)$. In the case of ECDSA, the choice of p being the reduction of the x-coordinate allows one to use this automorphism to forge a new signature for an existing signed message.

operation are a particular target for side-channel attacks. In ECDSA, the task is to compute $k \cdot G$. Any information concerning the value k that can leak during the computation might be used by an attacker to compute the secret key. Different side-channel techniques exist to get the information and various countermeasures have been proposed in the literature to defend against this kind of attack. For further discussion on this subject, the reader is referred to the Annex A and to the survey [508] by Oswald and Preneel.

Another side-channel technique that can be used is introducing faults during the computation of the signature (for a general introduction to fault attacks, see Sect. A.2). For example, and following the original idea of Bao *et al.* [33] against DSA, if an attacker is able to change one bit of the secret key v used by the signer, then the erroneous signature can be used to recover the original value of the bit. Indeed, the signer would output $s' = k^{-1}(h + vr \pm 2^i r)$, assuming the attacker changed the i-th bit of v, and depending on its original value. So the verification algorithm would give ${s'}^{-1}(h \cdot G + r \cdot V) = k \cdot G \pm (2^i r {s'}^{-1}) \cdot G$. Thus, computing all the possible values for $R = {s'}^{-1}(h \cdot G + r \cdot V) \pm (2^i r {s'}^{-1}) \cdot G$ and detecting if $r \overset{?}{=} \mathsf{i}_{\mathbb{F}}(R_x)$. the attacker will discover the original value of the bit that he flipped. In [58], Biehl *et al.* present another idea: faults are used to make the device apply the multiplication algorithm to a point that is actually on a different, probably cryptographically weak, elliptic curve. The result of this computation might be used to recover the secret key v. Countermeasures against fault attacks are necessary, e.g. checking the consistency of the output.

7.4.2 ESIGN

ESIGN was submitted by NTT [244] and can be viewed as a variant of RSA that has faster signing but relies on more demanding security assumptions.

7.4.2.1 The design

- *Domain parameters.* The security parameter is an integer l, e.g. 512. Let H be a hash function with $l-1$ bits of output and e be a small integer.
  In the original submission of ESIGN to NESSIE [244] the requirements are $l \geq 352$ and $e \geq 8$, with recommended values $l = 384$ and $e = 1024$. In the specification of ESIGN-D [496] the requirements are $l \geq 342$ and $\frac{3l}{2} \leq e \leq 2^{l/4}$, with recommended values $l = 512$ or 1024 and $e = 65537$.
- *Key generation algorithm.* Let p and q be distinct primes from $[2^{l-1}, 2^l]$ such that $n = p^2 q \in [2^{3l-1}, 2^{3l}]$. The keys are $\mathsf{pk} = n$ and $\mathsf{sk} = (p, q)$.
- *Verification algorithm.* The verification of a signed message $(m, s) \in \mathcal{M} \times \mathbb{Z}/n\mathbb{Z}$ begins with $h = \mathsf{H}(m)$ and $x = s^e \bmod n$, and checks if $x \overset{?}{=} h \cdot 2^{2l} + w$ where $w \in [0, 2^{2l-1}]$.
- *Basic signing algorithm.* To sign the message m one takes a random $w \in [0, 2^{2l-1}]$ and computes $h = \mathsf{H}(m)$, $x = h \cdot 2^{2l} + w$ and $s = x^{1/e} \bmod n$. The signed message is (m, s).
- *ESIGN fast signing algorithm.* To sign the message m one computes $h = \mathsf{H}(m)$. Then one takes a random $r < pq$ and computes $u = h \cdot 2^{2l} - r^e \bmod n$, $v = \left\lceil \frac{u}{pq} \right\rceil$

and $w = v \cdot pq - u$ until $w < 2^{2l-1}$. Then $t = \frac{v}{e \cdot r^{e-1}} \bmod p$ and $s = r + t \cdot pq$. The signed message is (m, s).
Both signing algorithms give the same output distribution.

7.4.2.2 Security analysis (see also Sect. 7.3.1)

Attacks on ESIGN and its variants. Necessary conditions for the security of ESIGN are one-wayness and collision resistance of H and intractability of the AER problem.

Security proofs for ESIGN and its variants. ESIGN is an FDH signature scheme based on the "truncated e-th power" function, which is defined by $\mathsf{f}(x) = \left\lfloor \frac{x^e \bmod n}{2^{2l}} \right\rfloor$. The original description of ESIGN [244] made the statement that the original security proof of FDH applies to ESIGN, but Stern *et al.* [597] noticed that f^{-1} is randomised, and therefore this proof only assesses the security against a $\mathsf{SO\text{-}CMA}$ attacker.

A variant named ESIGN-D has been described [274] and replaces the original submission. This variant uses the FDH-D technique described in Sect. 7.3.1.3. Another variant named ESIGN-R is described in the same document and uses PFDH with a seed of length at least $2 \log_2 q_S$.

To assess the security of ESIGN-D and ESIGN-R, we need to study the properties of f.

- *Definitions.* The set $\mathcal{I}_l$ contains all pairs (n, η) with suitable $n = p^2 q$ and $\eta \in \mathbb{Z}/n\mathbb{Z}$. Let us define $\mathcal{X} = \{x \in \mathbb{Z}/n\mathbb{Z} | x = 0 \| h \| 0 \| w$ with $h \in [0, 2^{2l-1}]$ and $w \in [0, 2^{2l-1}]\}$. The input sets are $\mathcal{S} = \mathcal{T} = \{x \in \mathbb{Z}/n\mathbb{Z} | x^e \bmod n \in \mathcal{X}\}$ and the output set is $\mathcal{H} = [0, 2^{2l-1}]$.
  We define $\mathsf{f}(x) = \left\lfloor \frac{x^e \bmod n}{2^{2l}} \right\rfloor$ and $\mathsf{g}(x) = \left\lfloor \frac{\eta \cdot x^e \bmod n}{2^{2l}} \right\rfloor$.
- *Computational properties.* These properties (OW1, OW2, OW3, CF1, CF2, CF3, TR1 and TR2) are implied by the description of the scheme, the probability distribution on $\mathcal{S} = \mathcal{T}$ being uniform, with a sampling algorithm uniformly taking an element of $\mathbb{Z}/n\mathbb{Z}$ until it is in $\mathcal{S}$.
- *Statistical properties.* Property UN (almost uniformity of the output of f and g) can be proved [217, 497, 596].
  Property TR3 comes from the following facts. Both signing algorithms generate values with their e-th power uniform in $\mathcal{X}$. The e-th power function is a bijection between $\mathcal{S}$ and $\mathcal{X}$.
- *Intractability properties.* Preimage resistance (property OW4) holds if the AER problem is hard.
  Claw-freeness (property CF4) holds if the Claw-AER problem is hard.
  Second preimage resistance (property OW5) holds if the 2nd-AER problem is hard.

Therefore, the following security results have been proven.

- If the AER problem is hard with a security level of k bits, then both ESIGN-D and ESIGN-R have a proven (in the random oracle model) security level of $k - \log_2 q_H$ bits. Non-malleability requires also that the 2nd-AER problem is hard.

- If the Claw-AER problem is hard with a security level of k bits, then ESIGN-D also has a proven security level of $k - \log_2 q_S$ bits and ESIGN-R a proven security level of k bits. Non-malleability still requires that the 2nd-AER problem is hard.

The hash function of ESIGN and its variants. Both the original description of ESIGN and the tweak to ESIGN-D use a hash function H, based on SHA-1, which is common to all the public keys. It may be a better design to improve the security in the multi-key setting by including pk in the input of H.

In the original description of ESIGN, the value $\mathsf{H}(m)$ is defined to be the first bits of SHA-$1^{80}_{\sigma}(0\|m)\|$SHA-$1^{80}_{\sigma}(1\|m)\|$..., where SHA-1^{80}_{σ} is a variant of SHA-1 with a different starting value and truncated to 80 bits. This is not an optimal design because it is too slow if m is a long message. The specification of ESIGN-D uses a better design: SHA-1(SHA-1(m)$\|0)\|$SHA-1(SHA-1(m)$\|1)\|$....

Side-channel attacks. For a general introduction to side-channel attacks, see the Annex A. We did not find any side-channel attack against ESIGN. The fact that it is a randomised algorithm protects ESIGN against some side-channel attacks.

ESIGN-D is a deterministic variant of ESIGN. We can use this to mount a side-channel attack that uses faults (see Sect. A.2). Imagine an attacker can disturb the signing algorithm by introducing faults during the computation of either u, v or t. This leads to an erroneous value $t' = t \pm \epsilon$, with $0 \leq \epsilon < p$. Thus, an erroneous signature s' is computed, with the following relation: $s' = s \pm \epsilon \cdot pq$, where $s = r + t \cdot pq$ is the correct signature. If an attacker can get both the correct signature and an erroneous one on the same message, then he gets ϵpq, which is not a multiple of N, so $\gcd(s - s', N) = pq$ and the modulus is factorised. This attack is particularly powerful because all it needs is a random error during the computation of the most time-consuming steps. So any implementation of ESIGN-D should be protected against fault attacks.

7.4.3 SFLASH (new version)

All signature schemes from the FLASH family are FDH-D schemes based on the intractability of variants of the HFE problem. These variants are named C^{*--} (see [515]).

7.4.3.1 The design

- *Domain parameters.* The parameters are four integers q, d, n and r, a security parameter l, a hash function H with output in $(\mathbb{F}_q)^{n-r}$ and a family of pseudorandom functions prf with an l-bit index, input in $(\mathbb{F}_q)^{n-r}$ and output in $(\mathbb{F}_q)^r$. The function $\mathbf{F}(x) = x^{q^d+1}$ is a bijection of the finite field $\mathbb{F}_{q^n}$ with inverse $\mathbf{F}^{-1}(x) = x^h$, where $h = (q^d+1)^{-1} \bmod (q^n - 1)$. The function $\phi : \mathbb{F}_{q^n} \to (\mathbb{F}_q)^n$ fixes a representation of $\mathbb{F}_{q^n}$ as an $\mathbb{F}_q$-vector space. Let $\mathbf{f} = \phi^{-1} \circ \mathbf{F} \circ \phi$.
  For SFLASH (new version) we have $l = 80$, $q = 128 = 2^7$, $d = 11$, $n = 37$ and $r = 11$ and H and prf based on SHA-1.

- *Key generation algorithm.* Two random affine bijections $\mathbf{s}$ and $\mathbf{t}$ of $(\mathbb{F}_q)^n$ and a random l-bit value Δ are generated. They are the private key.
  The public key is $(P_1, ..., P_{n-r})$ where $(P_1, ..., P_n)$ are the quadratic polynomials that describe $\mathbf{t} \circ \mathbf{f} \circ \mathbf{s}$.
- *Verification algorithm.* The verification of a signed message $(m, s_1, ..., s_n) \in \mathcal{M} \times (\mathbb{F}_q)^n$ begins with $(h_1, ..., h_{n-r}) = \mathsf{H}(m)$ and then computes $(y_1, ..., y_{n-r}) = (P_1, ..., P_{n-r})(s_1, ..., s_n)$ and checks if all $h_i \overset{?}{=} y_i$.
- *Signing algorithm.* To sign the message m one computes $(h_1, ..., h_{n-r}) = \mathsf{H}(m)$, then $(x_1, ..., x_r) = \mathsf{prf}_\Delta(h_1, ..., h_{n-r})$ and finally $(s_1, ..., s_n) = \mathbf{s}^{-1} \circ \mathbf{f}^{-1} \circ \mathbf{t}^{-1}(h_1, ..., h_{n-r}, x_1, ..., x_r)$. The signed message is $(m, s_1, ..., s_n)$.
  The fact that $x \mapsto x^q$ is linear over $(\mathbb{F}_q)^n$ allows for some tricks that help to do a fast computation of $\mathsf{F}^{-1}(x) = x^h$ and therefore of $\mathbf{f}^{-1}$.

7.4.3.2 Security analysis (see also Sect. 7.3.1)

Attacks on SFLASH (new version). The security of SFLASH is exactly the security of an FDH-D scheme based on the one-wayness of the function $\mathsf{f} = (P_1, ..., P_{n-r}) : (\mathbb{F}_q)^n \to (\mathbb{F}_q)^{n-r}$, where the P_j are quadratic polynomials in $\mathbb{F}_q$ generated with a C^{*--} trapdoor.

Necessary conditions for the security of SFLASH are one-wayness and collision resistance of H and one-wayness of f.

Security proof for SFLASH (new version). The preimage resistance of the function f is not well defined, but two techniques to find preimages can be described. This is not sufficient for a high level of confidence in the intractability of the C^{*--} problem, but it is sufficient for short term security.

- *Attack on the C^{*--} structure.* This attack [515] requires $\mathcal{O}(q^r)$ operations, which is more than 2^{80} Triple-DES operations for the parameters of SFLASH.
- *Resolution of a random set of quadratic equations.* This is called the MQ problem and is NP-hard. However, Gröbner basis finding [128], XL or FXL algorithms [164] or the more sophisticated F_5 and $F_5/2$ algorithms of Faugère [224] are relatively efficient at solving systems of polynomial equations over finite fields.
  At the time when NESSIE made its selection, for the parameters of SFLASH, those attacks required more computational power than 2^{80} Triple-DES operations. However, some improvements have been published since then [156] and the size of the parameters of SFLASH have been increased to resists this new attack [163].

The hash function of SFLASH (new version). In the specifications for SFLASH the message is hashed to the value $\mathsf{H}(m)$ defined to be the first bits of SHA-1(m)‖SHA-1(SHA-1(m)). The output of the hash function H is clearly not uniform random in $(\mathbb{F}_q)^{n-r}$. A better design is to take the first bits of SHA-1(SHA-1(m)‖0)‖SHA-1(SHA-1(m)‖1).

Other comments about the design. The value $\mathsf{prf}_\Delta(h)$ is defined to be the first bits of SHA-1($h \| \Delta$). This is an acceptable family of pseudo-random functions. HMAC may be preferred.

The affine part of $\mathbf{s}$ and $\mathbf{t}$ can be recovered from the public key alone, and therefore appears to be useless [253, 254].

Side-channel attacks. For a general introduction to side-channel attacks, the reader is referred to the Annex. A. When a message is signed with SFLASH, most of the computations are performed in finite fields of characteristic 2. This makes it difficult to imagine, for example, a fault attack where one would change a bit of the secret key and try to use the erroneous signature to recover the original value of the bit. However, other methods might exist to gain information about the secret key, for instance using differential power analysis (DPA) [595]. The submitters proposed in [12] to mask all the intermediate data with random values.

7.4.4 QUARTZ

QUARTZ is a CPC-D scheme based on the intractability of a variant of the HFE problem.

7.4.4.1 The design

- *Domain parameters.* The parameters are four integers d, n, v and r, a security parameter l, a hash function H with output in $\{0,1\}^{n-r}$ and a family of pseudo-random functions prf with l-bit index, input in $\{0,1\}^{n-r}$ and output in $\{0,1\}^{r+v}$.

  The function $\phi : \mathbb{F}_{2^n} \to \{0,1\}^n$ fixes a representation of $\mathbb{F}_{2^n}$ as a $\{0,1\}$-vector space. For any function $\mathbf{F}$ on the finite field $\mathbb{F}_{2^n}$ let $\mathbf{f} = \phi^{-1} \circ \mathbf{F} \circ \phi$.

  For QUARTZ we have $l = 80$, $d = 129$, $n = 103$, $v = 4$ and $r = 3$ and H and prf based on SHA-1.
- *Key generation algorithm.* A random affine bijection $\mathbf{s}$ of $\{0,1\}^{n+v}$, a random affine bijection $\mathbf{t}$ of $\{0,1\}^n$, a random family $(\mathbf{F}_V)_{V\in\{0,1\}^v}$ of polynomials over $\mathbb{F}_{2^n}$ and a random l-bit value Δ are generated. They are the private key.

  Each polynomial $\mathbf{F}_V$ is randomly chosen from the polynomials of degree at most d whose monomials $\{x^{2^a+2^b}\}_{a,b\geq 0}$ have constant coefficient, $\{x^{2^a}\}_{a\geq 0}$ have coefficient linear in V, and x^0 has coefficient quadratic in V.

  The public key is $P = (P_1, ..., P_{n-r})$ where $(P_1, ..., P_n)$ are the quadratic polynomials that describe the function$(\mathbf{t} \circ \mathbf{f}_V \circ \mathbf{s}) : \{0,1\}^{n+v} \to \{0,1\}^n$.
- *Signing and verification algorithms.* QUARTZ is exactly the CPC-D design with 4 rounds based on the trapdoor function $P : \{0,1\}^{n-r} \times \{0,1\}^{r+v} \to \{0,1\}^{n-r}$. The appendix is an element of $\{0,1\}^{n-r} \times (\{0,1\}^{r+v})^4$, with length $n + 3r + 4v = 128$ bits.

  The signer can compute P^{-1} because the knowledge of $\mathbf{F}_V$ is sufficient to compute a preimage for $\mathbf{f}_V$ and $P^{-1}(x) \in \{\mathbf{s}^{-1} \circ \mathbf{f}_V^{-1} \circ \mathbf{t}^{-1}(x, R)$ for $R \in \{0,1\}^r$, $V \in \{0,1\}^v$ $\}$. The probability that the preimage does not exist for fixed R and V is approximately $\frac{1}{e}$, so the average number of attempts is e and the probability that the signing algorithm fails is $\frac{1}{e^{128}} \simeq 2^{-185}$.

7.4.4.2 Security analysis (see also Sect. 7.3.1)

General security analysis. The preimage resistance of the function P is not well defined, but two techniques to find preimages can be described.

- *Attack on the QUARTZ structure.* If the degree of $\mathbf{F}_V$ is not bounded, then the function P is a random set of quadratic polynomials, but the signing time is too long. To improve the efficiency of the signing algorithm (which is still very slow) the maximal degree in QUARTZ is set some value d. No known attack directly exploits this difference between the QUARTZ problem and the MQ problem.
- *Resolution of a random set of quadratic equations.* This is called the MQ problem and is NP-hard. However, Faugère [225] showed experimentally that the classical algorithm to solve a system of polynomial equations (Gröbner basis finding) is much more efficient on QUARTZ systems than on general systems.
  The impact of this result has been studied by Courtois *et al.* [157] and their conclusion is that setting $d = 257$ increases the computational power of Faugère's attack to 2^{78}. The price to pay is a signing algorithm that is more than 3 times slower.

The hash function of QUARTZ. In the specifications for the first version of QUARTZ [159], the value $\mathsf{H}(m)$ is defined to be the first bits of $h_1 \| h_2 \| h_3$, where $h_1 = \text{SHA-1}(m)$, $h_2 = \text{SHA-1}(h_1)$ and $h_3 = \text{SHA-1}(h_2)$. The revised version of QUARTZ [161] uses the better design that takes the first bits of $\text{SHA-1}(h\|0)\|\text{SHA-1}(h\|1)\|\text{SHA-1}(h\|2)$, where $h = \text{SHA-1}(m)$.

Other comments about the design. The signature generation algorithm in the first version of QUARTZ only accepts the case where $\mathbf{F}_V^{-1}$ has a unique solution. The revised version also accepts the case where two solutions exist and chooses one in a deterministic way. The drawback of the first version is that the appendix space is restricted to the values where $\mathbf{F}_V$ has only one root, and that gives some information about $\mathbf{F}_V$ that might be usable for an attack.

Another drawback of Quartz is that the scheme is malleable (it is not strongly unforgeable) [328].

Side-channel attacks. Side-channel attacks are introduced in Annex A. The signing algorithm of QUARTZ does not use any of the classically vulnerable operations, and we did not find any side-channel attack against it.

7.4.5 RSA-PSS

RSA-PSS as submitted by RSA Labs [326] is based on the PSS design, and its security mainly relies on the intractability of the e-th root problem.

The differences between the submitted RSA-PSS and the generic design described in Sect. 7.3.1.5 of this document and in the original description of PSS [54] are justified in the submission and in supporting documents [322, 326].

7.4.5.1 The design

- *Domain parameters.* The parameters are the bitlength l of the modulus, the output size l' of a hash function $Hash$ and the public exponent $e \geq 3$.
- *Key generation algorithm.* Two random primes p and q are generated. The public key is $n = pq$ and the private exponent is $d = e^{-1} \bmod (p-1)(q-1)$.
- *Signing and verification algorithms.* The scheme uses the PSS design where the trapdoor function is $\mathsf{f}(x) = x^e \bmod n$ and where the two functions H and G are built from the common hash function $Hash$. A hash identifier may be used, and the complete definitions of H and G are given in Sect. 7.4.5.2 below.

  One consequence of this complete definition is that the trapdoor function f is used only on a subset of $\mathbb{Z}/n\mathbb{Z}$. More precisely, if we define $\mathcal{H}_0 = \{x \in \mathbb{Z}/n\mathbb{Z} \,|\, x = a\|b\|0\text{xBC}\}$, $\mathcal{H}_{\mathsf{Id}} = \{x \in \mathbb{Z}/n\mathbb{Z} \,|\, x = a\|b\|\mathsf{Id}\|0\text{xCC}\}$, $\mathcal{H} = \mathcal{H}_0 \cup \bigcup_{\mathsf{Id}} \mathcal{H}_{\mathsf{Id}}$ and $\mathcal{S} = \{x^{1/e} \,|\, x \in \mathcal{H}\}$, then the NESSIE submission is a PSS construction based on the restriction $\mathsf{f} : \mathcal{S} \to \mathcal{H}$.

Parameter and key generation. While any exponent $e \geq 3$ can be used, the NESSIE submission proposes small values like 3, 17, or 65537. The advantage of choosing a small e is that the verification algorithm is much faster.

The generation of random prime numbers p and q of a given size is crucial to the security of the scheme. The NESSIE submission suggests an algorithm that does a probabilistic primality testing. Other algorithms may be used for key generation.

7.4.5.2 Security analysis (see also Sect. 7.3.1)

Security proof for RSA-PSS. Following the generic security proof for the PSS design, the security of RSA-PSS is based on the clawfreeness of f and $\mathsf{g}(x) = \eta \cdot x^e \bmod n$ for a random η. This is easily shown to be equivalent to the e-th root problem for η.

But because the NESSIE submission is based on a restriction of the function f, the efficiency of the security proof is slightly worse than for the generic PSS design. This is shown by Jonsson [322].

On the hash functions of RSA-PSS. The description of the function H depends on an option t which can be 1 or 2 and decides whether a hash identifier [7] is used or not. If $t = 1$ and $Hash$ is a hash function with $hLen$-octet output, then H has an output of $hLen + 1$ octets and is defined by $\mathsf{H}(m\|r) = Hash(0_{(8\text{ octets})}\|Hash(m)\|r)\|0\text{xBC}$. If $t = 2$ and $Hash$ is a hash function with $hLen$-octet output and identifier Id, then H has an output of $hLen + 2$ octets and is defined by $\mathsf{H}(m\|r) = Hash(0_{(8\text{ octets})}\|Hash(m)\|r)\|\mathsf{Id}\|0\text{xCC}$. The output of the function $\mathsf{G}(h)$ is defined to be the first $\lfloor l/8 \rfloor - hLen - t$ octets of $Hash(h\|0_{(4\text{ octets})})\|Hash(h\|1_{(4\text{ octets})})\|...$ Moreover, the salt r being of length smaller than $\lfloor l/8 \rfloor - hLen - t$ octets, a constant $\bar{m} = 0...01$ is used.

While this is very close to the generic PSS design, the fact that H and G have variable output length, depending on t, makes a new proof necessary and may introduce subtle weaknesses. Another very similar design is easier to

[7] Some thoughts on hash identifiers have been published by Kaliski [334].

provide with a security proof, if the function H has a fixed output length for each public key. For example, $\mathsf{H}(m\|r)$ can be defined to be the last octets of $...\|Hash(1_{(8\text{ octets})}\|Hash(m)\|r)\|Hash(0_{(8\text{ octets})}\|Hash(m)\|r)\|\mathsf{Trail}$, where Trail is either 0x00BC or $\mathsf{Id}\|$0xCC. The security proof of this variant of RSA-PSS then makes the hypothesis that this function can be viewed as a random oracle (with output in $\mathcal{H}$), which is likely if $Hash$ is one-way and collision-resistant. [8]

Another improvement on the functions H and G can be suggested, which protects against parameter manipulation, especially against adversarial hashing [444]: the inclusion of some commitment to the parameters and the public key in the input of the hash functions, as suggested in Sect. 7.2.1.2.

One last improvement is the use of UOWHF (Universal One-Way Hash Functions) instead of CRHF (Collision-Resistant Hash Functions). The reason behind this is that there exist secure constructions of families of UOWHF, while no family of CRHF has been found.

RSA problem versus *e*-th root problem. It is important to notice that because the exponent e is a parameter of the scheme the security of RSA-PSS is provably based on the e-th root problem.

Some implementations of RSA-based schemes generate n before choosing e. If e is not randomly chosen from the numbers coprime to $\phi(n)$ (e.g. the implementation of GPG 1.2.1 `www.gnupg.org` chooses the smallest such e in the list 41, 257, 65537, 65539, ...) the security of the scheme is based on another intractability assumption.

However, flexibility in the choice of e can be useful for some RSA-based schemes, e.g. threshold RSA and mediated RSA [105, 190].

It is possible that the e-th root problem for small values of e like 3, 17, or 65537 has not the same intractability as the generic problem. It may be easier, or harder to solve. But small exponents make this problem easier to solve if some side-channel information can be obtained, e.g. some bits of the secret exponent d [107]. Moreover, some other theoretical arguments have been given that suggest that a prime $e \geq 65537$ is a conservative choice [110].

The random nonce. The random value r used in the signing algorithm should have *entropy*, otherwise the efficiency of the security proof decreases. If this value is deterministically generated, then the scheme is equivalent to an RSA-FDH scheme, and has not so tight reduction to the e-th root problem. Even if the deterministic generation of r is not pseudo-random, e.g. if $r = 0$, no concrete weakness of the resulting scheme is known.

Side-channel attacks. A general introduction to side-channel attacks can be found in Annex. A. In [104], Boneh *et al.* stressed how Chinese remainder based implementations of RSA signature schemes are vulnerable to faults. For efficiency, one would compute separately $S_p = x^d \bmod p$ and $S_q = x^d \bmod q$ and then use the Chinese remainder theorem to construct the signature $S = x^d \bmod n$. If an error occurs during only one of the two exponentiations, then an attacker who

[8] This design may even be better than the simple design $\mathsf{H}(m\|r) = Hash(m\|r)$, which has the extensibility property [127].

obtains this faulty signature and the correct one of the same message can factor the modulus. This attack has been improved by Joye *et al.* in [330]: one such erroneous signature and the corresponding plaintext x are enough to find out the entire secret exponent. This attack does not apply to RSA-PSS. Indeed, in this scheme, the message is at first encoded using the PSS encoding method, which introduces some random bits. Thus the user never signs the same message twice, and so the first version of the attack is avoided. Furthermore, given an erroneous signature, the full message x that has been signed cannot be recovered, so the second version of the attack does not work either.

For the same reason, it seems very hard to mount against RSA-PSS a fault based attack (based on [349]) that introduces faults in the secret exponent.

7.5 Digital signature schemes not selected for Phase II

7.5.1 ACE-Sign

7.5.1.1 The design

ACE-Sign is based on the merge of a RAND-secure scheme and of a secure chameleon hash, as described in Sect. 7.3.3.4.

- *Domain parameters.* The security parameters are two integers l (e.g. 160) and l' (e.g. 512). Let H be a hash function with an l-bit output.
- *Key generation algorithm.* Two l'-bit strong primes p and q are randomly chosen, and $n = pq$ is computed. h and x are randomly taken in QR_n and e' is a randomly chosen $(l+1)$-bit prime. The keys are $\mathsf{pk} = (n, h, x, e')$ and $\mathsf{sk} = (p, q)$.
- *Verification algorithm.* The verification of a signed message $(m, e, y, y') \in \mathcal{M} \times \mathbb{Z} \times \mathbb{Z}/n\mathbb{Z} \times \mathbb{Z}/n\mathbb{Z}$ checks that e is an odd $(l+1)$-bit integer different from e'. Then it computes $x' = (y')^{e'} h^{-\mathsf{H}(m)}$ and checks if $x \stackrel{?}{=} y^e h^{-\mathsf{H}(x')}$.
- *Signing algorithm.* A random $y' \in QR_n$ and a random $(l+1)$-bit prime e are generated, and $x' = (y')^{e'} h^{-\mathsf{H}(m)}$ and $y = (xh^{\mathsf{H}(x')})^{1/e}$ are computed. The appendix is (e, y, y').

In fact, ACE-Sign as submitted to NESSIE does not use a collision resistant function H but a family of universal one-way hash functions (UOWHF). The index of the hash function used is added to the signed message. This increases its size but weakens the security hypothesis on the hash.

Moreover, the random prime e is generated by some specific randomised algorithm that output e together with some certification values. The properties of this algorithm are: the probability that the same e is generated twice is negligible, and it is intractable to generate e and the certification without using this algorithm. This modification increases the size of the signed message but allow to reduce the security of ACE-Sign to the RSA assumption in the random oracle model.

7.5.1.2 Security analysis (see also Sect. 7.3.3)

General security analysis. We can apply the general security result for this construction (cf. Sect. 7.3.3.4), but ACE-Sign is also provided with an explicit security proof, under the hypothesis that H is collision resistant (or taken from a family of UOWHF).

If the forgery is (m, e, y, y') we define $x' = (y')^{e'} h^{-\mathsf{H}(m)}$, and for all answers (m_i, e_i, y_i, y'_i) to signature queries we define $x'_i = (y'_i)^{e'} h^{-\mathsf{H}(m_i)}$. This proof defines three types of forgeries:

- In a type I forgery for some j we have $e = e_j$ and $x' = x'_j$. It leads to a solution of the RSA problem for (n, z, e') with tight reduction. This corresponds to a forgery of the chameleon hash. One can notice that $e = e_j$ is not used in the proof.
- In a type II forgery for some j we have $e = e_j$ but $x' \neq x'_j$. It leads to a solution of the RSA problem for (n, z, e_j) with not so tight reduction. This corresponds to the GUESS forgery of the RAND scheme.
- In a type III forgery for all i we have $e \neq e_i$. It leads to a solution of the strong RSA problem for (n, z) with tight reduction. This corresponds to the FLEX forgery of the RAND scheme.

The random nonces. The random value y' used in the signing algorithm and the randomness needed to generate e only need to be *unpredictible*. These values can be deterministically generated from the message and a secret value, like it is done for the FDH-D design (cf. Sect. 7.3.1.3).

Comments about the design. ACE-Sign is the first of a family of digital signature schemes which have a security proof in the real world and for which all the operations used for signature or verification are efficient. While this design is very promising, ACE-Sign has the drawback of having a not so tight reduction. This is also the case for all the variants of this scheme.

For this reason its advantage over RSA-PSS, namely the fact that the security proof holds in the real world, is only meaningful if parameter size is increased to counteract the non-optimal reduction. For example, to have 80-bit security with $q_S \simeq 2^{32}$, ACE-Sign would need a modulus of 4096 bits, where RSA-PSS only needs 1536 bits. The impact in terms of performance is important.

7.5.2 FLASH

7.5.2.1 The design

This scheme is very similar to SFLASH (new version). The only change is in the parameter choice, where $q = 256 = 2^8$ instead of 2^7. See also [513].

7.5.2.2 Security analysis

There is no known difference in security between FLASH and SFLASH (new version) and the performance is similar. However, the fact that 7 is prime implies that no other subfield can be found in $\mathbb{F}_{128}$ than $\mathbb{F}_2$, while the FLASH base field $\mathbb{F}_{256}$ can also be seen as a vector space over $\mathbb{F}_{16}$ or $\mathbb{F}_4$. This may introduce additional weaknesses in FLASH.

7.5.3 SFLASH (old version)

7.5.3.1 The design

This scheme is very similar to SFLASH (new version). The only change is in the secret key choice, where all components of **s** and **t** are 0 or 1. The advantage of this restriction is that the keys are much shorter than for SFLASH (new version).

7.5.3.2 Security analysis

Gilbert and Minier [259] have found how to use this special property to break SFLASH (old version). Their attack is practical, so SFLASH (old version) is totally insecure.

8. Digital identification schemes

8.1 Introduction

We present in this chapter asymmetric techniques to identify one entity (the prover, also called Alice) with regard to another (the verifier, also called Bob), in a way that any attacker (also called Charlie) would most certainly fail to substitute himself for Alice. We will first review some standard techniques to achieve this goal, starting from the least secure. Then we will see how to prevent a wide variety of active attacks through the notion of zero-knowledge and witness-indistinguishable properties. Finally we will scrutinise the only submission in this category, namely GPS, comparing it with some existing standards.

8.1.1 Identification through Password

The most widespread identification protocol is the identification through a password. For instance, under a UNIX system, it works as follows. Every user U chooses a password x_U and sends it to the server. The server applies a one-way function f (based on the DES block cipher for UNIX) to compute $y_U = f(x_U)$ and then stores (U, y_U). Each time a user wants to access his account, he will send his password x to the server that will then check whether $y_U = f(x)$.

Unfortunately, this identification does not satisfy the modern security requirements. Indeed any eavesdropper will be able to recover the secret password (which is sent unencrypted) therefore causing the scheme to be totally vulnerable to a passive attack (cf. Def. 8.2).

8.1.2 Lamport's Protocol

This protocol is akin to the previous password protocol, with the difference that any password is only used once. The price to pay is a larger storage capacity, or an enhanced computational power. It works as follows.

– Initialisation
  1. Every user U chooses and saves a secret value $x_U = x_0$.
  2. He then computes $x_1 = f(x_0), x_2 = f(x_1), \ldots, x_{1000} = f(x_{999})$, where f is as before a one-way function.
  3. He then publishes $y_U = x_{1000}$ to the server (verifier) who stores (U, y_U).

[0] Coordinators for this chapter: UCL — Mathieu Ciet and Francesco Sica

– Identification
1. When first identifying himself, the prover sends, together with his identity, $x = x_{999}$. The verifier then checks that $f(x) = y_U$ and updates the value of y_U by setting $y_U = x$.
2. Successive identifications are carried out as previously, with the prover sending $x_{998}, x_{997}, \ldots$ and so on and the verifier updating the value of y_U after each identification. Since f is supposed one-way, only the prover is able to produce this sequence of values.

There are two problems with this kind of identification. Since successive values $x_{1000}, x_{999}, \ldots$ become public after doing several executions of the protocol, one cannot safely replay this protocol. In particular, in case of disk crash, if the last value sent (say x_{765}) was not stored, but say only up to x_{900} was stored, anyone having recorded any value between x_{899} and x_{765} will be able to identify himself instead of the prover if the server replays previous identification rounds.

Another issue lies in the fact that this identification scheme can only be used with one verifier (this is a problem for electronic transactions, for instance).

8.2 Security Requirements

8.2.1 Passive Attacks and Interactive Proofs

As we saw in the previous section, a major threat is represented by replay attacks, that are the most dangerous passive attacks. To thwart the possibility of passive attacks, one solution is to make the identification protocol interactive with a series of questions and answers between Alice and Bob. The minimal properties of this protocol against any attack are the completeness and the soundness. They are explained in the following.

Definition 8.1 (Goal of attacker). *An identification scheme is* totally broken *by an attacker if the attacker can impersonate the legitimate prover at will (to any verifier) using his public key.*

Definition 8.2 (Passive and active attacks). *A* passive attack *involves an adversary who tries to break the identification scheme by simply recording data exchanged between prover and verifier and thereafter analysing it. An* active attack *involves an adversary who can deviate from the verifier's protocol in order to extract more information about the secret key.*

Definition 8.3 (Completeness property). *An interactive proof is called* complete *if given a honest prover and a honest verifier, the verifier accepts the prover with overwhelming probability.*

Definition 8.4 (Soundness property). *An interactive proof is called* sound *if whenever Charlie tries to impersonate Alice during the identification protocol, he will fail with overwhelming probability.*

How do we build in practice such interactive proofs of knowledge? Asymmetric cryptography comes to our help: Alice will be the holder of a secret key S_A, with corresponding public key I_A. This public key is a word of a language $L \in \mathsf{NP}$ (see start of Sect. 8.2.5 for a definition of NP). The secret key S_A *witnesses* that $I_A \in L$. One chooses L in such a way that finding such a witness is untractable. The identity proof consists in Alice trying to persuade Bob that she possesses such a witness S_A. Also if Charlie wants to successfully identify himself, he will have to know a witness.

We can reformulate the soundness property by making it more precise.

Definition 8.4b (Soundness property). *An interactive proof is called* sound *if there exists an expected polynomial-time algorithm M with the following property: if an attacker impersonating the prover can, with non-negligible probability, successfully execute the protocol and be accepted by the verifier then M can recover a prover's witness using the attacker as a subalgorithm.*

We will now present a few mathematical problems on which such interactive proofs can be based.

8.2.2 Trusted Hard Mathematical Problems

These problems can be divided into two classes, namely the popular number-theoretic problems (RSA, discrete logarithm, ...) which are not NP-complete but extensively studied, and proven NP-complete problems (PKP, SD and following, see below). NP-complete languages L are the natural candidates for identification tasks, because by definition if $L \notin \mathsf{P}$ (supposing $\mathsf{P} \neq \mathsf{NP}$) then checking whether a word x belongs to L cannot be done in polynomial time, whereas the knowledge of a witness (certificate) y allows to do so.

Let us describe these problems (cf Section 6.2.3).

- Let g be an element of a group, and let g have order q. The discrete logarithm problem (DLP) is the problem of finding a when given (g, g^a). The DLP assumption is that the DLP cannot be solved by a polynomial-time (with respect to q) algorithm. We will use the version where the group is the set $(\mathbb{Z}/n\mathbb{Z})^\times$ of invertible residues modulo n.
- Short exponent problem: this is a sub-instance of the discrete logarithm problem. Given two coprime integers n, g, an integer $S << \varphi(n)$ and $g^s \mod n$ the problem consists of recovering the exponent s, knowing that $s \leq S$.
- An RSA key is a pair (n, e) where $n = pq$ with p and q primes, and $1 \leq e < n$ with $g.c.d.(n, e) = 1$. The RSA problem is the problem of finding an integer $1 \leq x \leq n$ such that $x^e = y$ when given an RSA key (n, e) and a randomly selected integer $1 \leq y \leq n$.
- The e-th root problem is similar to the RSA problem except the exponent e is thought of as a constant. The e-th root problem is the problem of finding an integer $1 \leq x \leq n$ such that $x^e = y$ when given a modulus $n = pq$ (with p and q primes) and a randomly selected integer $1 \leq y \leq n$. A special case of this problem is the square root problem, when $e = 2$.

- RSA-omi (one more inversion) problem: given two integers n, e such that $\gcd(e, \varphi(n)) = 1$, a challenge oracle outputting a random $y \in (\mathbb{Z}/n\mathbb{Z})^\times$ for each query and an inversion oracle which on input $y \in (\mathbb{Z}/n\mathbb{Z})^\times$ outputs x such that $y \equiv x^e \pmod{n}$, the problem is for an adversary making t queries to the challenge oracle and getting $y_1, \ldots, y_t$ to find $x_1, \ldots, x_t$ such that $y_i = x_i^e \pmod{n}$ for $i = 1, \ldots, t$ with *at most* $t - 1$ queries to the inversion oracle.
- One-more discrete logarithm (omdl) problem: this is defined similarly to the RSA-omi, where the inversion oracle is replaced by a discrete logarithm oracle.
- PKP (Permuted Kernel Problem): Given an $m \times n$ matrix A with entries in $\mathbb{Z}/p\mathbb{Z}$ and a vector $(v_1, \ldots, v_n) \in (\mathbb{Z}/p\mathbb{Z})^n$, find a permutation σ (if it exists) on the set $\{1, \ldots, n\}$ such that $(v_{\sigma(1)}, \ldots, v_{\sigma(n)})$ lies in the kernel of A.
- SD (Syndrome Decoding problem): Given a $(n, n-k)$-linear binary code with parity matrix H and a k-bit vector i, find a minimum-weight solution to the linear equation $H(s) = i$.
- CLE (Constrained Linear Equations problem): This is the problem of finding solutions to a system of linear equations modulo a small prime, when imposing that the solutions must be taken from a specified set of residues.
- PPP (Permuted Perceptrons Problem): Define an ϵ-matrix (resp. vector) to have all entries equal to ±1. Let A be an ϵ-matrix of size $m \times n$ and S be a set of m nonnegative integers. The problem consists in finding an ϵ-vector V of size n such that the set of the components of the vector AV is equal to S.

8.2.3 Protection against Active Attacks

The reason to base interactive protocols on hard problems is to offer resistance to the more pernicious active attacks. We can follow the paradigm of asymmetric schemes proofs in order to show that a given scheme is secure under active attacks. The following definitions are adaptations of the similar Def. 6.2 and the definition found at the beginning of Sect. 7.2.3.

We point out that an active attacker will usually act as the verifier to Alice for a number of times and then try to impersonate her successfully with other (honest) verifiers.

Definition 8.5. *A* (t, ε)-solver *for a problem is a probabilistic Turing Machine* $\mathcal{A}$ *that runs in time bounded above by* t *and outputs a solution for the problem with probability at least* ε.

A (t, ε, q_I)-impersonator *for a digital identification scheme is a probabilistic Turing Machine* $\mathcal{A}$ *that runs in time bounded above by* t*, makes at most* q_I *challenges to the prover and succeeds in breaking the scheme with probability at least* ε.

Definition 8.6. *A* proof of security *is the description of a (randomised) algorithm called* reduction algorithm*. This algorithm is a* (t', ε')*-solver for some mathematical problem and interacts with a* (t, ε, q_I)*-impersonator for the identification scheme. A proof of security explains how a solver, using a* (t, ε, q_I)*-impersonator as a subalgorithm, can solve the underlying mathematical problem* *in time t' with probability ε'. The proof must relate t to t' and ε to ε'. In particular, $\varepsilon' \leq \varepsilon$ and $t' \geq t$.*

Thus if the underlying mathematical problem is hard, so that there exists no (t', ε')-solver for it, then there cannot exist any (t, ε, q_I)-impersonator of the identification scheme. We say that the identification scheme is (t, ε, q_I)-secure.

Usually $q_I \ll t$, and we will require $k = \log_2(t/\varepsilon) = 80$ (the attacker's computing power is estimated to be 2^{80} triple-DES executions) and $\log_2 q_I = 30$ (the attacker cannot require more than a billion executions of the identification protocol).

However, some algorithms until recently did not have any security proof according to this paradigm, and even the known proofs [51] use some strong underlying assumption, like the RSA-omi for GQ2 or the one-more discrete logarithm for Schnorr.

This is the reason for which the zero-knowledge property proposed by Goldwasser, Micali and Rackoff [272] is preferred as a shield against active attacks.

8.2.4 Zero-Knowledge

8.2.4.1 The Power of Zero-Knowledge Interactive Proofs

Is it possible for Alice to convince Bob that she knows a secret S_A without revealing anything other than this fact? The theory of zero-knowledge gives an affirmative answer. In other words, during a zero-knowledge identification protocol, Alice is effectively sending Bob only one bit of information, namely that she is indeed who she claims she is.

This section is largely inspired from Goldreich's book [265, Chapter 6], to which we refer for more technical details and proofs. The following is a formalisation of Alice and Bob and some rephrasing of concepts already seen. Note that one can view, in light of the preceding sections, an interactive identification protocol as a proof that some word x belongs to a language L.

Definition 8.7 (Loose formalisation of prover and verifier). *Alice and Bob are* interactive Turing machines *(ITM), denoted respectively P and V. Each of these Turing machines has a certain number of tapes, most important of which are a common input tape (read-only), auxiliary input tapes (read-only and specific to each), random seed or coin tapes (read-only and specific to each), work tapes (read-write and specific) and two communication tapes (shared, on one of them P can read and V can write, on the other one V can read and P can write).*

Also, P and V have two states (active and idle) and they cannot be both active (they are both idle only before and after the identification protocol).

Definition 8.8 (Time complexity of ITMs). *Let A and B be two linked ITMs and $t\colon \mathbb{N} \to \mathbb{N}$ a non-decreasing function. Then A has* time complexity t *if for any B and any bit string x, machine A on interacting with B with common input x always (i.e. for any distribution of outputs of tapes specific to B) halts within time $t(|x|)$, where $|x|$ denotes the bit length of x.*

We say A has polynomial-time *complexity if t can be chosen to be a polynomial.*

Definition 8.9. *We denote by $\langle A(y), B(z)\rangle(x)$ the random variable consisting of the local output of B after interacting with A, over all choices of the random seed tapes, with common input x and auxiliary inputs y for A and z for B.*

Definition 8.10 (Completeness and soundness revisited). *The identification protocol between P and V which is an interactive proof system for a language L is*

– complete *if for every common input $x \in L$, there exists y such that for any z*

$$\mathrm{Prob}(\langle P(y), V(z)\rangle(x) = 1) \geq \frac{2}{3} \ , \tag{8.40}$$

– sound *if for every $x \notin L$, for any ITM B and for any bit strings y, z*

$$\mathrm{Prob}(\langle B(y), V(z)\rangle(x) = 1) \leq \frac{1}{3} \ .$$

Remark. *The numbers* 2/3 *and* 1/3 *can be equivalently replaced in this definition by $1-\epsilon$ and ϵ for any $0 < \epsilon < 1/2$. Also, given $x \in L$, we denote $P_L(x)$ to be the set of y satisfying* (8.40)*.*

We come to the main definition of this section.

Definition 8.11 (Zero-knowledge property). *Let (P, V) be an interactive proof for a language L. We say (P, V) is* auxiliary-input zero-knowledge *if for every probabilistic polynomial time interactive machine V^* there exists a probabilistic algorithm $M^*(\cdot, \cdot)$, running in polynomial time with respect to its first input, so that the following two distributions are indistinguishable:*

– $\{\langle P(y), V^*(z)\rangle(x)\}_{x\in L, y\in P_L(x), z\in\{0,1\}^*}$ *and*
– $\{M^*(x, z)\}_{x\in L, z\in\{0,1\}^*}$.

In this case, M^* is called a simulator *of the interaction of V^* with P.*

What this says is that the interaction of P and V can be simulated polynomially by an algorithm controlled by V, with the same inputs (to V). It is important that this simulation be polynomial-time, as we will later see. Also note that an eavesdropper seeing only the interaction between P and V will be unable to tell whether he is witnessing a real interaction (variable $\langle P(y), V^*(z)\rangle(x)$) or a fake one ($M^*(x, z)$).

In this definition, one has to give a meaning to the term indistinguishable. If the two distributions are truly the same, the property is called *perfect* zero-knowledge. But in practice, given the limited power an attacker can have, the weaker notion of *computational* zero-knowledge offers enough security.

Definition 8.12 (Computational indistinguishability). *The distributions defined by the random variables $\{\langle P(y), V^*(z)\rangle(x)\}_{x\in L, y\in P_L(x), z\in\{0,1\}^*}$ and*

$\{M^*(x,z)\}_{x \in L, z \in \{0,1\}^*}$ are computationally indistinguishable if for every probabilistic algorithm $D(\cdot,\cdot,\cdot)$ running in polynomial time with respect to its first input, every polynomial p, all sufficiently long $x \in L$, all $y \in P_L(x)$ and $z \in \{0,1\}^*$, one has

$$|\mathrm{Prob}(D(x,z,\langle P(y),V^*(z)\rangle(x)) = 1) - \mathrm{Prob}(D(x,z,M^*(x,z)) = 1)| < \frac{1}{p(|x|)} \ .$$

Remark. *By zero-knowledge we will henceforth mean computational zero-knowledge.*

Note that for the time complexity of ITMs and hence M^* and D we require polynomial-time only in the common input x, not in the auxiliary inputs z or y which can be very large. If we did, then we would allow for instance D's that could run in exponential time and worse, in practice shredding the notion of computational indistinguishability.

The interesting property of zero-knowledge proofs is that they remain zero-knowledge after polynomially many repetitions of the protocol.

Theorem 8.1 (Closure under sequential composition). *Let (P,V) be an interactive auxiliary-input zero-knowledge proof for a language L. Let Q be a polynomial. Define P_Q to be the sequential running of P with same x as common input $Q(|x|)$ times and independent random tapes (similarly for V_Q). Then (P_Q,V_Q) is an auxiliary-input zero-knowledge proof for L. Moreover if (P,V) is perfect auxiliary-input zero-knowledge, then so is (P_Q,V_Q).*

Remark. *Actually, V plays no role here, because it is clear that the zero-knowledge property is a property of the prover P only.*

One should be aware here that the assertions of the theorem are not necessarily valid if one uses alternative definitions of zero-knowledge, as the original definition of Goldwasser, Micali and Rackoff [272]. In particular, allowing auxiliary inputs is essential in order to get closure under sequential composition.

This theorem is used in practice to construct protocols where the probability of cheating is as low as one desires. For instance, in the Fiat-Shamir protocol (see Sect. 8.4.1) a legitimate prover is always accepted, but a cheater can be accepted with probability 1/2. Therefore by repeating the basic 3-round protocol a number k of times, one can ensure that a cheater will only be accepted with probability $1/2^k$, while retaining the zero-knowledge character of the protocol.

A very nice result is that all languages in NP have a zero-knowledge proof provided one-way functions exist.

Theorem 8.2. *Suppose one-way functions exist. Then every language in NP has an auxiliary-input zero-knowledge proof system. Furthermore, the prescribed prover in this system can be implemented in probabilistic polynomial-time, provided it gets the corresponding NP witness as auxiliary input.*

8.2.4.2 Negative Results on Zero-Knowledge

The theory of zero-knowledge does not solve all our problems, since we will see that to obtain adequate security, zero-knowledge protocols have to be repeated sufficiently many times and thus lose in efficiency.

We begin by recalling the definition of the complexity class BPP.

Definition 8.13 (BPP). *The class* BPP *consists of all languages L for which there exists a polynomial-time randomised algorithm V such that for all x*

$$\mathrm{Prob}(V(x) = 1 \mid x \in L) \geq \frac{2}{3}$$

and

$$\mathrm{Prob}(V(x) = 1 \mid x \notin L) \leq \frac{1}{3} \ .$$

As before, the numbers appearing in the right-hand sides can be made arbitrarily close to 1 (resp. 0). It is clear that languages in this class have trivial interactive proofs of knowledge, in the sense that the verifier has no interaction with the prover.

Definition 8.14 (Black-box zero-knowledge). *The prover P for the language L is* black-box zero-knowledge *if there exists a universal probabilistic polynomial-time algorithm M which uses any verifier V^* as a black box, such that $\{\langle P(y), V^*(z)\rangle(x)\}_{x\in L, y\in P_L(x), z\in\{0,1\}^*}$ and $\{M^{V^*(z)}(x)\}_{x\in L, z\in\{0,1\}^*}$ are computationally indistinguishable.*

Hence the main difference with traditional zero-knowledge definitions is that the simulator here is universal and does not depend on the verifier V^*. In practice, all known proofs of zero-knowledge proceed by exhibiting a universal M, so that in the end, one always demonstrates black-box zero-knowledge properties. However this theoretical definition is important because of the following result.

Theorem 8.3. *Suppose that (P, V) is an interactive proof system for the language L with negligible error probability. Suppose also that there exists a constant ρ such that for every $x \in L$, on input x the prover P sends at most ρ messages (constant round), the messages sent by the verifier are* predetermined *consecutive segments of its random tape (public coins or Arthur-Merlin game) and that P is black-box zero-knowledge. Then $L \in$* BPP*.*

Thus, except for "trivial" languages, if (P, V) satisfies all the properties above, it must be that the error probability is not negligible, hence to make it small, you must increase the number of sequential runs of the protocol. Another conclusion is that in order to construct low error probability zero-knowledge protocols it is wise to allow the verifier to use secret coins.

In the cases that we will consider, namely 3-round protocols, the distinction between public and secret tosses of a coin does not exist (since there is only one random toss), hence we get that there are no 3-round black-box zero-knowledge identification systems with negligible error probability. The number of rounds here is believed to be optimal in the sense that there exist 4-round black-box zero-knowledge proofs for languages believed to be outside BPP and assuming the existence of claw-free permutations (see 7.3.1.1) there exist 5-round zero-knowledge interactive proofs for all languages in NP.

There is one important consequence of this fact. If we consider *parallel* executions of the protocol, then the error probability can be made negligible (for instance less than $1/2^m$ for m parallel executions of Fiat-Shamir). If (P, V) play an Arthur-Merlin game, then the same is true for the parallel version and the number of rounds stays the same. Hence the theorem implies that this parallel version cannot be black-box zero-knowledge. Although it may still be zero-knowledge, the fact that all known zero-knowledge proofs are actually black-box zero-knowledge proofs is a bad omen. Indeed the following was proved.

Theorem 8.4 (Non-closure under parallel composition). *There exist two zero-knowledge provers P_1 and P_2 such that the prover P which runs both of them in parallel leaks knowledge.*

This and the fact that zero-knowledge protocols are expensive (in terms of performance) has led to consider a larger class of protocols that includes zero-knowledge protocols but can still be used for secure identification.

8.2.5 Witness Indistinguishability

Let L be a language in NP. By definition, this means that there exists a binary relation, called *witness relation*, R_L such that $(x, y) \in R_L$ implies $|y| \le p_L(|x|)$ for some polynomial p_L. Also, $(x, y) \in R_L$ can be checked in polynomial-time and

$$L = \{x \colon \exists y \colon (x, y) \in R_L\} \ .$$

We call y a witness to $x \in L$. In general, $x \in L$ may have more than one witness, so that the cardinality of $R_L(x) = \{y \colon (x, y) \in R_L\}$ is larger than 1. We say an interactive proof for L is witness-indistinguishable if after running the protocol, the verifier cannot tell which witness the prover used as auxiliary input.

Definition 8.15 (Witness indistinguishability/independence). *Let $L \in \mathsf{NP}$ with a witness relation R_L, (P, V^*) an interactive proof for L, where V^* is any polynomial-time ITM . Denote by $\mathrm{view}_{V^*(z)}^{P(y)}(x)$ a random variable describing the contents of the random-tape of V^* and the messages V^* reads on the communication tape (written by P), on common input x, auxiliary input y (for P) and z (for V^*). We say P is* witness-indistinguishable *if for every V^* and every two sequences $(w_x^1)_{x \in L}$ and $(w_x^2)_{x \in L}$ such that $w_x^i \in R_L(x)$, the following two distributions are computationally indistinguishable*

- $\{x, \mathrm{view}_{V^*(z)}^{P(w_x^1)}(x)\}_{x \in L, z \in \{0,1\}^*}$ *and*
- $\{x, \mathrm{view}_{V^*(z)}^{P(w_x^2)}(x)\}_{x \in L, z \in \{0,1\}^*}$.

If the two distributions are the same, then we use the term witness-independent.

Note that any zero-knowledge proof for a language in NP is witness-indistinguishable, since the view corresponding to each witness can be approximated by the same simulator. Likewise, perfect zero-knowledge proofs are witness independent.

Theorem 8.5 (Closure under sequential and parallel composition). *The sequential composition of witness-indistinguishable (resp. witness-independent) provers is witness-indistinguishable (resp. witness-independent).*

For parallel composition: let $L \in \mathsf{NP}$, R_L *a witness relation,* P *a witness indistinguishable (resp. witness-independent) prover for* R_L *that runs in probabilistic polynomial time. Let* Q *be a polynomial and* P_Q *denote the ITM that on common input* $x_1, \ldots, x_{Q(n)} \in \{0,1\}^n$ *and auxiliary input* $y_1, \ldots, y_{Q(n)} \in \{0,1\}^*$ *invokes* P *in parallel* $Q(n)$ *times, so that in the* i^{th} *copy* P *is invoked with common input* x_i *and auxiliary input* y_i*. Then* P_Q *is witness-indistinguishable (resp. witness-independent) for the relation* R_{LQ} *such that* $\big((u_1, \ldots, u_{Q(n)}), (v_1, \ldots, v_{Q(n)})\big) \in R_{LQ}$ *iff* $(u_1, v_1) \in R_L, \ldots, (u_{Q(n)}, v_{Q(n)}) \in R_L$*.*

Remark. *It is important that* P *be probabilistic polynomial-time.*

Theorems 8.2 and 8.5 together with the observation that zero-knowledge implies witness-indistinguishability can be put together to arrive to the following result.

Theorem 8.6. *Assume there exist one-way functions, then every language in* NP *has a* constant round *witness-indistinguishable proof system with* negligible error probability*. In fact, the error probability can be made exponentially small.*

8.2.6 Resettable Zero-Knowledge Proofs

The notion of resettable zero-knowledge was introduced in [134] to provide natural solutions to physical problems in the implementation of zero-knowledge protocols, for instance when the prover is implemented on a smart card that can be reset by simply disconnecting its power supply.

Also this stronger property offers security (is closed) under concurrent (parallel) executions of the protocol, something we have seen to be false for the plain zero-knowledge property.

Finally, this notion provides an alternative way of constructing identification schemes that are fundamentally different from the ones constructed following the Fiat-Shamir paradigm, see Sect. 8.3.1.

We will not discuss this property at length, since it is not satisfied by the submission or any of the existing standards. We mention it for completeness and because of its practical relevance.

Loosely speaking, a resettable zero-knowledge proof is an interactive proof where the prover is forced to use a random but *fixed* coin in a polynomial number of executions, each time interacting with the verifier in the usual fashion. The verifier can of course send messages that are related from one execution to another. If the output of the verifier is indistinguishable from a polynomial-time simulator (that depends only on the common input, previously denoted by x), we say the proof is *resettable zero-knowledge* (rZK).

Analogously, one can define *resettable witness-indistinguishability* (rWI). The main results are the following.

Theorem 8.7. *Under the DLP assumption, there is a non-constant round (resp. constant round) rZK (resp. rWI) proof for* NP.

Hence we can modify, under classical assumptions, everything previously written to make it resistant to reset attacks.

8.2.7 Classification of Attacks

In the following, we give a list of possible attacks against identification schemes.

Distinction is sometimes made between adversaries based on the type of information available to them.

Definition 8.16. *An* outsider *is an adversary who has no special knowledge beyond that generally available, e.g. by eavesdropping on protocol messages over open channels.*

An insider *is an adversary with access to additional information, (e.g. commitment of prover, see Sect. 8.3.1). He can be a* one-time insider*, if he obtains such information at one point in time for use at a subsequent time or a* permanent insider *if he has continual access to privileged information.*

We list some active attacks (online or offline) on identification schemes.

Concurrent attack. A concurrent attack is an active attack whereby the attacker interacts with several copies of the same prover (same secret key) using different commitments g in the notation of Sect. 8.3.1. This attack is realistic when identification protocols are used in the context of Internet.

Man-in-the-middle attack. A man-in-the-middle attack is an attack whereby an intruder will communicate anonymously with both the verifier and the prover in such a way as to cut off the direct information exchange between them. As a result the (honest) verifier and prover will think to have successfully executed the identification protocol while in fact they will have leaked privileged information that may prove fatal to the prover.

Replay attack. A replay attack is an attack whereby the attacker is able to achieve his goal by using some previous honest execution of the protocol, either with the same verifier or a different one.

Interleaving attack. An interleaving attack is an attack whereby the attacker uses a selective combination of information from several previous honest protocol executions. A replay attack is a type of interleaving attack.

Reflection attack. A reflection attack is an interleaving attack whereby the attacker passes all queries from the verifier to the originator of the honest information and forwards all replies from this entity back to the verifier.

Forced delay attack. This attack occurs when an attacker intercepts a message and relays it at some later point in time.

Chosen text attack. This is an attack whereby an attacker impersonates the verifier and chooses messages in such a way as to obtain information about the prover's key.

Reset attack. In this type of attack the attacker is able to reset the prover to a previous state, for instance by forcing it to always use the same random seed (coin). This is a very powerful physical attack that can be implemented easily on smart cards deprived of internal power supply. Note that resistance against reset attacks, in a slight generalisation of Sect. 8.2.6, implies resistance to concurrent attacks defined above.

Side-channel attack. An attack by side channel is an attack on an implementation of an algorithm whereby an intruder tries to uncover secret information by a measurement of physical traces, such as the time to perform certain operations, or the electrical, electromagnetic or infrared power trace. This attack is very effective and requires specific countermeasures that will usually slow down performance, for the sake of security. See Annex A for more details.

8.2.8 Assessment Process

The submitted asymmetric identification scheme is assessed with respect to generic common identification schemes and specific attacks. To assess the security of an asymmetric identification scheme which is a zero-knowledge protocol we will follow the Feige-Fiat-Shamir methodology in proving completeness, soundness and the zero-knowledge property.

8.3 Overview of Common Designs

8.3.1 Interactive 3-round Identification Protocols

An identification protocol relying on a zero-knowledge proof of knowledge usually follows the four steps which we will describe. This paradigm is often called the Fiat-Shamir paradigm, since it was first described in the Fiat-Shamir protocol (see Sect. 8.4.1) for knowledge of a square root modulo n.

This kind of protocols is commonly called interactive 3-round identification protocol.

- Commitment: the prover randomly selects a *commitment* g (also called *coin*) and sends it under a mask x to the verifier.
- Challenge: the verifier chooses a random *challenge* c and sends it to the prover.
- Response: the prover uses his secret key together with g to compute a response value y, which is sent back to the verifier.
- Verification: the verifier uses y together with x to check the identity of the prover. He may either accept or reject.

There is also a need when using an asymmetric proof of identity to get assurance of the validity of a public key of the prover. This is the role of the *trusted authority* (TA) which is always supposed honest: the prover chooses a secret key and computes the related public key and the TA certifies his choice. Some of the schemes such as GQ, GPS and Feige-Fiat-Shamir (but not GQ2 or Schnorr) can be made *identity-based*, meaning that the public key is related very closely to the prover's identity (e.g. his email address) and his secret key is then computed by the TA (who holds some super-secret unknown to all provers) who then passes it on to the prover.

Note that it is obvious in all protocols described in Sect. 8.4 are vulnerable to reset attacks, since this is a common weakness to all zero-knowledge proofs of knowledge (by definition, which we do not recall here, see [265, Chap. 6]).

8.3.2 Current standards

- ISO/IEC 9798-5 specifies three identification techniques: one family based on the RSA problem, of which Fiat-Shamir and GQ are instances, one family based on the discrete logarithm problem, of which Schnorr is an example and a mechanism based on an asymmetric encryption scheme, derived from the Brandt-Damgaard-Landrock-Pedersen scheme [120].

8.4 Digital identification schemes considered during Phase II

In this section we describe the identification schemes of Fiat-Shamir, GQ, Schnorr and GPS, with a more thorough evaluation of the last one, which is the only submission to NESSIE.

8.4.1 Fiat-Shamir

We describe the Fiat-Shamir protocol [229], historically the first zero-knowledge protocol and an example emulated by later schemes.

Public parameters. The TA chooses a security parameter n, a large $N = pq$, with p and q primes of size $n/2$ and publishes N.

Private and public keys. Alice chooses her secret key $S \in (\mathbb{Z}/N\mathbb{Z})^*$ and computes the public key $I = S^2 \bmod N$.

One identification round.

1. *Commitment*: Alice chooses a random number $r \in_R (\mathbb{Z}/N\mathbb{Z})^*$, computes $x = r^2 \bmod N$ and sends x to Bob.
2. *Challenge*: Bob chooses a random bit $b \in_R \{0, 1\}$ and sends it to Alice.
3. *Response*: Alice then computes $y = rS^b \bmod N$ and sends it to Bob.
4. *Verification*: Bob checks whether $y^2 = xI^b \bmod N$.

$$\begin{array}{lcl}
\textbf{Prover} & & \textbf{Verifier} \\
\text{choose } r \in (\mathbb{Z}/N\mathbb{Z})^* & & \\
\text{compute } x = r^2 \pmod{N} & \xrightarrow{x} & \\
 & \xleftarrow{b} & \text{choose } b \in \{0,1\} \\
 & & \text{check } y \in (\mathbb{Z}/n\mathbb{Z})^* \\
\text{compute } y = rS^b \pmod{N} & \xrightarrow{y} & \text{accept if } y \equiv xI^b \pmod{N}
\end{array}$$

Fig. 8.22. A round of Fiat-Shamir

The interaction between prover and verifier in one round is reproduced schematically in Fig. 8.22.

Theorem 8.8. *The sequential Fiat-Shamir protocol is an interactive proof of knowledge of a square root of I. It is sound, complete and zero-knowledge.*

As explained after Theorem 8.1, one has to repeat this protocol sequentially k times to obtain a cheating probability of $1/2^k$. Hence it is important that k grows faster than any expression $C \log n$ for any constant $C > 0$ to achieve superpolynomial security, but must remain less than $n^{C'}$ for some $C' > 0$, in order to preserve the zero-knowledge property (see Theorem 8.1).

8.4.2 Schnorr

The prototype of many identification schemes is Schnorr's scheme [560], which we will describe briefly.

Public parameters. A TA publishes

- two primes p, q, such that q divides $p-1$ and such that the discrete log problem is difficult to solve in $\mathbb{F}_p^*$ (typically $\log_2 p = 1024$ and $\log_2 q = 160$),
- an element $g \in \mathbb{F}_p^*$ of order q and
- a security parameter t such that $q > 2^t$.

Private and public keys. Each prover chooses $s \in \{1, \ldots, q-1\}$ at random and computes $I = g^{-s} \pmod{p}$, publishing I through the TA as their public key.

One identification round.

1. *Commitment*: the prover picks $r \in \{0, \ldots, q-1\}$ at random and sends to the verifier $x = g^r \pmod{p}$.
2. *Challenge*: the verifier chooses a random $c \in \{0, \ldots, 2^t - 1\}$ and sends it to the prover.
3. *Response*: the prover computes $y = r + sc \pmod{q}$ and sends it to the verifier.
4. *Verification*: the verifier computes $z = g^y I^c \pmod{p}$ and accepts the prover if and only if $z = x$.

The interaction between prover and verifier in one round is reproduced schematically in Fig. 8.23.

Prover		**Verifier**
choose $r \in \{0, \dots, q-1\}$ compute $x = g^r \pmod{p}$	$\xrightarrow{x}$	
	$\xleftarrow{c}$	choose $c \in \{0, \dots, 2^t - 1\}$
compute $y = r + cs \bmod q$	$\xrightarrow{y}$	accept if $g^y I^c = x \pmod{p}$

Fig. 8.23. A round of Schnorr

Theorem 8.9. *The Schnorr protocol is an interactive proof of knowledge of the discrete logarithm of I to the base g in $\mathbb{Z}/p\mathbb{Z}$. It is sound, complete and zero-knowledge for fixed t.*

Note that trivially if Charlie tries to impersonate Alice, his probability of fooling Bob is at most $1/2^t$. However, if one lets t grow super-polynomially, then the protocol cannot be simulated in polynomial time and thus cannot be zero-knowledge. Hence to decrease the error probability while keeping adequate security against active attacks, one must play this protocol several times sequentially.

8.4.3 GPS

GPS is the only identification scheme submitted to NESSIE. The scheme has good performance with high security. The submitted documents contained some minor flaws in the specifications, but these were corrected at the beginning of phase II.

8.4.3.1 The Design

The GPS scheme consists of an interactive zero-knowledge identification scheme, that combines provable security based on the problem of integer factorisation and the short exponent problem, with a small identity-based key and minimal on-line computations.

It is essentially a modified version of the well-known Schnorr identification scheme (see Sect. 8.4.2). Unlike the Schnorr scheme, GPS uses a generator g with unknown order and the exponent is calculated in $\mathbb{Z}$ rather than modulo p.

Public parameters. The GPS identification scheme consists of ℓ iterations of an identification round. This ℓ is part of the security parameters of the scheme. Let A, B, S be parameters with $|A| \geq |S| + |B| + 80$, $|B| = 32$ and $|S|$ greater than 140 bits. The submitters propose that $|S| = 180$ meaning A is approximately a 300-bit number. Let n be a RSA modulus ($n = pq$ where p, q are for instance 768-bit primes).

Private key. The private (prover's) key is a random $s \in [1, S]$.

Public key. The public key is $I = g^{-s} \bmod n$.

We now describe one round of the GPS identification scheme.

1. *Commitment*: the prover picks $r \in \{0, \ldots, A-1\}$ at random and sends to the verifier $x = g^r \pmod n$.
2. *Challenge*: the verifier chooses a random $c \in \{0, \ldots, B-1\}$ and sends it to the prover.
3. *Response*: the prover checks that $c \in \{0, \ldots, B-1\}$, computes $y = r + sc$ and sends it to the verifier.
4. *Verification*: the verifier checks that $y \in \{0, \ldots, A + (B-1)(S-1) - 1\}$, computes $z = g^y I^c \pmod n$ and accepts the prover if and only if $z = x$.

The interaction between prover and verifier in one round is reproduced schematically in Fig. 8.24.

Prover		**Verifier**
choose $r \in \{0, \ldots, A-1\}$ compute $x = g^r \pmod n$	$\xrightarrow{x}$	
	$\xleftarrow{c}$	choose $c \in \{0, \ldots, B-1\}$
check $c \in \{0, \ldots, B-1\}$ compute $y = r + cs$	$\xrightarrow{y}$	check $y \in \{0, \ldots, A + (B-1)(S-1) - 1\}$ accept if $g^y I^c = x \pmod n$

Fig. 8.24. A round of GPS

This scheme is proved complete, sound (a prover accepted after ℓ rounds with probability greater than $1/B^\ell$ must know the discrete logarithm of I), and perfectly zero-knowledge and honest-verifier zero-knowledge if B is not too large, see Sect. 8.4.3.2.

GPS is designed to be used in situations where authentication has to be done "on the fly" with smart cards.

8.4.3.2 The Security of GPS

The submitters show that GPS is complete, sound and zero-knowledge. More precisely they prove the following three assertions [526].

Theorem 8.10 (Completeness). *The execution of the protocol between a prover who knows the secret key corresponding to his public key and a verifier is always successful.*

Theorem 8.11 (Soundness). *Assume some adversary is accepted in polynomial time with non-negligible probability by honest verifiers, that* $\log(|n|) = o(\ell \cdot |B|)$ *and that* ℓ *and* B *are polynomial in* $|n|$*. Then there exists a polynomial-time algorithm that solves the discrete log with short exponent problem.*

Theorem 8.12 (Zero-knowledge). *The GPS protocol is computationally zero-knowledge if* ℓ *and* B *are polynomial in* $|n|$ *and* $\ell SB/A$ *is negligible. As a consequence it is also honest-verifier computationally zero-knowledge under the same assumptions.*

In [527], the authors actually relate the soundness of the protocol to the factorisation of the modulus n, under the same hypothesis.

It may be remarked that the proof of the zero-knowledge property assumes that $|B|$ is constant, as in the proof for the original Schnorr protocol. On the other hand, Pointcheval [522] proves that under some hypothesis on n and g, assuming $|B| > 2\operatorname{ord}(g)$, GPS still retains its witness indistinguishable property, and active attacks are then related to the factorisation of n. This leads to a more efficient and secure identification protocol since one can then safely take $\ell = 1$.

8.4.3.3 Various Attacks on GPS [572]

Verifier Cheating over Value of c. In the original version of GPS the response did not include any check that $c < B$. If the verifier is dishonest then he could send $c = A$ as a challenge to the prover. The prover then computes $y = r + sA$ and sends this value to the verifier. Since (assuming the prover is following the protocol correctly) $r < A$, the verifier can easily compute

$$s = \left\lfloor \frac{y}{A} \right\rfloor \ .$$

This problem was pointed out by Daniel Bleichenbacher (see NESSIE forum) and the protocol was consequently modified.

Breaking the Pseudo-Random Number Generator. Suppose that the pseudo-random number generator used to generate r in the commitment step is weak, so that given k random values $r_1, r_2, \ldots, r_k$ we can predict r_{k+1}. Then the verifier can break the system by sending $c = 0$ k times in order to find the values $r_1, r_2, \ldots, r_k$ and thus predict the value of r_{k+1}. On the $(k+1)$-th application of the algorithm, the verifier can send an arbitrary value of c and then compute the secret key s which is given by

$$s = \frac{y - r_{k+1}}{c} \ .$$

Alternatively, suppose that the prover is dishonest and wishes to pass himself off as the holder of the private/public key pair (s, I) without actually knowing the private key s. If such a dishonest prover could break the pseudo-random number generator used to generate the challenge c then he could fool the verifier as follows:

– the dishonest prover sends the commitment $x = g^{r+c}I^c$ to the verifier. The verifier sends the (not so) random challenge c to the prover.
– The dishonest prover sends $y = r + c$ to the verifier. The verifier checks that $x = g^y I^c = g^{r+c} I^c$ and that $y \in [0, A + (B-1)(S-1)]$.

Both of these checks will be accepted as correct by the verifier who will then assume that the dishonest prover is in fact the holder of the private key s.

Therefore GPS is only secure when the used pseudo-random number generator is unbroken.

8.4.3.4 Fault and Chosen-Modulus Attacks on GPS [207, 208]

The first fault attack targets the secret key and requires the following fault model (see also Annex A): bit-flip fault model, complete control on the number of faulty bits induced and complete or loose control on the fault location. The fault can be induced before the computation starts. The idea is to flip exactly one bit of the secret key used by the prover. With his response to the challenge, an attacker can recover the original value of the bit he flipped. Repeating this operation, he can recover all the bits of the secret key. This attack has been introduced by [104] and is fully described in [208]. If the attacker does not know exactly which bit of the secret key he flipped, he can make several tries and find out which one it was and its original value. Obviously the attack is then less efficient.

The second fault attack targets an intermediate value and works in the following model: bit-flip fault model, complete control on the number of faulty bits induced, and complete or loose control on the fault location. The value targeted is namely the random value r chosen by the prover at the first step. The prover has to store this value as he is going to use it again when responding to the verifier's challenge. If an attacker is able to swap exactly one bit of this value while the prover is waiting for the challenge, and to repeat this operation several times sending the same challenge to the prover, he can recover the secret key s. The attacker must have at least the following control on the fault location: the fault concerns the value r. If he has complete control, so that he knows exactly which bit has been flipped, the attack is more efficient. This attack has been first described against Schnorr's identification scheme in [104], and the version against GPS is described in [208].

GPS is also vulnerable to a chosen-modulus attack. Indeed, the modulus n is needed by the prover, so a modification of this value can lead to an attack. An attacker who is able to replace this value by a value of his choice can recover the secret key. The first step of the protocol consists in the prover choosing a commitment r and sending the value $g^r \bmod n$ to the verifier. If the attacker has replaced the original modulus by a number so that he can easily solve the discrete logarithm in the new field, he can thus find the value r and so the secret key.

8.4.4 GQ

In this section we sum up what the research community already knows about the security of the GQ protocol. We will see that GQ satisfies the following properties: completeness, soundness, perfect zero-knowledge and security against concurrent attacks under the RSA-omi assumption. We will then discuss more practical aspects of the security of GQ, namely we see which attacks on RSA apply to GQ, then we notice that GQ is immune from side-channel attacks on exponents.

8.4.4.1 The Design

The GQ protocol was first published in [278]. It is of the same design as the Fiat-Shamir identification scheme. It provides security based on the RSA assumption, identity-based public keys, and allows the use of the same modulus by multiple users. It needs a trusted authority to generate some of the parameters.

Public Parameters. Let n be a RSA modulus, and v a *prime* RSA exponent. These parameters can be shared by several users.

Parameters of the authority. The factorisation of n and the integer s such that $v.s \equiv 1 \pmod{\varphi(n)}$ are kept secret by the authority.

Parameters of the users. Each user's private key is an integer Q, and the corresponding public key is an integer G verifying $GQ^v \equiv 1 \pmod n$ (the equation $G \equiv Q^v \pmod n$ can also be used).

Remark. *The use of the second equation only affects the verification step.*

A round of GQ. We describe one round of the GQ identification algorithm.

1. *Commitment*: Alice chooses a random number $r \in_R (\mathbb{Z}/N\mathbb{Z})^*$, computes $R = r^v \bmod n$ and sends R to Bob.
2. *Challenge*: Bob chooses a random $d \in \{0, \ldots, v-1\}$ and sends it to Alice.
3. *Response*: Alice checks that $d \in \{0, \ldots, v-1\}$, then computes $D = rQ^d$ mod n and sends it to Bob.
4. *Verification*: Bob checks that $D \in (\mathbb{Z}/n\mathbb{Z})^*$ and whether $R \equiv D^v G^d$ mod n (or $RG^d \equiv D^v$ mod n if second equation is used).

The interaction between prover and verifier in one round is reproduced schematically in Fig. 8.25.

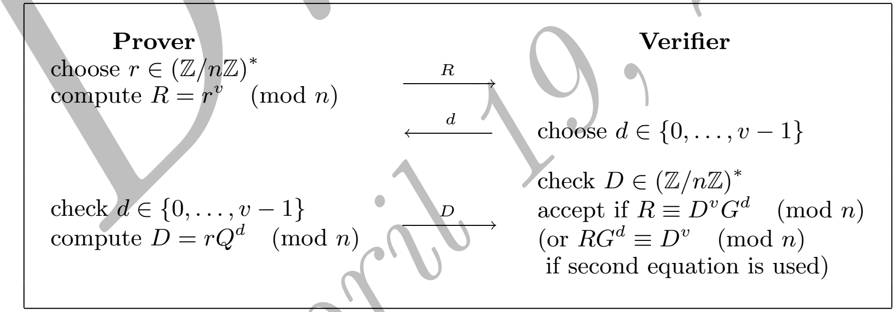

Prover
Verifier
choose $r \in (\mathbb{Z}/n\mathbb{Z})^*$
compute $R = r^v \pmod n$
$\xrightarrow{R}$
$\xleftarrow{d}$
choose $d \in \{0, \ldots, v-1\}$
check $d \in \{0, \ldots, v-1\}$
compute $D = rQ^d \pmod n$
$\xrightarrow{D}$
check $D \in (\mathbb{Z}/n\mathbb{Z})^*$
accept if $R \equiv D^v G^d \pmod n$
(or $RG^d \equiv D^v \pmod n$
if second equation is used)

Fig. 8.25. A round of GQ

8.4.4.2 The Security of GQ

The following properties of GQ have been proved.

Theorem 8.13 (Completeness). *The execution of the protocol between a prover who knows the secret key corresponding to his public key and a verifier is always successful (it is obvious, see [278]).*

Theorem 8.14 (Soundness). *Assume some adversary knows a commitment such that he can succeed in an identification with probability greater than v^{-1}, then this cheater can recover the real prover's private key in polynomial time. This proof was first published in [277].*

Theorem 8.15 (Zero-knowledge). *The GQ protocol is perfect zero-knowledge if v is polynomial in n (this was shown in [130]). As a consequence it is also honest-verifier computationally zero-knowledge under the same assumption.*

Theorem 8.16. (Security against Impersonation under Concurrent Attacks). *The GQ protocol is secure against impersonation under concurrent and active attacks, if the RSA-omi problem is hard. This result was published in [51].*

8.4.4.3 Practical security problems

RSA-related security problems. The GQ protocol requires the use of a RSA modulus n, and the private GQ key (or its inverse modulo n, depending upon which version is being used) is the RSA signature of the public GQ key. This implies that the choice of the modulus n and the private exponent s must be done with the same care as in RSA.

– The modulus n must be large enough to prevent elliptic curve factorisation
– One can wonder if Wiener's attack [622], Boneh-Durfee's attack [106], or Blömer-May's attack [98] apply to GQ. It could in theory, but it will not work in practice. These attacks work on small secret exponents (i.e., $s < n^{\frac{1}{4}}$ for Wiener's attack, $s < n^{0.292}$ for Boneh-Durfee's attack and $s < n^{0.29}$ for Blömer-May's attack). But the secret exponent cannot be small in GQ because $v.s > \varphi(n)$ and v is small. For example, if we take $|n| = 1024$ and $|v| = 16$, we have $|s| > 1000$.
– Since the factorisation of n is not known by the prover, he can't use the CRT function, and the fault attack on chinese remainder based implementations [104] will not work on GQ.

A remark about side-channel attacks. An attacker might try to find some information on the private key by using a side-channel attack to recover one or both of the exponents used by the prover. In this paragraph, we notice that even if the attacker recovers both of the exponents it doesn't give him any advantage.

Let us give at look at the computations made by the prover during a proof (notice that the first computation can be made in advance).

– *Commitment :* The prover calculates $R = r^v \bmod n$. The exponent used here, v, is public.
– *Answer :* The prover calculates $D = rQ^d \bmod n$. The exponent used here, d, is the challenge and is not secret (more precisely, an eavesdropper can see it during the proof).

Recovering one of the exponents, or both of them, will not provide an attacker any additional information about the private key.

A. Side-channel attacks

Ever since cryptography became part of our everyday life, not only the cryptographic algorithms have been subject to attacks, but also their implementations in hard- or software. The traditional cryptographic model, however, does not take the aspect of implementation attacks into account. In the traditional scenario, Alice and Bob secure their communication by some mathematical function, namely the cryptographic algorithm. The adversary, Eve, is only assumed to have knowledge about this mathematical function and some plain- and ciphertext pairs. Consequently, security proofs only involve these components. Despite the given proofs of security for any cryptographic algorithm in any theoretical model, a communications system based on this algorithm can still be vulnerable to quite a number of other attacks. A very dangerous class of such attacks is commonly referred to as side-channel attacks. This appendix is devoted to the consideration of the NESSIE primitives in terms of their vulnerabilities to side-channel attacks and their ability to resist such attacks.

Currently there is very little theoretical framework in which one can assess such attacks. Our research has concentrated on the properties of an algorithm which allow it to be protected against such attacks. The results of this research have been collected in several survey articles. In Sect. A.1 we deal with the passive types of implementation attacks, which are known as side-channel or information-leakage attacks. Such attacks do not require the adversary to actively manipulate the computation, but only to monitor the side-channel leakage during the computation. In Sect. A.2 we deal with active attacks. As can be deduced from their name, this type of implementation attacks assume an attacker actively manipulating the execution of a cryptographic algorithm. Information which is specifically related to the NESSIE primitives can be found directly in the sections devoted to the individual primitives, in the main part of this document.

A.1 Passive Attacks

Passive attacks were published by Kocher in [374] for the first time. In this article, timing information was used to gain knowledge about the secret key from implementations of the RSA, DSS, and other cryptosystems. The attack described in this article required an attacker to be able to simulate or predict the timing

[0] Coordinator for this appendix: KUL — Elisabeth Oswald

behaviour of the attacked device rather accurately. The second article by Kocher *et al.* [375] presented a similar, but far more dangerous attack. This second article introduced the usage of power consumption information to determine the secret key. Because of its statistical nature, one of the attacks in [375] is called *differential* power analysis. Another type of side-channel information was introduced only recently. The first articles, [535] and [251], about the usage of electromagnetic emanations were presented in 2000 and published in 2001.

A.1.1 Types of Information Leakage

We briefly discuss the different types of information leakage that have been exploited by attacks published in the open literature so far.

Execution Time Leakage. Often, a device takes slightly different amounts of time to execute an algorithm. Explanations for this behaviour include differing input data which might cause some instructions to take different amounts of time for their executions, performance optimisations and branching instructions. Practical implementations of attacks using this kind of information leakage, such as [198] and [295], indicate that such attacks are difficult to realise in practice owing to the difficulty of measuring the real execution time. In many modern processors, even on smart cards, instructions can be cached and so the execution time is more and more related to other influences.

Countermeasures appear to be easy to implement, and to work efficiently in practice. Since their first introduction, most work in the area of side-channel attacks has been dedicated to the exploitation of side-channels with a higher amount of information.

Power Consumption Leakage. Most commonly used cryptographic devices are implemented in CMOS (complimentary metal-oxide semiconductor) logic. The power consumption characteristics of CMOS circuits can be summarised as follows. Whenever a circuit is clocked, the circuit gates change their states simultaneously. This leads to a charging and discharging of the internal capacitors, and this in turn results in a current flow which is measurable at the outside of the device.The measurements can be acquired easily using either a data acquisition card or a digital oscilloscope. The current flow can be measured directly with a current probe, or by putting a small resistor in series with the device's ground-input or power-input. Power analysis attacks are the most popular attacks at the time of writing owing to their effectiveness and simplicity. While the first publication [375] was mainly concerned with power-analysis attacks on secret-key cryptosystems, Messerges *et al.* [443] presented such an attack for public-key cryptosystems. In [56] and [11], the methods of some power-analysis attacks are refined. An introduction to the practical aspects of power-analysis attacks can be found in [10].

Electromagnetic Radiation Leakage. The same charging and discharging which occurs whenever a circuit is clocked creates, besides the current flow, also a certain electromagnetic (short EM) field. *Direct emanations* are caused by intentional current flow which is caused by the execution of an algorithm. *Unintentional emanations* are caused by the miniaturisation and complexity of modern

CMOS devices. This miniaturisation and complexity results in coupling effects between components in close proximity. EM attacks are becoming more and more popular at the time of writing because of the high amount of information this side-channel can leak and because the information can be exploited at a large distance from the attacked device [9].

Error Message Leakage. An error message attack usually targets a device implementing a decryption scheme. In the standard model, there is a feedback from the device to tell whether or not the message has been successfully decrypted. If the attacker can somehow know the reason why the decryption operation failed, he might gain some secret information by sending well chosen ciphertexts to the device, or by observing others do the same. Attacks exploiting this side-channel are summarised in Sect. A.1.4.

Combining Side-Channels. Not much research has been done on this topic so far. However, the following simple observation has been used for attacks. Timing attacks can suffer from the difficulty of obtaining precise measurements. Attacks are even more difficult when only one intermediate operation is targeted. In such a case, power measurements lead directly and more precisely to timing information about the intermediate operation if this intermediate operation is visible in the power consumption trace. Besides in [556], such an attack is also presented in [617].

A.1.2 Simple Side-Channel Attacks

Most attacks presented so far have been performed with power consumption leakage information. A *trace* refers to a measurement taken for one execution of the attacked cryptographic operation. In a simple side-channel attack, only one measurement is used to gain information about the device's secret key. Obviously, for such an attack to work, the side-channel information needs to be strong enough to be directly visible. Additionally, the secret key or hidden message needs to have some simple, exploitable relationship with the operations visible in the side-channel trace. Such an attack typically targets implementations which use key dependent branching.

A.1.2.1 Attacking Implementations of Symmetric Schemes

A special class of simple power-analysis attacks, the so called *Hamming weight attacks*, exploit a strong relationship between the Hamming weight of the secret key and the power-consumption trace. In [80] such an attack is presented on an implementation of the DES algorithm and in [413] one is presented on an implementation of the AES algorithm. For this type of attack it is vital that the implementation is based on relatively small data-words, as happens for example in an 8-bit implementation. Usually, this type of attack is applied to implementations of ciphers with a simple key schedule. For implementations that try to achieve a protection against first-order differential power-analysis attacks, this method can be used to determine information on the mask used. *Collision Attacks* on implementations of the DES algorithm have been examined in [562]. Internal collisions are detected by their power trace in these attacks.

A.1.2.2 Securing Implementations of Symmetric Schemes

To counteract the type of simple power-analysis attack that uses Hamming weight information, a designer has to assure that the Hamming weight information which is leaked is not correlated with the intermediate values that are processed. In dedicated hardware implementations this can be achieved by using a special logic-style or by masking intermediate values (this can be achieved by bus encryption as well as by masking the operations of the algorithm in general). In software implementations the intermediate values have to be masked. It is imperative to implement a decent masking scheme to counteract attacks such as presented in [142] and [153]. A good noise generator on chip will also help to counteract such attacks, at least in the case of a power attack.

A.1.2.3 Attacking Implementations of Asymmetric Schemes

Scenarios in which simple side-channel attacks are a possible threat have been considered in [508] and can be summarised as follows. If a *multiplication of a known and a secret value* has to be calculated, then a simple side-channel attack is theoretically possible, but unlikely to work in practice. An *exponentiation of a known with a secret value* is also in principle vulnerable to simple side-channel attacks. The practical feasibility of the attack is heavily dependent on the implementation. Unprotected *Scalar multiplications* of a known elliptic curve point by an unknown scalar are highly vulnerable to this kind of attack, regardless of the underlying hardware. Also, implementations based on addition-subtraction chains can leak enough information to recover the private key [507]. Considerations of the security of implementations of S-Flash and Quartz can be found in [12]. In Klima *et al.* [349] a variant of a Hamming weight attack is applied to RSA.

A.1.2.4 Securing Implementations of Asymmetric Schemes

Attacks against a multiplication can be counteracted by switching multiplier and multiplicand. To protect implementations of modular exponentiations, an always-square-and-multiply approach can be helpful. The same is valid for implementations of the scalar point-multiplication on elliptic curves. In general, there are more implementation options to secure elliptic curve cryptosystems. An overview of countermeasures published in the open literature is given in [508].

A.1.2.5 Conclusions and Recommendations

Hamming weight attacks are a practical threat to all (unprotected) software implementations of symmetric algorithms. Countermeasures in hard- and software have been published in the open literature (see [509] for a survey). Since the efficient implementation of countermeasures is most important, we recommend choosing algorithms which allow such efficient implementations. Summarising our considerations in [509], we can say that ciphers without too many different algebraic structures are easier to protect. Rijndael, Khazad and Camellia are algorithms which are favourable from our point of view.

Simple side-channel attacks can be applied in practice on (unprotected) software implementations of a modular exponentiation, and on potentially all kinds of (unprotected) implementations of elliptic-curve scalar point-multiplications. Such attacks are less likely to be realisable in practice on implementations of modular exponentiations than on implementations of scalar point-multiplications. But there are many more ways to counteract such attacks in implementations of scalar point-multiplications.

A.1.3 Differential Side-Channel Attacks

Differential side-channel attacks exploit the correlation between the processed data and the instantaneous side-channel leakage of the attacked cryptographic device. Because this correlation is usually very small, statistical methods must be used to exploit it efficiently. In a differential side-channel attack the output(s) of the real physical device and the output of a hypothetical model of the device (working with a hypothetical key) are compared. Only if the hypothetical key equals the real key is the output of the hypothetical model correlated with the output of the real device. By comparing the two outputs, the attacker can determine the secret key. If the hypothetical model only outputs a single value (i.e. it predicts, for example, the power consumption of the real device for only one moment in time), then the attack is called a *first-order* differential side-channel attack. If a model can output more values for the same side-channel then the attack is called a *higher-order* differential side-channel attack. For example, if two output-values are used in an attack then the attack is a second-order differential side-channel attack. The term “differential side-channel attack” used on its own generally refers to a first-order differential side-channel attack.

A.1.3.1 Attacking Implementations of Symmetric Schemes

The strength of an attack depends largely on the quality of the hypothetical model used by the attacker. Dedicated hardware implementations of Feistel ciphers without an initial bit-wise addition of the key allow the implementation of a very powerful hypothetical model. The statistical qualities of the S-boxes also influence the strength of an attack. However, none of the block ciphers which we considered in [509] showed any special properties in this regard.

A.1.3.2 Securing Implementations of Symmetric Schemes

As we pointed out in the previous sections, software countermeasures are usually based on masking the data and the key during a computation. Ciphers which allow the cheap implementation of masking schemes are certainly preferable. For hardware countermeasures, their cheap realisation is also a priority. If a special logic style is used then it is certainly an advantage if only a few different types of gates have to be designed in this logic style. The simpler a cipher’s description, the more suited it is for such an implementation.

A.1.3.3 Attacking Implementations of Asymmetric Schemes

Typical targets for an attack are again implementations of the modular exponentiation and implementations of scalar point-multiplication on an elliptic curve. Three different types of differential side-channel attacks have been introduced. Two of them, the SEMD (Single-Exponent Multiple-Data) and the MESD

(Multiple-Exponent Single-Data) attacks, do not require the attacker to have knowledge of or a model for the attacked device. The ZEMD (Zero-Exponent Multiple-Data) attack, which is essentially the same attack as proposed in [150], assumes that the attacker has a model and can predict intermediate values of the computation. Several refinements of the basic ideas behind differential side-channel attacks have been presented. An overview can be found in [508].

A.1.3.4 Securing Implementations of Asymmetric Schemes

There have been significantly fewer published articles dealing with countermeasures for implementations of the modular exponentiation than with countermeasures for implementations of the scalar point-multiplication on an elliptic curve (see [508] for a survey). Countermeasures for the scalar point-multiplication include randomising points, randomising curves, randomising the scalar and randomising the algorithms for the scalar point-multiplication. Since practical realisations of elliptic curve cryptosystems are software implementations (which probably make use of some accelerator unit anyway), most of these countermeasures are cheap to implement and to combine with one another.

A.1.3.5 Conclusions and Recommendations

Summarising our considerations in [509], we can say that ciphers without too many different algebraic operations are easier to protect. AES, Khazad and Camellia are algorithms which are favourable from our point of view. We did not consider any hash functions, stream ciphers or MAC algorithms in detail. However, the attack techniques and countermeasures would be exactly the same as for block ciphers.

Because of the variety of available countermeasures for elliptic curve cryptosystems, they seem to be favourable in the case of asymmetric schemes.

Hardware countermeasures suffer from the same drawbacks as we described in Sect. A.1.2.5. Another difficulty in the case of asymmetric schemes is the inherent complexity of their implementation. Dedicated hardware implementations of asymmetric schemes are significantly larger than such implementations of symmetric schemes. This amplifies the difficulties in the application of countermeasures.

A.1.4 Error Message Attacks

This kind of attack was first introduced by Bleichenbacher in [97], which describes a chosen ciphertext attack against the RSA encryption standard PKCS#1. In this standard, the decryption operation fails if the result of the RSA decryption is not in the correct format (more precisely, the first two bytes are fixed). The attack demonstrated that it is then possible to compute the RSA decryption of any ciphertext, by sending well chosen "ciphertexts" to the device and using it as an oracle to know if the corresponding plaintext is in the right format. There may be other integrity checks applied, in addition to the format checking, but the attack is still reliable if the different failures can be separated. This is the case, for instance, if different error messages are sent, or if the whole verification process takes more time to achieve than the first failure condition (and in that case, the attack will be combined with a side-channel technique, see Sect. A.1).

Other error message attacks are mentioned in the literature: [414] against RSA-OAEP, [192] against the NESSIE candidate EPOC-2 (this one recovers the secret key) and [614] against the CBC mode in various standards. This shows that no information about the reasons why a decryption failed should leak from the device.

A.1.5 Consideration of Hash-function, MACs and Stream Ciphers

There has been no research on attacks against MAC primitives and hash functions. Only in the case of the stream cipher SOBER is a timing attack known [390]. The attack-techniques, however, are essentially the same as the techniques which were developed for block ciphers. Scenarios in which hash functions or MACs would be subject to attacks include constructions in which these primitives are used keyed.

A.2 Active Attacks

In a passive attack, the attacker only eavesdrops on some side-channel information, which is analysed afterwards to reveal some secret information. An active attack involves an attacker that takes *active* part in the attack: we make the assumption that the attacker is able to somehow deviate the device from its normal behaviour, and tries to gain additional information by analysing its reactions. Some passive techniques seen in the previous section can be used to determine these reactions. This can be done, for instance, by modifying some internal data used by the device.

In the following we describe the most popular type of active attack: fault attacks. We will give examples of how successfully they have been applied, and see how they can be avoided.

A.2.1 Fault Attacks

When an attacker has physical access to a cryptographic device, he may try to force it to malfunction. A fault attack is an attack in which information about the message or the secret key is leaked from the output of erroneous computations. This kind of attack can be applied to both symmetric and asymmetric cryptosystems, and was first introduced in [104].

There are several ways to introduce an error during the computation performed by the cryptographic device. Though the description of these practical means is beyond the scope of this introduction, we cite some non-invasive methods:

- spike attacks work by deviating the external power supply more than can be tolerated by the device. This will surely lead to a wrong computation.

– glitch attacks are similar to spike attacks, but target the clock contact of the integrated circuit.
– optical attacks work by focusing flash-light on the device in order to set or reset bits. It seems that very precise faults can be induced with this technique, as shown by Anderson *et al.* in [16].

We will now focus on what the attacker is able to do (i.e. the attack model) instead of considering the practical means by which he does it. We will first sort these attacks according to two criteria. The first one is the attack model: how can the attacker modify the value? Which are exactly the assumptions about his capabilities? The second one concerns the value targeted: which data used by the device does the attacker modify?

A.2.1.1 Attack models

There are a lot of different fault attacks. Most of the time, they differ by the assumptions made about the attacker's capabilities: the way he can access and modify the memory, the power he has upon the fault occurrence time, etc. In Blömer *et al.* [99], the authors characterise fault attacks according to different criteria:

– control on the fault location;
– control on the fault occurrence time;
– control on the number of faulty bits induced;
– the fault model.

On the three first items, an attacker can have either no control, loose control or precise control. We have to clarify the fault models we will consider. Of the models proposed in [99], we selected the following ones: the random fault model, the bit flip model, and the bit set or reset model.

The authors assume that the bit flip and bit (re)set models can be achieved with complete control on fault location and precise timing using optical attacks. In that case, an attacker can mount what we will call a *chosen value/modulus attack*: he can replace a value used by the device by a value of his choice. The attacker may or may not know the original value. This kind of attack is described in Sect. A.2.2.

A.2.1.2 Target values

We now differentiate the different values that can be the target of a fault attack.

In asymmetric cryptography, one party owns some secret and the other party only knows public values. Let Bob be the one who possesses the secret (the decrypter, the signer or the prover, depending on the type of primitive considered), and Alice the one who knows only the public values. First we should separate into two parts the set of data that can be made public. The *parameters* are the set of public data which is initially chosen and which defines the general setting (e.g., the characteristics of the elliptic curve in elliptic curve cryptography). The *public key* is then chosen among all the public keys that the parameters permit. This public key can be modified without changing the parameters. The *private* *key* is a part of the whole set of data needed by Bob to perform his computation, the one that changes when we change the public key.

So, the public data that is not modified when the public key changes are part of the parameters. We call the subset of the private key that has to be kept secret the *secret key.* Note that the private key depends on implementation choices. For instance, if a public modulus $n = pq$ is needed, one can choose to store n, or to store the values p and q and to compute n. The secret key may also depend on some choices.

In symmetric cryptography, the private key is reduced to the secret key and if public data is needed then it is regarded as parameters (e.g., constant words, S-boxes).

So, in order to perform his part of the computation, Bob needs the private key (which may contain a part of the public key) and maybe some parameters. Thus a modification of any part of this set of data can possibly lead to an attack giving some information. This is important to notice because the data which does not need to be kept secret might be stored in an unprotected memory location, so that it is easy to modify it. If a modification of some public data permits an attack giving information about the secret data, this data should be protected as well as the secret data, and some additional countermeasures might be useful.

Another kind of data that can be the target of a fault attack is intermediate data. An attacker could introduce faults in the registers of the device while they are holding some intermediate values.

A.2.1.3 Published attacks

Introducing faults in the secret key of asymmetric schemes. This kind of attack has been applied by Bao *et al.* [33] to the RSA decryption (or signature) scheme, and to the El Gamal, Schnorr and DSA signature schemes. It has been extended to various RSA-type signature schemes in [34], to the encryption scheme RSA-KEM in Klíma *et al.* [349], and to ACE-KEM, ECIES-KEM and PSEC-KEM in [197]. It works efficiently in the following model: bit flip fault model, complete control on the number of faulty bits induced, complete control on the location. For the timing, the only requirement is that the fault occurs before the critical computation, so we need only loose control on it.

The idea is to flip one bit of the secret key and to use the erroneous computation of the device to get the value of this bit. This is particularly easy for discrete logarithm based schemes because we can use the simple relation between the secret and the public keys to successively guess the bits of the secret key.

Introducing faults in registers. Here, faults are introduced in an intermediate value stored in the registers of the device. This kind of attack has been applied to the Fiat-Shamir and the Schnorr identification schemes in Boneh *et al.* [104], where the random value r chosen by the prover is the target of the fault introduction. This attack works against GPS, as shown in [208]. The prover has to store this value r, as he is going to use it again when responding to the verifier's challenge. The idea is to swap exactly one bit of r while the prover is waiting for the challenge. Several such erroneous computations are used to recover the secret key.

Random faults. This is the most powerful attack, as the fault model and the number of faulty bits induced are random, and controls on the location and the timing are loose. The well known Bellcore attack [104] against RSA using the Chinese Remainder Theorem belongs to this category. This attack has been improved by Joye *et al.* [330]. The NESSIE candidate ESIGN-D (see Sect. 7.4.2) is also vulnerable to an attack of this type [207]. Here, a random error occurs during some (more or less time consuming) step of the computation, and the erroneous output completely reveals the secret key.

Fault attacks against elliptic curve cryptosystems. The use of elliptic curves can lead to specific fault attacks. A fault attack of this type is described in Biehl *et al.* [58]. In this paper, the idea is to somehow modify the point involved in the scalar multiplication step in such a way that the resulting point is on a cryptographically weak curve, where we can solve the discrete logarithm problem, and thus find out the secret key. The authors examine different attacks models based on this idea.

Fault attacks against symmetric cryptosystems. Fault attacks as introduced in [104] use algebraic properties of the asymmetric cryptosystems. In Biham *et al.* [79], the authors apply fault attacks to symmetric cryptosystems, introducing *differential fault analysis*, which uses statistical methods. Various fault models are considered, and several cryptosystems are attacked, among them the full DES. Fault attacks against the AES are considered by Blömer *et al.* in [99].

A.2.2 Chosen Modulus Attacks

Chosen modulus attacks can be viewed as a particular kind of fault attack. In a chosen modulus attack, the attacker replaces a value used by the device to perform its cryptographic computation. This can be done, for instance, by applying several bit sets/resets at a precise memory location of the device in order to replace a (possibly unknown) value by another value, this one known and chosen. The target value is likely to be a public one (either a part of the public key, or some parameter of the scheme), which may not be as well protected as a secret value.

The schemes vulnerable to this kind of attack are typically the ones where a modular exponentiation with secret exponent is performed. Usually, the (public) modulus will be replaced by a value well chosen by the attacker, enabling him to recover the secret exponent. This is done against RSA-KEM in [349]. The authors explain how to recover the secret exponent using the decryption device with a Trojan modulus. A chosen modulus attack against GPS is described in [207].

A.2.3 Other attacks

The problem of public keys validation is discussed in Brown *et al.* [23]. The authors demonstrate that attacks can be mounted against some elliptic curve based schemes if the receiver of an elliptic curve point does not check that the point lies on the right curve. This shows the importance of validating values received from the outside.

A.2.4 Preventing fault attacks

As we have seen, fault attacks are very powerful attacks that may permit the cryptanalysis of theoretically secure schemes. Several software countermeasures have been proposed, among them:

- Double computation: for encryption schemes, this could be a solution. However, it doubles the computational time, and does not protect against permanent faults.
- Checking the output: this can be done quite efficiently with signature and identification schemes. However, it assumes that the device contains the whole public key, and this is not always the case.
- Randomisation: here random bits are introduced in the computation. They are either XOR-ed to sensitive data to blind them, or appended to the message, as in the signature scheme RSA-PSS.

In [333], Joye *et al.* show that some countermeasures can sometimes help the attacker. However, we feel confident that hardware countermeasures used in combination with software countermeasures can prevent the large range of existing fault attacks.

Part C

Performance evaluation

Book II Part C of this final report was first published under the name ”Performance of Optimized Implementations of the NESSIE Primitives” as Deliverable D21 of NESSIE.

Table of Contents

List of Tables

List of Figures

1. Introduction

1.1 Overview

This Part of the book provides a detailed review of all performance procedures and results undertaken up to January 2003. It provides new and improved speed estimates for the candidates submitted to NESSIE (along with standard primitives and other primitives we implemented), with emphasis on the primitives accepted for the second phase of NESSIE. These estimates are the results of a long and thorough work, in which we created a test suite including codes for all the candidates (except for some of the asymmetric ones) following a special NESSIE API, along with a special software for the actual measurement.

In total we have tested 285 different implementations for over 138 different variants of the measured primitives, with a total CPU time of several thousands hours on all platforms.

1.2 The submissions received by NESSIE

- **Block ciphers**
  - **Anubis.** [37] (tweaked version is not considered)
  - **Camellia.** [24]
  - **CS-cipher.** [234]
  - **Grand Cru.** [112]
  - **Hierocrypt.** [493] – Hierocrypt-L1 and Hierocrypt-3
  - **IDEA.** [387]
  - **Khazad.** [39] (original+tweak)
  - **MISTY1.** [426]
  - **Nimbus.** [410]
  - **Noekeon.** [180]
  - **NUSH.** [391]
  - **Q.** [434]
  - **RC6.** [324]
  - **SAFER++.** [420]
  - **SC2000.** [569]
  - **SHACAL.** [281] – SHACAL-1 and SHACAL-2.
- **Stream ciphers and pseudo-random number generators**

  - **BMGL.** [285] (original+tweak),
  - **SNOW.** [214] (original+tweak)
  - **SOBER.** [289] – SOBER-t16 and SOBER-t32
  - **LEVIATHAN.** [435]
  - **LILI-128.** [187]

- **Hash functions**
  - **Whirlpool.** [38]

- **Message authentication codes**
  - **UMAC.** [379]
  - **Two-Track-MAC.** [609]

- **Asymmetric encryption**
  - **ACE-KEM.** [563] – Upgrade of ACE-Encrypt
  - **EPOC.** [239] – EPOC-1, EPOC-2 and EPOC-3
  - **ECIES.** [320]
  - **PSEC.** [238] – PSEC-1, PSEC-2, PSEC-3 and PSEC-KEM
  - **RSA-OAEP.** [325] – Revised to RSA-KEM [584]

- **Digital signature schemes**
  - **ACE Sign.** [563]
  - **ECDSA.** [319]
  - **ESIGN.** [244] – Tweaked to ESIGN-D
  - **FLASH family.** [514] – FLASH and SFLASH (has been tweaked)
  - **QUARTZ.** [161]
  - **QUARTZ.** [161] – QUARTZ was tweaked twice; in this document we do not consider the second tweak
  - **RSA-PSS.** [326]

- **Digital identification schemes**
  - **GPS.** [525]

1.3 Performance Evaluation Methodology

Performance evaluation is an essential part in determining the practicality of a cryptographic algorithm. An algorithm that performs well is more likely to be adopted for practical applications.

NESSIE primitives will be used on a variety of platforms: PCs, smart cards, hardware, and in various other applications. Some application areas impose very high performance requirements (such as hard disk encryption) and protection of high speed communications (Gigabit networks). For others, an acceptable performance is required in a low-end hardware platform and/or in compact hardware (cellular phone, smart-card). In general, we retain the candidates which are flexible, i.e. perform well on more than one platform, and those which perform above average on a particular platform.

1.3.1 Performance Criteria

On PCs (under Windows and Linux) and Unix machines, speed is the main concern. We measure speed in the most uniform possible way: using clock counts as measured by the C function `clock()`. The performance of hardware and smartcard implementations also influences the selection.

1.3.2 Comparison with Standards and Well-Known Algorithms

The NESSIE project also takes into account existing and emerging standards, even if these have not been formally submitted to the NESSIE project. Two recent examples in this context come from the standardisation efforts run by NIST. The NESSIE project has contributed extensive comments to the AES process and Vincent Rijmen, one of the designers of the AES algorithm 'Rijndael', is a former member of the NESSIE project team. It was therefore decided that in the evaluation of block ciphers, the Rijndael algorithm should be used as a benchmark. One can expect that the research by NESSIE on the security of block ciphers may well increase the confidence in and acceptability of the Rijndael algorithm as a standard. The NESSIE project also studied the security and performance of SHA-256, SHA-384 and SHA-512, the new hash algorithms recently standardised by NIST to extend the result of SHA-1 to hash results between 256 and 512 bits. The speed performance of candidates is compared with well-known algorithms:

- DES and triple-DES for 64-bit block ciphers,
- Rijndael, the FIPS standard, for 128-bit block ciphers,
- Kasumi, the new 3GPP block cipher,
- Skipjack, the NSA's escrow encryption cipher,
- the AES finalists: Mars, Serpent, Twofish (RC6 was submitted to NESSIE, and for Rijndael see above),
- RC4, such as distributed in the OpenSSL package, for synchronous stream ciphers,
- SHA-1, SHA-256, SHA-384 and SHA-512 for hash functions,
- HMAC-SHA-1 and EMAC (a CBC-MAC) with Rijndael and DES as underlying block ciphers for MACs,
- other non-standard primitives, such as RC5, Seal, Scream, various HMAC's and EMAC's, etc.

2. Theoretical Analysis

2.1 Block Ciphers

Block ciphers are symmetric encryption primitives that typically encipher blocks of 64 or 128 bits. For most block ciphers, the ciphertext is obtained by repeatedly applying a relatively simple cryptographic function to the input. The simple cryptographic function is called a *round function* and the key-derived material used in the round function is called a *round key*. The round keys are computed from the key using a *key-schedule* algorithm. An *SP-network* is a type of block cipher which has the effect of modifying the entire data block in each round. A *Feistel cipher* is a type of block cipher which modifies only half of the block in each round.

The NESSIE call for primitives specified the following security levels for block ciphers:

- *High.* Key length at least 256 bits. Block length at least 128 bits.
- *Normal.* Key length at least 128 bits. Block length at least 128 bits.
- *Normal-Legacy.* Key length at least 128 bits. Block length 64 bits.

Table 1 lists the block ciphers, the sizes of the blocks, and the security levels with the corresponding key sizes. The top part lists the ciphers studied in the second phase of NESSIE, the middle part the other ciphers submitted to NESSIE, and the bottom part other ciphers whose performance we measured.

NOTES: NUSH was designed with two different block sizes: 64 bits and 128 bits. RC6 has a variable block length of $4w$ bits, where $w \geq 32$ is recommended by the designers. There are two variants of SAFER++, one with 64-bit blocks and one with 128-bit blocks.

We can separate the NESSIE block ciphers into five different categories:

1. 64-bit block ciphers: CS-Cipher, Hierocrypt-L1, IDEA, Khazad (and the tweaked variant), MISTY1, Nimbus, SAFER++, NUSH,
2. 128-bit block ciphers: Anubis, Camellia, NUSH, Grand Cru, Hierocrypt-3, Noekeon, Q, SC2000, SAFER++, NUSH, RC6,
3. 160-bit block ciphers: SHACAL-1,
4. 256-bit block ciphers: SHACAL-2, RC6.

A summary of the theoretical results is given in Table 2. More details can be found in D14 [477].

Table 1. Block ciphers submitted to NESSIE, along with block ciphers whose performance was measured by NESSIE.

Name of cipher	Block size	Submitted/tested keysizes		
		Normal Legacy	Normal	High
IDEA	64	128		
Khazad	64	128		
Khazad - tweaked	64	128		
MISTY1	64	128		
SAFER++	64,128	128	128	256
Camellia	128		128, 192	256
RC6	128,256		128	256
SHACAL-1	160			512
SHACAL-2	256			512
CS-Cipher	64	128		
Hierocrypt-L1	64	128		
Nimbus	64	128		
NUSH	64,128	128	128	256
Anubis	128		128, 160, 192, 224	256, 288, 320
Grand Cru	128		128	
Hierocrypt-3	128		128, 192	256
Noekeon Direct	128		128	
Noekeon Indirect	128		128	
Q	128		128	256
SC2000	128		128, 192	256
CAST-128	64	128		
DES	64	56		
Triple-DES	64	168		
Kasumi	64	128		
RC5	64	64		
Mars	128		128,192	256
Rijndael	128,256		128,192	256
Serpent	128		128,192	256
Skipjack	64	80		
Seed	128		128	
Twofish	128		128,192	256

Table 2. Block ciphers theoretical results.

Theoretical Analysis	Block size (bits)	Word size (bits)	Key size (bits)	Sub-key size (bits)	Table lookups (table size in bits×bits)	Shifts& rotations + multiplications	XOR, ADD (bit size)	AND, OR, NOT (bit size)	Total table lookups	Total logical operations (8-bit)	Total multiplications (8-bit)	Code size (kB)	Optimised in submission?
CS-cipher	64	8	128	576	192(8x8)	96(8bit)	488(8bit)	96(8bit)	192	584	0	7	yes
Hierocrypt-L1	64	8	128	896	8(8x8), 48(8x32), 40(8x64)	72(32bit)	60(32bit), 36(64bit)	96(32bit)	96	912	0	12	no
Idea	64	16	128	832		42+34 32-bit multiplications	388 (32bit)		0	1552	136	22.6	no
Khazad	64	8	128	576	64(8x64)	64	64(64bit)		64	512	0	58	no
Khazad - tweaked	64	8	128	576	64(8x64)	64	64(64bit)		64	512	0	58	no
MISTY1	64	32	128	512	24(7x7), 48(9x9)	0	56(32bit), 72(16bit), 24(8bit)	10(32bit), 24(16bit), 10(32bit)	72	390	0	6.9	no
Nimbus6/1	64	64	128	704	80(4x64)	75+5 multiplications	86(64bit)	80(64bit)	80	1328	40		no
Nimbus6/2	64	64	128	704	40(8x64)	35+5 multiplications	46(64bit)	40(64bit)	40	688	40		no
Nimbus6/3	64	64	128	704	5(4x64), 25(12x64)	25+5 multiplications	36(64bit)	30(64bit)	30	528	40		no
NUSH64 (legacy)	64	16	128, 192, 256	1280		36(16bit)	116(16bit)	36(16bit)	0	304	0	10	yes
SAFER++(legacy)	64	8	128	1152	64(8x8)	128	17(32bit), 404(8bit)		64	472	0	35	no
DES	64	32	56	768	128;8(6x4), 8;8(8x32), 64;11(8x48), 16;16(8x64), 0;8(8x56)		6(32bit), 64(48bit), 14(64bit)		216	520	0		n. a.
Triple DES	64	32	56		384;8(6x4), 8;8(8x32), 192;11(8x48), 16;16(8x64), 0;8(8x56)		6(32bit), 192(48bit), 14(64bit)		600	1288	0		n. a.
Anubis ($R = keysize/32 + 8$)	128	8	128-320	$128 * (R+1)$	$192+16*(R-12)$ (8x128)	$180 + 15 * (R-12)$	$192 + 16 * (R - 12)$ (128bit)	$192 + 16 * (R - 12)$ (128bit)	$192+16*(R-12)$	$6144 + 512 * (R-12)$	0	32	no
Camellia	128	8	128	1664	144(8x64)	130	2(128bit), 162(64bit), 8(32bit)	144(64bit), 4(32bit), 4(32bit)	144	2544	0	8.2	no
Camellia	128	8	192 & 256	2176	192(8x64)	174	2(128bit), 216(64bit), 12(32bit)	192(64bit), 6(32bit), 6(32bit)	192	3392	0	8.2	no
Grand Cru	128	8	128	4224	50(5x8), 672(8x8)	144(8bit), 160(8bit)	1372(8bit), 32(8bit)	144(8bit)	722	1548	0	16	no
Hierocrypt-3	128	8	128	1792	16(8x8), 96(8x32), 80(8x128)	144(32bit)	120(32bit), 76(128bit)	192(32bit)	182	2464	0	15	no
Hierocrypt-3	128	8	192	2048	16(8x8), 112(8x32), 96(8x128)	168(32bit)	140(32bit), 91(128bit)	224(32bit)	224	2912	0	15	no
Hierocrypt-3	128	8	256	2304	16(8x8), 128(8x32), 112(8x128)	192(32bit)	160(32bit), 106(128bit)	256(32bit)	256	3360	0	15	no
Noekeon	128	32	128	2048		128	16(128bit), 304(32bit)	32(32bit), 32(32bit), 32(32bit)	0	1856	0	10.5	no
NUSH128	128	32	128, 192, 256	4608		68(32bit)	212(32bit)	68(32bit)	0	1120	0	15	yes
Q (bit-sliced)	128	32		1280	128(8x8)	24	26(128bit), 144(32bit)	56(32bit) 16(32bit)	128	1280	0	11.3	no
RC6 (default)	128	32	128	1408		40(32bit), 80(32bit) +40(32bit) multiplications	40(32bit), 84(32bit)		0	496	160	8	no
SAFER++/128	128	8	128	1920	112(8x8)		15(64bit), 456(8bit)		112	576	0	35	yes, for 8-bit and 32-bit
SAFER++/256	128	8	256	2688	160(8x8)		21(64bit), 648(8bit)		160	816	0	35	yes, for 8-bit and 32-bit
SC2000 (128)	128	32	128	1792	24(10x10), 48(11x11)	48(32bit)	87(32bit), 12(64bit), 14(128bit)	145(32bit)	72	668	0	20	no
SC2000 (192-256)	128	32	192-256	2048	28(10x10), 56(11x11)	56(32bit)	98(32bit), 14(64bit), 16(128bit)	168(32bit)	84	760	0	20	no
Rijndael	128	8	128	1408	160(8x32)	30	11(128bit), 120(32bit)		160	656	0		n. a.
SHACAL-1	160	32	512	2560		224(32bit)	272(32bit), 320(32bit)	180(32bit)	0	3088	0	8	no

2.2 Synchronous Stream Ciphers

The following stream ciphers were submitted to NESSIE:

LILI-128
–
- LEVIATHAN
- SOBER-t16 and SOBER-t32
- SNOW
- BMGL

Table 3 lists the stream ciphers, the security levels and the corresponding key sizes, and the IV size. The top part lists the ciphers studied in the second phase of NESSIE, the middle part the other ciphers submitted to NESSIE, and the bottom part other ciphers whose performance we measured.

Table 3. Stream ciphers submitted to NESSIE, along with stream ciphers whose performance was measured by NESSIE.

Name of Cipher	Submitted Keysizes		IV size
	Normal	High	
BMGL	128		–
BMGL with IV	128		128
Snow	128	256	–
Snow with IV	128	256	64
Sober-t16	128	256	–
Sober-t16 with IV	128	256	128
Sober-t32	128	256	–
Sober-t32 with IV	128	256	128
LEVIATHAN	128, 192	256	–
LILI-128	128		–
RC4	128		–
Scream	128		128
Seal	160		32

A summary of the theoretical results is given in Tables 4 and 5.

More details can be found in D14 [477].

The NESSIE call for primitives specified the following security levels for stream ciphers:

- *High.* Key length of at least 256 bits. Internal memory of at least 256 bits.
- *Normal.* Key length of at least 128 bits. Internal memory of at least 128 bits.

2.3 Collision Resistant and One-Way Hash Functions

The candidates in this category:

- Whirlpool

Table 4. Stream ciphers theoretical results: key setup.

Key setup	Table lookups	Shifts	Rotations	And Or Xor	Add
BMGL (8-bit word)	560	0	0	0	544
Snow128 (32-bit word)	256	268	64	736	158
Snow256 (32-bit word)	256	280	64	768	174
Sober-t16/128 (16-bit word)	371	70	0	189	212
Sober-t32/128 (32-bit word)	371	78	0	253	242
LEVIATHAN 128 or 256					
LILI-128					
RC4	256(log$_2$ (byte-keylength) x8bit), 512(8x8bit), 512 table storages (8x8bit)	0	0	0	512

Table 5. Stream ciphers theoretical results: key stream generation.

Key stream generation	Table lookups	Shifts	Rotations	And Or Xor	Add	Output bits /LFSR cycle
BMGL (32bit word)	160	80	0	224	20	32
Snow128 (32bit word)	64	112	16	224	77	32
Snow256 (32bit word)	64	112	16	224	77	32
Sober-t16/128 (16bit word)	138	39	0	156	63	16
Sober-t32/128 (32bit word)	288	111	0	262	119	32
LEVIATHAN 128 or 256						
LILI-128						
RC4	3(8x8bit), 2(8x8bit) table storages	0	0	0	3	8[1]

[1] This number measures bits/stream cipher cycle since there is no LFSR for RC4.

- SHA-1
- SHA-256
- SHA-384
- SHA-512

Table 6 lists the hash functions and the hash sizes. The top part lists the hash function submitted to NESSIE, namely Whirlpool, and the bottom part other hash functions whose performance we measured. Note that BCHASH-Rijndael is the hash function that results by replacing the internals of the compression function of SHA-1 by Rijndael (leaving the feed-forward mixing untouched), i.e., by replacing the SHACAL-1 part of SHA-1 by Rijndael (or in other words BCHASH is defined such that SHA-1 = BCHASH-SHACAL-1).

Table 6. Hash functions submitted to NESSIE, along with hash functions whose performance was measured by NESSIE.

Name of Primitive	Hash Sizes
Whirlpool	512
MD4	128
MD5	128
RIPEMD	160
SHA-0	160
SHA-1	160
SHA-256	256
SHA-384	384
SHA-512	512
Tiger	192
BCHASH-Rijndael	128

A summary of the theoretical results is given in Table 8. More details can be found in D14 [477].

2.4 Message Authentication Codes

There are two candidates in this category:

- Two-Track-MAC
- UMAC

Table 7 lists the message authentication codes (MACs), the MAC sizes, and the key sizes. The top part lists the MACs submitted to NESSIE, and the bottom part other MACs whose performance we measured. The implemented CBC-MAC is EMAC as defined by Algorithm 2 of the ISO/IEC 9797-1 [303], i.e., the message is padded by one 1-bit, followed by as many 0-bits as required to fill the last block. Then, the result is CBC-encrypted with the EMAC key, and the last block of the ciphertext is encrypted again with the key obtained by XORing all bytes of the EMAC key with $\texttt{F0}_\texttt{x}$. The MAC value is the result of this last encryption.

Table 7. MACs submitted to NESSIE, along with MACs whose performance was measured by NESSIE.

Name of Primitive	MAC Sizes	Keysizes
TTMAC	160	160
UMAC	64	160
HMAC-Whirlpool	512	512
HMAC-MD4	128	512
HMAC-MD5	128	512
HMAC-RIPE-MD	160	512
HMAC-SHA-0	160	512
HMAC-SHA-1	160	512
HMAC-SHA-256	256	512
HMAC-SHA-384	384	512
HMAC-SHA-512	512	512
HMAC-Tiger	192	512
EMAC-Rijndael	128	128
EMAC-DES	64	56
EMAC-SHACAL-1	512	160

Table 8. Hash functions and message authentication codes theoretical results.

Theoretical Analysis	**Word size (bits)**	**Sub-key size (bits)**	**Table look-ups (table size in bits× bits)**	**Shifts/ Rotations (bit size)**	**XOR, ADD, MULT (bit size)**	**AND, OR, NOT (bit size)**	**Total table look-ups**	**Total logical operations (8-bit)**	**Total multiplications (8-bit)**	**Code size (kB)**	**Optimised in submission?**
Whirlpool	8	5120	1280 (8x64)	1188	1175 (64bit)	1268 (64bit)	1280	19544	0	65	no
SHA-1	32			0/224 (32bit)	312 (32bit), 325 (32bit)	100 (32bit)	0	2948	0		
SHA-256	32			96 (32bit)/ 576 (32bit)	640 (32bit), 600 (32bit)	384 (32bit)	0	6496	0		
SHA-384, SHA-512	64			128 (64bit)/ 736 (64bit)	816 (64bit), 760 (64bit)	480 (64bit)	0	16448	0		
UMAC16	16	8192			2048 (16bit), 1024 (32bit), 1024 (32bit mult.)		0	8192	4096	146	no
UMAC32	32	8192			512 (32bit), 256 (64bit), 256 (64bit mult.)		0	4096	2048	146	no
TTMAC	32	160		320	128 (32bit), 613 (32bit)	384 (32bit)	0	4500	0	13	no

A summary of the theoretical results is given in Table 8. More details can be found in D14 [477].

2.5 Asymmetric Encryption Schemes

The primitives in this category are:

- ACE Encrypt – tweaked to ACE-KEM
- ECIES
- EPOC-1
- EPOC-2 (tweaked)
- EPOC-3
- PSEC-1
- PSEC-2 – tweaked to PSEC-KEM
- PSEC-3
- RSA-OAEP
- RSA-KEM

More details can be found in D14 [477].

Table 9. Usual parameter lengths (in bytes).

Scheme	Public Key Length	Private Key Length	Ciphertext Length for 16-Byte Messages
ACE-Encrypt	1348	160	432
ECIES	20	20	36
EPOC-1	432 (288)	144	160
EPOC-2	432 (288)	144	160
EPOC-2 - tweaked	432 (288)	144	160
EPOC-3	432 (288)	144	160
PSEC-1	20	20	60
PSEC-2	20	20	76
PSEC-3	20	20	96
RSA-OAEP	129	256 (129)	128
ACE-KEM over $(\mathbb{Z}/p\mathbb{Z})^\times$	512	80	400
ACE-KEM over EC	80	80	76
PSEC-KEM	20	20	52

[1] For elliptic-curve-based schemes we assume that a point-compression technique is used.

Table 9 contains the practical parameter lengths and the ciphertext length for each scheme. The parameter lengths are chosen as follows. For schemes based on integer factorisation, the modulus is a 128-byte integer, for schemes based on discrete logarithm, the cyclic group is $(\mathbb{Z}/P\mathbb{Z})^\times$ for P of length 128 bytes or an elliptic curve where the field size and the size of the order of the base point are 20 bytes. For schemes based on discrete logarithm, the decryption of the group and the base point are not considered to be part of the key. The private key also includes the public parameters needed to decrypt. Sometimes, the key size can be reduced, but the encryption/decryption time will then be increased by some preprocessing; this reduced key size is in parentheses.

The table also contains the ciphertext length corresponding to messages of size 16 bytes. We assume the output of the hash functions is 20 bytes (160 bits), the output of the symmetric encryption scheme used in the EPOC and PSEC schemes is 16 bytes (128 bits), and that the length of the random string (salt) is 16 bytes.

Table 10 describes the number of group operations required by each scheme.

Table 10. Number of group operations for each asymmetric encryption scheme.

	RSA		EPOC-2		RSA-OAEP		ECIES		PSEC-KEM		ACE-KEM	
	ENC	DEC	ENC	DEC	ENC	DEC	ENC	DEC	ENC	DEC	ENC	DEC
group exponentiations	1	1	2	3	1	1	2	1	2	2	5	3
group multiplications	–	–	1	2	–	–	–	–	–	–	1	–
random numbers	–	–	1	–	1	–	1	–	1	–	1	–
hash calls [1]	–	–	3	3	3	3	1	1	2	2	2	2
symmetric cipher calls	–	–	1	1	–	–	1	1	1	1	1	1
MAC calls	–	–	–	–	–	–	1	1	–	–	–	–

[1] This includes calls to functions that predominantly rely on hash functions, including key derivation functions (KDFs) and mask generating functions (MGFs).

2.6 Asymmetric Digital Signature Schemes

The primitives in this category are:

- ACE Sign
- ECDSA
- ESIGN – tweaked to ESIGN-D
- RSA-PSS
- FLASH
- SFLASH (old version) – tweaked to SFLASH (new version)
- QUARTZ (tweaked)

Details about them can be found in D14 [477].

Table 11 and Figure 1 specify the lengths of the data handled by the algorithms. Most important are the signature size, which can cause an overhead for a message transmission, and the public key size, which has to be handled by the public key infrastructure. When (a part of) the public key is needed to sign, we include the corresponding length in the private key length.

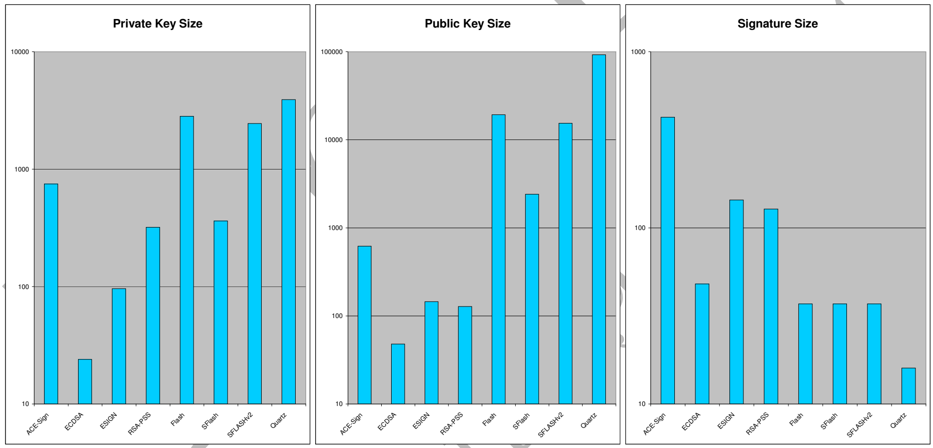
For most submitted signature schemes, security and performance depend on the choice of some parameters. For table 11, the choices have been 1024 bit moduli for two factors based schemes (ACE Sign and RSA-PSS), 1152 bit moduli for three factors based schemes (ESIGN), and 163 bit base field for discrete logarithm based schemes (ECDSA). The RSA public exponent is fixed to 3.

Most of the submitted schemes use a hash function which gives a limitation on the message length, e.g., 2^{61} bytes for SHA-1. Only ACE-Sign has a signature length depending on the message length. We assume that the messages to be signed have lengths between 20 bytes and 1 Mbyte. The signature length of ACE-sign can be made constant using CRHF instead of UOWHF.

Table 11. Data lengths.

Scheme	Public Key	Private key	Signature
ACE-Sign	620 bytes	748 bytes	425 to 705 bytes
ECDSA	48 bytes	24 bytes	48 bytes
ESIGN	145 bytes	96 bytes	144 bytes
RSA-PSS	128 bytes	320 bytes	128 bytes
FLASH	19266 bytes	2822 bytes	37 bytes
SFLASH (old version)	2409 bytes	362 bytes	37 bytes
SFLASH (new version)	15400 bytes	2450 bytes	37 bytes
Quartz (and tweak)	71 to 90 kB	3914 bytes	16 bytes
KCDSA	48 bytes	24 bytes	48 bytes

Fig. 1. Data lengths.



2.7 Asymmetric Identification Schemes

The only candidate in this category is:

- GPS (tweaked)

Details can be found in D14 [477].

3. Claimed Performance

Here, we present performance results, mainly concentrating on the C code running on a Pentium III.

3.1 Block Ciphers - Legacy

3.1.1 CS-Cipher

Efficiency results for standard C implementation on a Pentium 133 MHz.

Processor	Encryption (cycles/block)	Encryption (cycles/byte)
Pentium	4053.12	506.64

3.1.2 Hierocrypt-L1

Best results in 10 trials to carry out 1000000 block encryptions in ECB mode for 128-bit key version on Pentium III 650 MHz, running MS Windows 98 CE, and MSVC++ 6.0. Data obtained from *Proceedings of the 2nd NESSIE Workshop*.

Processor	Encryption Decryption (cycles/block)	Encryption Decryption (cycles/byte)	Encryption Decryption (cycles/byte)	Encryption Decryption (cycles/byte)
Pentium	199	374	24.8	46.7

3.1.3 IDEA

Here we give the performance claimed by the submitters for their candidate.

Processor	Encryption Decryption (Mbits/sec)	Encryption Decryption (cycles/byte)
90 MHz Pentium	4.2	171.4
366 MHz Pentium II	31	94.4
600 MHz Pentium III	61	78.7

In the next table we present the results given by Lipmaa, who uses MMX and encrypts 4 blocks in parallel (i.e., CBC mode cannot be performed with this speed).

Processor	Encryption Decryption Block size	Encryption Decryption (cycles/block)	Encryption Decryption (Mbytes/sec)	Encryption Decryption (cycles/byte)
Pentium III	4x64	440	55.5	13.75
Pentium MMX	4x64	543	45.0	16.9
Pentium MMX	64	358	17.0	44.7

3.1.4 Khazad and its Tweaked Variant

Khazad is an involutional cipher, so the effort required for encryption and decryption is the same. Because of this, decryption is not obtained by applying the encryption components in reverse order, and so the key schedules for encryption and decryption are not identical. The tweaked variant should have the same performance as the original variant.

Processor	Language	Key setup (cycles/key)	Encryption Decryption (cycles/byte)	Encryption Decryption (Mbits/sec)

Pentium III [1]	ANSI C	717(encrypt)
		1206(decrypt)
			67	65.7

[1] IBM PC/ AT compatible PC, Intel Pentium III, 550 MHz.

3.1.5 MISTY1

Processor	Language	Key setup (cycles/key)	Encryption (cycles/byte)	Encryption (Mbits/sec)
Pentium II [1]	ANSI C	170	56.25	71.1
Pentium II [2]	ANSI C	300	37.5	106.7

[1] IBM PC/ AT compatible PC, Intel Pentium II, 500 MHz, 128 MB memory, Windows 98, straightforward implementation.

[2] IBM PC/ AT compatible PC, Intel Pentium II, 500 MHz, 128 MB memory, Windows 98, optimised implementation requiring more memory.

3.1.6 NUSH

Efficiency results for C implementation.

Processor	Encryption (cycles/block)	Key setup (cycles/key)	Encryption (cycles/byte)
Pentium	180	64	22.5

3.1.7 SAFER++

Processor	Key setup[1] (cycles/key)	Encryption[2] (cycles/byte)	Decryption[2] (cycles/byte)
Pentium	1333	169	160

[1] Efficiency of reference implementation (not optimised).

[2] Efficiency results for 128-bit key version on Pentium III 667 MHz, running MS Windows 2000, and MSVC++ 6.0 (optimised source code).

3.1.8 DES

The performance of DES has been studied well in the past; we give here some results to compare with the NESSIE candidates.

		Encryption		
Origin	Processor	(cycles/byte)	(Mbits/sec)	(MBytes/sec)
Bosselaers	90 MHz Pentium	42.5	16.9	2.12

3.2 Block Ciphers - Normal and High (128-bit blocks)

3.2.1 Anubis

Anubis is an involutional cipher, so the effort required for encryption and decryption is the same. Because of this, decryption is not obtained by applying the encryption components in reverse order, and so the key schedules for encryption and decryption are not identical.

Processor Language	Key size	Key setup (cycles/key)	Encryption Decryption (cycles/byte)	Encryption Decryption (Mbits/sec)

Pentium III [1]
ANSI C
	128	3352(encrypt)
		4527(decrypt)
			36.8	119.5

	160	4445(encrypt)
		5709(decrypt)
			39.3	112.1

	192	6644(encrypt)
		8008(decrypt)
			41.6	105.9

	224	8129(encrypt)
		9576(decrypt)
			43.8	100.5

	256	9697(encrypt)
		11264(decrypt)
			46.3	95.1

	288	11385(encrypt)
		11291(decrypt)
			48.5	90.7

	320	13475(encrypt)
		15169(decrypt)
			50.8	86.6

[1] Intel Pentium III, 550 MHz.

3.2.2 Camellia

Processor Language	Key size	Key setup (cycles/key)	Encryption (cycles/byte)	Encryption (Mbits/sec)
Pentium III [1] Assembly	128	160	23.1	241.5
	192	222	30.8	181
	256	226	30.9	181
Pentium II [2] ANSI C	128	263	36	66.6

[1] IBM PC/ AT compatible PC, Intel Pentium III (700 MHz), 256 KB L2 cache, FreeBSD 4.0R, 128 MB main memory.

[2] IBM PC/ AT compatible PC, Intel Pentium II (300 MHz), 512 KB L2 cache, Windows 95, 160 MB main memory.

3.2.3 Grand Cru

Efficiency results of reference ANSI C implementation for 128-bit key version using gcc on an Intel Pentium 200 MHz, running Linux 2.0.38, and on an Intel XEON 550 MHz, running Linux 2.2.14.

Processor	Encryption (cycles/byte)	Decryption (cycles/byte)	Key setup (cycles/key)
Pentium	2812.5	4062.5	200000
XEON	4062.5	5625	300000

3.2.4 Hierocrypt-3

Best results in 10 trials to carry out 1000000 block encryptions in ECB mode for 128-bit key version on Pentium III 650 MHz, running MS Windows 98 CE, and MSVC++ 6.0. Data obtained from *Proceedings of the Second NESSIE Workshop.*

Processor	Key size	Encryption (cycles/byte)	Decryption unoptimised	Key setup (cycles/key)
Pentium	128	37.5	63.2	370
	192	44.4	78.1	386
	256	50.5	88.7	468

3.2.5 Noekeon

Note that Noekeon has two key schedules. In direct mode the user-selected key is used directly, and so requires no operations before encryption or decryption can start. In indirect mode the user-selected key is encrypted once using the all-zero key, and the ciphertext becomes the working key. This indirect key schedule thus takes the same time as one encryption.

Processor Language	Key size	Key setup (cycles/key)	Encryption Decryption (cycles/byte)

Pentium II [1]
ANSI C
	0(direct)
	525(indirect)
		32.8	48.8

[1] IBM PC/ AT compatible PC, Intel Pentium II, 200 MHz, Windows NT 4.0, Microsoft Visual C/C++ 6.0 compiler.

3.2.6 NUSH

Efficiency results for C implementation.

Processor	Key Size (bits)	Encryption (cycles/byte)	Key setup (cycles/key)
Pentium	128	21.2	112

3.2.7 Nimbus

Processor	Block Size	Encryption (cycles/byte)	Key setup (cycles/key)
Pentium	64	66	24665

3.2.8 Q

Processor	Key Size	Encryption Decryption (cycles/byte)	Key setup (cycles/key)
Pentium	128	36.5	500

3.2.9 RC6

We present the performance claimed by the submitters of RC6, for the C code, in cycles/byte for 20-round and 128 bit-block version.

Processor	Compiler	Encryption Decryption (cycles/byte)
Pentium II	Borland	39
Pentium Pro	MSVC	30
	MSVC + optimiser	16
	GCC	26
	GCC + optimiser	23

3.2.10 SAFER++

Processor	Key size	Key setup [1] (cycles/key)	Encryption [2] (cycles/byte)	Decryption [2] (cycles/byte)
Pentium	128	2333	40	76
	256	3227	57	101

[1] Efficiency of reference implementation (not optimised).
[2] efficiency results for 128-bit key version on Pentium III 667 MHz, running MS Windows 2000, and MSVC++ 6.0 (optimised source code).

3.2.11 SC2000

Processor	Key size	Key setup (cycles/key)	Encryption [1] (cycles/byte)	Decryption [1] (cycles/byte)
Pentium	128	488	23.93(assembly)	25.187(assembly)
	192	525	27.37(assembly)	28.75(assembly)
	256	526	27.37(assembly)	32.81(assembly)

[1] Best efficiency results for 128-bit key version in MSVC++ 6.0 + Assembly (inline) on an Intel Pentium III 550 MHz, running Microsoft Windows NT 4.0; key schedule in C language.

3.2.12 Rijndael

Rijndael was studied well in the AES process. We give some significant results to compare them with those of the NESSIE candidates, in addition to our own tests.

Origin	Processor	Block size	Cycles/byte	MBytes/s
Lipmaa	Pentium II/III	128	28.6	53.3

3.3 Block Ciphers - Normal and High (large blocks)

3.3.1 SHACAL-1

Processor	Block size (bits)	Encryption (cycles/byte)	Decryption (cycles/byte)	Key setup (cycles/key)
Pentium II/III	160	140	116.5	3200
updated	160	124	116	2280

The SHACAL-1 submitters estimate the computational efficiency at 2800 cycles per 20-bytes-block encryption (block size), 2330 per 20-byte-block decryption and 3200 per 64 byte key setup, on a PC with an AMD K6 processor running at 233 MHz.

3.3.2 SHACAL-2

Processor	Block size (bits)	Encryption (cycles/byte)	Decryption (cycles/byte)	Key setup (cycles/key)
Pentium II/III	256	112.5	115	2800

3.4 Stream Ciphers

3.4.1 BMGL

No claimed performance given.

3.4.2 LEVIATHAN

No claimed performance given.

3.4.3 LILI-128

Processor	Key size (bits)	Key setup (cycles/byte)	Keystream generation (cycles/byte)
Pentium III 650 MHz	128	–	1200

3.4.4 Snow

Processor	Key size (bits)	Key setup (cycles/byte)	Keystream generation (cycles/byte)
Pentium III	128	2000	6.75
500 MHz	256	2000	6.75

3.4.5 Snow with IV

Processor	Key size (bits)	Key setup (cycles/byte)	IV setup (cycles/byte)	Keystream generation (cycles/byte)
Pentium III	128	few	1000	6.75
500 MHz	256	few	1000	6.75

3.4.6 Sober-t16

Processor	Key size (bits)	Key setup (cycles/byte)	Keystream generation (cycles/byte)
Sun Ultrasparc	0	1084	42.6
248 MHz	64	1346	42.6
	128	1581	42.6
	192	1812	42.6
	256	2062	42.6

For Sober-t16 with IV, there are no claimed results for the IV setup.

3.4.7 Sober-t32

Processor	Key size (bits)	Key setup (cycles/byte)	Keystream generation (cycles/byte)
Sun Ultrasparc 248 MHz	0	1110	24.51
	64	1208	24.51
	128	1348	24.51
	192	1480	24.51
	256	1564	24.51

For Sober-t32 with IV, there are no claimed results for the IV setup.

3.5 Hash Functions

3.5.1 Whirlpool

Processor	Hash (cycles/byte)
Pentium III 550 MHz	133

3.6 MACs

3.6.1 Two-Track MAC

Processor	MAC (cycles/byte)	Key setup (cycles/key)
Pentium [1]	95.8	10
Assembly	16	10

[1] Efficiency results for non-optimised reference C code.

3.6.2 UMAC

Processor	Message Size (bytes) [1]			
	43	256	1500	$256 \cdot 2^{10}$
Pentium	16.3	3.8	2.1	1.9

[1] Efficiency results in cycles/byte on a 700 MHz Pentium III under gcc 2.95, mixed C/Assembly. Figures for UMAC32 submitted to NESSIE and simply called UMAC. Data obtained from *Proceedings of First NESSIE Workshop.*

3.7 Asymmetric Primitives

The claimed performance of the asymmetric encryption schemes is given in Table 12. Chinese remainders are used for RSA. No new claims have been made for

Table 12. Claimed performance of the asymmetric encryption schemes.

Scheme	Key setup		Encryption		Decryption		Architecture
	time	cycles	time	cycles	time	cycles	
ACE-Encrypt	14 sec	3724M	230 ms	61M	97 ms	26M	Pentium @ 266
PSEC-1			4.4 ms	3080K	4.4 ms	3080K	Pentium @ 700
PSEC-2			4.4 ms	3080K	4.4 ms	3080K	Pentium @ 700
PSEC-3			4.4 ms	3080K	2.4 ms	1680K	Pentium @ 700
ECIES			5.8 ms	1160K	4.4 ms	880K	Pentium @ 200
EPOC-1			13 ms	9100K	20 ms	14M	Pentium @ 700
EPOC-2			10 ms	7000K	17 ms	12M	Pentium @ 700
EPOC-3			10 ms	7000K	7 ms	4900K	Pentium @ 700
RSA-OAEP			1 ms	450K	27 ms	12M	Celeron @ 450
PSEC-KEM			4.4 ms	3080K	4.4 ms	3080K	Pentium @ 700

the tweaks ACE-KEM, PSEC-KEM and EPOC-2. The changes for the latter two should not influence their performance. ACE-KEM is faster than ACE-Encrypt.

The claimed performance of the asymmetric digital signature schemes is given in Table 13.

Table 13. Claimed performance of the digital signature schemes.

Scheme	Key setup		Signature		Verification		Architecture
	time	cycles	time	cycles	time	cycles	
ACE-Sign	36 sec	9576M	62 ms	16492K	73 ms	19418K	Pentium @ 266
ECDSA $\mathbb{F}_{2^{163}}$	1.5 ms	600K	2.1 ms	840K	4.1 ms	1640K	Pentium @ 400
ECDSA $\mathbb{F}_p$	2.1 ms	840K	2.6 ms	1040K	6.5 ms	2600K	Pentium @ 400
ESIGN			3.4 ms	2980K	0.4 ms	280K	Pentium @ 700 *
RSA-PSS	1.9 sec	855M	27 ms	12.1M	3 ms	1350K	Celeron @ 450
FLASH			49 ms	24.5M	50 ms	25M	Pentium @ 500
SFLASH (old)			44 ms	22M	50 ms	25M	Pentium @ 500
SFLASH (new)*	1 sec	500M	2.7 ms	1350K	0.8 ms	400K	Pentium @ 500
Quartz			30 sec	15G	50 ms	25M	Pentium @ 500
Quartz, tweaked*	4 sec	2000M	10 sec	5000M	0.9 ms	450K	Pentium @ 500

The different performances of the two versions of SFLASH are due to a more efficient implementation of the new version. The same holds for Quartz and its tweak.

The claimed performance of the GPS and its tweak is given in Table 14.

Table 14. Claimed performance of GPS and its tweak.

Scheme	Key setup		Commitment		Answer		Verification		Architecture
	time	cycles	time	cycles	time	cycles	time	cycles	
GPS and tweak	0.7 sec	315M	10 ms	4500K	1 μs	450	12 ms	5400K	Pentium @ 450

4. Practical Software Implementation

We tested the performance of 285 implementations of 138 different variants of primitives. The tests were performed on 11 different kinds of platforms (on over 20 computers), on various operating systems and with various compilers. On some processors (e.g, Pentiums) we made the tests on two operating systems with 4 compilers, and even several different versions of some compilers, in order to achieve the best results. In total, our performance tests ran several thousands of computer hours.

4.1 Measurements for Symmetric Primitives

We tested all NESSIE symmetric candidates along with standard primitives and many 'non-standard' primitives. Our tool measures the time for key setup, encryption, decryption, IV setup, and hash and MAC initialisation and finalisation. The tool checks the correctness of all codes by comparing the encryption results to the supplied test vectors. For encryption, e.g., we measure the time in the following way (decryption, key setup, ... analogously):

- First, we encrypt random plaintexts for about one second. Based on the number of plaintexts encrypted in one second, we estimate how many encryptions are expected to run in 10 seconds.
- Then, we run the estimated number of encryptions and measure their run time.

The actual measurement is executed with many different keys for many different encryption/decryption blocks. We calculate the encryption, decryption, hash, MAC time in units of *cycles/byte* and the key setup, IV setup time and hash and MAC initialisation and finalisation in *cycles/invocation*.

We compare the results on various machines (The machines are listed in Table 29.) The speed results for the various types of processors and operating systems are given in Tables 30–35, where Table 30 summarises the results for legacy block ciphers, Table 31 those for normal and high block ciphers, Table 32 those for block ciphers with 160- and 256-bit blocks, Table 33 those for stream ciphers, Table 34 those for hash functions, and Table 35 those for MACs. Tables 39–48 present these results for various processors and compilers. Figures 2–23 show these results as bar histograms, and Figures 24–45 show these results in a sorted order (sorted by performance). Each table is divided into three parts: the first part contains the candidates accepted for the second phase of NESSIE, the second part contains the other candidates submitted to NESSIE, and the third part contains primitives that we implemented and compared, but were not submitted to NESSIE; the (unsorted) figures are ordered in the same way (from bottom to top), Table 33 (stream ciphers) and the corresponding figures have an additional division for ciphers which do not accept initial values, and ciphers that require initial values. Our results show very high consistency between different machines of the same type, especially between various PIIIs (at different speeds and with different memory sizes). In many cases we obtained clock counts with differences of less than one unit.

For the measurement of speed, we compiled all ciphers with all the available compilers, with various optimisation options (as adequate for the machine and compiler), and selected the best speed that resulted from all these options. In many cases, higher optimisations (such as -O3) resulted in poorer speeds than lower optimisations (such as -O1), and in many cases optimisations targeted to older processors (such as optimisation targeted for 386 when running on PIII) gave better results than optimisations targeted to the newer ones (such as Pentium or Pentium-pro). For this reason, on most machines, our measurements consisted of more than a dozen compilations with different optimisation options and target machines, to ensure that we do not miss the best code that the compiler can generate. In the case of PIII with Linux, we performed the measurement under three different versions of the gcc compiler, with over 40 different optimisation options for the newer version.

We also ensured that the compiled code is correct by regenerating the test vectors in each run with each compilation option. In those rare cases where some compilation option generated wrong code on some machine, we ignored the speed results of the runs with the wrong results. For example, the compilations of Anubis on Alpha machines generated wrong codes when the optimisation options were -O2 or higher. We ignored the resultant speeds of these wrong runs, although they were faster than the speeds we will list later.

We also ensured that the main test program was compiled with the same optimisation option in all cases, although the code of the primitives was compiled with different options, in order to make the overhead of the test program as fixed as possible. In order to measure this overhead, we measured the speed of dummy ciphers (that do nothing) and verified that their computation time is negligible. It should be noted that all codes (of each family of primitives) use the same API (which, among other things, ensures that the keys are set up into structures that can later be passed as parameters to the encryption (decryption, etc.) function, and that no global or static variables depending on the key, state, ... are used), and thus, the overhead of all codes of the same type and block size is expected to be similar.

It can be seen from the results that the codes for the primitives are quite optimised. This is the result of several rounds of optimisations of the submitted codes by several people in the NESSIE project. For about 50% of the ciphers, our codes are even faster than the submitters claim, and for several others ciphers, the results are only a few cycles slower than claimed. In particular, we wish to refer the reader to our optimisation of Camellia, which uncovers the design of Camellia, whose round function is designed for 32-bit processors, although the description in the submission only describes a byte-level implementation. We also wish to mention our novel optimisation of Idea [63], which is, as far as we know, the fastest implementation of Idea in C.

In most cases, the order of the primitives by decreasing speed is similar on all machines. Exceptions are primitives that are optimised for 64-bit machines, which become the fastest on Alpha, although they are not so on other machines. Two examples are RC6 with 256-bit blocks (using 64-bit multiplications), which, on Alpha, is even twice faster than the standard RC6 with 128-bit blocks, although it is twice slower on all other machines, and Tiger, which, on Alpha, becomes even faster than MD4, although it only has a medium speed on all other machines.

Note that the same implementations of block ciphers and stream ciphers were also subjected to the NESSIE statistical tests, and all of them passed these tests.

We have also measured the amount of memory required for the various implementations, and verified that the speeds we report can be reached with a reasonable amount of memory.

We did not distinguish the cases where the key setup can be faster for encryption-only or decryption-only applications. In all cases, we measured the time required for a full key setup. In the cases of Idea, Rijndael, and several others, the time required to setup encryption-only keys may be significantly faster than the full time of the key setup.

The test vectors we used for verifying the correctness of our codes, were sent to the submitters for verification. In Table 15, we list the status of these test vectors in February 2003.

Finally, a primitive is listed with two implementations in cases where there are inherent tradeoffs between the listed speeds, e.g., when encryption can be made faster with a slower key schedule, or when one implementation uses a feature that is usually not used in our implementations (e.g., MMX instructions).

4.1.1 Non-NESSIE Assembly Results

The NESSIE project did not implement the primitives in assembly languages, except for a few MMX-optimised codes that were written in the C files and processed by the C compilers. For the selection of the primitives, it was not necessary to know the exact speedup gained by assembly code, and the average saving of a few cycles per byte was not worth directing our cryptanalytic efforts towards additional implementation work. For completeness, we list in Table 16 the claimed speeds of non-NESSIE implementations.

4.2 Measurements for Asymmetric Primitives

We chose the smallest parameters for which the submissions claimed to have a security of 2^{80}, i.e., 1024 or 1152 bits for factorisation or discrete logarithm based schemes, and 160 bits for elliptic curves. We use an RSA public exponent of 3,

Table 15. Status of the test vectors of the symmetric primitives.

	Verified by Submitters	**Compared to Submitted Vectors**	**Unknown**	**Non-NESSIE, Verified by Designers**	**Non-NESSIE, Verified by Us**	**Unverified**
Block Ciphers	Anubis Camellia Idea Khazad Hierocrypt-L1 Hierocrypt-3 MISTY1 Noekeon NUSH SAFER++ SHACAL-1 SHACAL-2 RC6	CS-cipher Grand Cru Nimbus Q SC2000		Kasumi Serpent CAST-128 RC5	DES Triple-DES Rijndael Mars Twofish Skipjack	Seed
Stream Ciphers	BMGL Snow Sober		LEVIATHAN LILI-128	Scream-S RC4	Scream-0/F Seal	
Hash Functions	Whirlpool			Tiger	SHA-0 SHA-1 SHA-256 SHA-384 SHA-512 MD4, MD5	RIPEMD BCHASH-Rijndael**
MACs	TTMAC UMAC			HMAC-Tiger	EMAC-DES	HMAC-Whirlpool** HMAC-SHA-0** HMAC-SHA-1* HMAC-SHA-256** HMAC-SHA-384** HMAC-SHA-512** HMAC-MD4* HMAC-MD5* HMAC-RIPE-MD** EMAC-Rijndael** EMAC-SHACAL**

* Verified consistency of several implementations, and verified relatively to the underlying hash function

** Verified relatively to the underlying primitive

Table 16. Assembly performance results.

Primitive	**Machine**	**Speed (cycles/byte)**		**Reference**
		Encryption	Decryption	
Camellia	Sparc	22.2	22.2	[173]
	Alpha	17.6	17.6	[173]
4-Way IDEA	PIII	13.75		[400]
	PI/MMX	16.96		[400]
MISTY1	PII	26.6	26	[173]
	Alpha	25.4	25.8	[173]
RC6	P.ProII	16		[558]
	Pentium	44		[558]
	PIII	22.18		[401]
Rijndael	PIII	14.13	14.93	[401]
	PIII	14.8		[173]
	P.ProII	18		[558]
	Pentium	20		[558]
Mars	P.ProIII	20		[558]
	Pentium	34		[558]
Twofish	P.ProII	16		[558]
	Pentium	19		[558]
WHIRLPOOL	PIII/MMX	36.52		[461]
SHA-1	PIII/MMX	8.3		[461]
SHA-256	PIII/MMX	20.59		[461]
SHA-512	PIII/MMX	40.18		[461]
MD5	PIII/MMX	4.31		[461]
RIPEMD-160	PIII/MMX	11.34		[461]
UMAC	PII	1.93		[91]
	PPC	1.58		[91]
	Alpha	2.78		[91]

type B parameters for (untweaked) EPOC, elliptic curves over the binary prime field GF(2^{163}) for ECIES and ECDSA and over a prime field for PSEC. Note that OEFs (Optimal Extension Fields) are more efficient but may weaken the discrete logarithm assumption. (For EPOC-2-tweaked, a different set of parameters is required.)

The measurement was performed with the same test suite and the same processors, operating systems and compilers as the symmetric case, but in the asymmetric case, not all primitives were implemented, due to the inherent slow speeds of these primitives and the fact that it is easy to derive approximate times for some primitives from the times of others (such as when the key generations are the same for several primitives, or when the most time-consuming operation is common for several primitives, as in the case of modular exponentiation that dominates the computation time). However, note that our asymmetric implementations are not optimised, and should only be treated as an order of magnitude.

4.2.1 Asymmetric Encryption Schemes

The results of our performance tests for asymmetric encryption schemes are given in Tables 36 and 49, where the top part lists results of (not necessarily optimised) codes of the NESSIE test suite, and the bottom part lists results based on the submitters' codes (without all the extra features and tests of the suite).

Table 17. Estimated performance of asymmetric encryption schemes.

Scheme	Encryption		Decryption		
	ms	cycles	ms	cycles	group size
ACE-Encrypt	50 ms	25M	45 ms	22.5M	1024-bit
PSEC-1	5 ms	2500K	5 ms	2500K	160-bit
PSEC-2	5 ms	2500K	5 ms	2500K	160-bit
PSEC-3	5 ms	2500K	2.5 ms	1250K	160-bit
ECIES	5 ms	2500K	2.5 ms	1250K	160-bit
EPOC-1	15 ms	7.5M	20 ms	10M	1152-bit
EPOC-2	10 ms	5M	15 ms	7.5M	1152-bit
EPOC-2 - tweaked	10 ms	5M	15 ms	7.5M	1152-bit
EPOC-3	10 ms	5M	7 ms	3.5M	1152-bit
RSA-OAEP (e=3)	1 ms	500K	11 ms	5.5M	1024-bit
ACE-KEM over $\mathbb{F}_p^*$	50 ms	25M	35 ms	17.5M	1024-bit
ACE-KEM over EC	12.5 ms	6250K	7.5 ms	3750K	160-bit
PSEC-KEM	5 ms	2500K	5 ms	2500K	160-bit
Rabin-SAEP	0.5 ms	250K	11 ms	5.5M	1024-bit

Table 17 shows estimations of the performance for encryption and decryption on a Pentium III desktop, based on the theoretical analysis of Section 2. RSA-KEM has similar performance as RSA-OAEP.

4.2.2 Asymmetric Digital Signature Schemes

The results of our performance tests for asymmetric digital signature schemes are given in Tables 37 and 50, where the top part lists results of (not necessarily optimised) codes of the NESSIE test suite, and the bottom part lists results (mostly) based on the submitters' codes (without all the extra features and tests of the suite).

We also tested the speed of invalid verifications, and found that in all the cases of Table 37, the times are about the same as those of valid verifications.

4.2.3 Asymmetric Identification Schemes

The only identification scheme submitted to NESSIE is GPS. GPS has five parts: the generation of the parameters, the generation of public and private key, the commitment, the answer, and finally the verification. We used the parameters suggested in the submission, which guarantee enough security, namely $|S| = 160$, $|B| = 35$, and $|A| = 275$.[1] We tested the performance of the submitters' code, and obtained the results listed in Tables 38 and 51 .

We also tested the performance of GPS in Java with the submitted Java code. We note that the code is really short with only about 2 KB, including some short comment, but uses external Java libraries. The results are summarised in Table 18. In this table, Machine I is a Pentium 550 MHz with 1 GB of memory, running under Windows 2000, and Machine II is a laptop Pentium 1 GHz with 256 MB of memory and running under Windows 2000.

Table 18. Performance of GPS in Java.

	Machine I	Machine II
Parameters Generation	1.4 ms	1.5 ms
Commitment	9 ms	5 ms
Answer	0.018 ms	0.006 ms
Verification	11 ms	6.2 ms

[1] With the notations of the submission, A/BS must be large enough to guarantee the statistical zero knowledge property, $|A| \geq |S| + |B| + 80$, which is the case with the chosen values.

5. Hardware and Smartcard Implementations

In addition to the implementations in software, the NESSIE project has studied implementations on hardware and smartcards. In this section, we describe the results.

5.1 Hardware Implementations

5.1.1 Previous Results

Table 19 lists known implementations external to the NESSIE project.

For block ciphers, the quoted results apply to encryption without key schedule. FPGA and ASIC implementations are shown together, despite their differences. This table is a collection of known results. These results should not be considered as a fair comparison of the primitives, as each implementation was done with a different methodology, and the implementations were not evaluated by NESSIE.

5.1.2 Block Ciphers

Reprogrammable devices such as Field Programmable Gate Arrays (FPGAs) are highly attractive options for hardware implementations of encryption algorithms. Table 20 summarises results of implementations for the block ciphers MISTY1, Khazad, Rijndael and DES. These results were synthesised with FPGA Express (SYNOPSYS), implemented with XILINX ISE 4 within a VIRTEX1000 and they do not use RAM blocks. Note that the pipeline version of Rijndael cannot fit into a VIRTEX1000. For more information see [547, 591–594]. Table 21 summarises the results of our implementations of Rijndael on VIRTEX3200E.

5.1.3 Estimations for PSEC-KEM

In this section we give time estimations for the PSEC-KEM [238] signature generation and verification algorithm on an EC processor developed at KUL [506].

Algorithm 1, ..., Algorithm 5 are the 5 algorithms related to PSEC-KEM. The total latency to execute Algorithm 1 is basically the latency of the random number generation, which is 14.843 ms. The total latency to execute Algorithm 2 is 29.686 ms. The total latency to execute Algorithm 3 is 14.843 ms. Step 3 of Algorithm 4 takes 14.843 ms and the latency of the other two steps is negligible when compared with EC point multiplication.

Table 19. Non-NESSIE implementations in hardware.

Primitive	Type of Hardware (ASIC)	Speed	More Details
CS-cipher	30 MHz	73 Mbps	estimation; 1mm2
	30 MHz	2000 Mbps	estimation; 15mm2
Hierocrypt-L1	0.14μ 128 MHz	586 Mbps	38.2K gates
IDEA	0.25μ 100 MHz	240 Mbps	720 Mbps in ECB
MISTY1	0.35μ	72 Mbps	7.6K gates
	0.35μ	800 Mbps	50K gates
Triple-DES	0.35μ [24]	407 Mbps	128K gates
Camellia	0.35μ [24]	1170 Mbps	273K gates
	0.35μ [24]	220 Mbps	11K gates
Hierocrypt-3	0.14μ 126 MHz	897 Mbps	81.5K gates
	0.14μ 185 MHz	85 Mbps	26.7K gates
RC6-128	0.5μ	104 Mbps	iterative
	0.5μ	2200 Mbps	pipelined
	0.35μ [24]	204 Mbps	1.6M gates
Rijndael-128	0.5μ	524 Mbps	iterative; 81mm2
	0.5μ	5100 Mbps	pipelined
	0.35μ [24]	1950 Mbps	613K gates
Primitive	Type of Hardware (FPGA)	Speed	More Details
Hierocrypt-L1	11 MHz	44 Mbps	11K logic cells
Camellia	XC4000XLμ [24]	122 Mbps	874 CLBs
	ALTERA 13.1 MHz [545]	240 Mbps	3.0K LE 48k EAB
Hierocrypt-3	7.4 MHz [493]	53 Mbps	23K logic cells
	9 MHz [493]	4.1 Mbps	6.3K logic cells
	8 MHz [545]	115 Mbps	8.6K LE 48k EAB
	ALTERA 11.9 MHz [545]	190 Mbps	9.5K LE 48k EAB
	ALTERA 15.6 MHz [545]	304 Mbps	9.8K LE 48k EAB
	ALTERA 21.7 MHz [545]	397 Mbps	26K logic cells
RC6-128	XCV1000 14 MHz	127 Mbps	feedback
	XCV1000 38 MHz	2400 Mbps	non-feedback
Rijndael-128	XCV1000 14 MHz	300 Mbps	feedback
	XCV1000 32 MHz	1940 Mbps	non-feedback
Primitive	Type of Hardware (DSP)	Speed	More Details
RC6-128	TMS320C6201	18cycles/byte	feedback
	TMS320C6201	13cycles/byte	non-feedback
	TMS320C64x	10cycles/byte	non-feedback

Table 20. FPGA implementations of block ciphers within VIRTEX1000

Cipher	Nbr of slices	Output every (clock edges)	Estimated frequency (MHz)	Throughput (Mbits/s)
MISTY1	6322	1	159	10176
Fast Khazad	8800	1	148	9472
Low area Khazad	7175	1	123	7872
Pipeline Rijndael*	17984	1	–	–
Sequential Rijndael	2257	5/52	127	1563
Pipelined DES	3681	1	175	11200
Sequential DES	189	1/18	176	626
Sequential 3-DES	604	1/18	165	587

* Does not fit into VIRTEX1000

Table 21. Rijndael implementations on VIRTEX3200E

Type	Nbr of LUT	Nbr of reg.	Nbr of slices	RAM blocks	Latency (cycles)	Output every (cycles)	Freq. after Synt. (MHz)	Freq. after Impl. (MHz)
Pipeline	4912	7792	5144	100	42	1	285	112
Pipeline	4272	6832	4032	100	32	1	232	92
Pipeline	3516	3840	2784	100	21	1	208	92
Sequential	1036	1452	866	10	52	5/52	285	147
Sequential	965	1372	739	10	42	4/42	232	135
Sequential	877	932	550	10	31	3/31	208	117
Sequential	877	668	542	10	21	2/21	208	119
Modified sequential	709	413	405	10	20	2/20	192	87

Algorithm 1 KGP-PSEC

```
Inputs: $E$: an elliptic curve subgroup with generator $P$,
    $KDF$: a key derivation functions,
    $hLen$: a nonnegative integer
Outputs: $PK$: PSEC public key, $(E, W, KDF, hLen)$,
    $s$: PSEC private key, a nonnegative integer, $0 \leq s < p$
Generate a random integer $s \in \{0, ..., p-1\}$.
Let $W := sP$.
$PK = (E, W, KDF, hLen)$ and $s$.
```

Algorithm 2 EP-PSEC

```
Inputs: $PK$: PSEC public key,
    $\alpha$: random value, a nonnegative integer, $0 \leq \alpha < p$
Outputs: $Q$: a point on $E$, $C_1$: a point on $E$
Let $Q := \alpha W$.
Let $C_1 := \alpha P$.
Output $(Q, C_1)$.
```

Algorithm 3 DP-PSEC

```
Inputs: $PK$: PSEC public key,
    $C_1$: a point on $E$,
    $s$: PSEC private key, a nonnegative integer, $0 \leq s < p$
Outputs: $Q$: a point on $E$
Let $Q := sC_1$.
$Q$.
```

Algorithm 4 ES-PSEC-KEM-Encrypt

```
Inputs: $PK$: PSEC public key
Outputs: $c_0$ an octet string,
    $k$ an octet string
Let $(\alpha, k, r) := EME - PSEC - KEM - A(PK)$. (See [238] Section 7.1.1.)
Let $(Q, C_1) := EP - PSEC(PK, a)$. (See Algorithm 2.)
Let $c_0 := EME - PSEC - KEM - B(PK, Q, C_1, r)$. (See [238] Section 7.1.2.)
$(c_0, k)$.
```

Algorithm 5 ES-PSEC-KEM-Decrypt

```
Inputs: $PK$: PSEC public key
    $s$: PSEC private key, a nonnegative integer, $0 \leq s < p$
    $c_0$: an octet string
Outputs: $k'$: an octet string
Let $(C_1, c_2, g) := EME - PSEC - KEM - C(PK, c_0)$. (See [238] Section 7.1.3.)
    If the decoding operation returns "invalid", then assert "invalid" and stop.
Let $Q' := DP - PSEC(PK, C_1, s)$. (See [238] Section 5.3.)
Let $(\alpha', k') := EME - PSEC - KEM - D(PK, c_2, g, Q')$. (See [238] Section 7.1.4.)
Check $C_1 = DP - PSEC(PK, P, \alpha')$. (See [238] Section 5.3.) If it holds, output $k'$.
    Otherwise, assert "invalid" and stop.
```

5.1.4 Estimations for ECDSA

In this section we give time estimations for the ECDSA [319] signature generation and verification algorithm on an EC processor developed at KUL [506]. Most of the ECDSA computation time is devoted to modular reduction, modular multiplication, modular addition and modular multiplicative inversion operations. A generic arithmetic processor for these operations has been developed at K.U.Leuven, and the implementation of ECDSA uses this generic arithmetic processor.

If the elliptic curve is defined over $GF(p)$, we use the elliptic curve processor (ECP) developed at KUL [506] for the EC point multiplication and the other modular operations needed for signature generation and verification. If the elliptic curve is defined over $GF(2^m)$, we use Orlando and Paar's processor over $GF(2^m)$ [504] for the EC point multiplication and the ECP for the modular operations. To estimate the costs of implementing SHA-1, we use the data given by Helion Technology Lim. [293]. (For a comparison of implementations of elliptic curve point multiplications see Table 22.)

ECDSA Signature Generation. We give estimations on the ECP for the ECDSA signature generation algorithm. To sign a message m, with domain parameters $D = (q, FR, a, b, G, n, h)$ and associated key pair (d, Q), the operations and the corresponding numbers of clock cycles on the ECP are as follows:

1. Compute $kG = (x_1, y_1)$ and $r = x_1$: If q is an odd prime, $l_1(51l_2 + 66)$, $l_1 = \log_2 k$, $l_2 = \log_2 q$. If $q = 2^m$, m odd prime, $10.5m^2 - 8.5m + 1.5 \lfloor \log_2(m-1) \rfloor m$.
2. Compute $k^{-1} \bmod n$: $9/2l_3^2 + 6l_3$, $l_3 = \log_2 n$.
3. Compute $e = \text{SHA-1}(m)$: 889.
4. Compute $s = k^{-1}(e + dr) \bmod n$: $8l_3 + 9$.

If $l \approx l_1 \approx l_2 \approx l_3$ and q is an odd prime, the total number of clock cycles for the ECDSA signature generation is $55.5l^2 + 80l + 898$. If $m \approx l_2 \approx l_3$ and $q = 2^m$, m odd prime, it is $15m^2 + 5.5m + 1.5 \lfloor \log_2(m-1) \rfloor m + 898$.

ECDSA Signature Verification. The operations for the verification of a signature and the corresponding numbers of clock cycles on the ECP are as follows:

1. Compute $e = \text{SHA-1}(m)$: 889.

Table 22. Area utilisation and throughput of known hardware implementations for elliptic curve point multiplication.

Multiplier	Device	Field	Digit Size	Frequency (MHz)	# Clock cycles per mult.	# Mult. per sec.	Throughput (bits/sec)	# Gates	# CLSs	# FFs	RAM
AMV93	VLSI	$GF(2^{310})$	32	40			444×10^3	11.000			
[8]		$GF(2^{155})$			275862	145	60×10^3				
AMV89	VLSI	$GF(2^{155})$	32	40	4444444	9		11.000			
[6]	M68008 controller										
AMV92 [7]	VLSI	$GF(2^{155})$	32	40	4310	580	200×10^3	11.000			6 Kbytes
SES98 [603]	XC4020XL	$GF(2^{155})$	32	10-15				17.000			
GSS99 [252]	XC4010XL	$GF(2^5)$			126	179856			272		
	XC4013XL	$GF(2^{11})$			825	19230			478		
	XC4028XL	$GF(2^{29})$			7158	1653			962		
	XC4044XL	$GF(2^{53})$			26753	417			1626		
Leung	XCV300-4	$GF(2^{113})$		45	166738	270			645		1808 bits
et al. [398]		$GF(2^{155})$		36	266443	147			784		2480 bits
		$GF(2^{281})$		33	474504	69			1311		4496 bits
OP00 [504]	XCV400E8BG432	$GF(2^{167})$	4	85.7	82212	1818			1627	1745	40 kbits
			8	74.5	45336	2857			2136	1753	40 kbits
			16	76.7	27776	4762			3002	1769	40 kbits
OP01 [505]	XCV1000E8BG680	$GF(p)$ $p = 2^{192} - 2^{64} - 1$		40					11416	5735	140 kbits
S01 [589]	X4000XL	$GF(2^{191})$				85					
JT	Atmel FPSLIC	$GF(2^8)$		10	768	13020			668		
et al. [317]				200		260416					
		$GF(2^{16})$		10	3072	3255			852		
				200		65104					
		$GF(2^{72})$		10	62208	160			2189		
				200		3215					
		$GF(2^{192})$		10	442368	22			4097		
				200		452					

2. Compute $w = s^{-1} \bmod n$: $9/2l_3^2 + 6l_3$.
3. Compute $u_1 = ew \bmod n$ and $u_2 = rw \bmod n$: $6l_3 + 8$.
4. Compute $X = u_1G + u_2Q$: If q is an odd prime, $2l_3(51l_2 + 66) + 42l_2 + 56$. If $q = 2^m$, m odd prime, $21m^2 - 6m + 3\lfloor \log_2(m-1) \rfloor m$.

If $l \approx l_2 \approx l_3$ and q is an odd prime, the total number of clock cycles for the ECDSA signature verification is $106.5l^2 + 186l + 953$. If $m \approx l_3$ and $q = 2^m$, m odd prime, it is $25.5m^2 + 6m + 3\lfloor \log_2(m-1) \rfloor m + 897$.

The area and speed and FPGA resource values of the used circuits for estimations of the ECDSA circuit are given in Table 23.

Table 23. The area and speed values of the used circuits for estimations of the ECDSA circuit.

Operation Block	LUTs	FFs	Block RAMs	Clock Frequency (MHz)
ECP over $GF(p)$	11 227	6 959		91.971
ECP over $GF(2^m)$	1 627	1 745	10	
SHA-1	722		2	95

The times needed for signature generation and verification for ECDSA over $GF(p)$ on an FPGA at 91.971 MHz are given in Table 24.

Table 24. The times needed for signature generation and verification for ECDSA over $GF(p)$ on an FPGA at 91.971 MHz.

Operation	with curve over $GF(2^{192} - 2^{64} - 1)$	with curve over $GF(2^{167})$
Signature Generation	22.42 ms	4.59 ms
Signature Verification	43.09 ms	7.73 ms

5.2 Smartcard Implementations

5.2.1 Non-NESSIE Results

Table 25 lists known non-NESSIE implementations. Results not known are marked by N/A. The results should not be considered as a fair comparison of the primitives, as each implementation was done with a different methodology, and the implementations were not evaluated by NESSIE. In the following we describe our implementations on a 8051 Smartcard.

5.2.2 Implementation of Khazad, MISTY1, and SAFER++

Table 26 summarises the results for our Khazad, MISTY1 and SAFER++ (64-bit block) implementations on a low cost 8-bit smartcard (8051). The RAM and

Table 25. Previous implementations on smartcards.

Primitive	Smartcard	Speed	More Details
CS-cipher	6805	1600 cycles/byte	non optimised
Hierocrypt-L1	Z80/JT6N55	2425 states/byte	RAM/ROM:26/2447bytes
MISTY1	Z80/JT6N55	3185 states/byte	RAM/ROM:44/1598bytes
Triple-DES	6805	6547 cycles/byte	RAM/ROM:213/379bytes
Camellia	8051	638.5 cycles/byte	submission document [24]
RC6-128	6805	2046 cycles/byte	RAM/ROM:200/639bytes
	8051	900 cycles/byte	RAM/ROM:221/596bytes
	ARM	49 cycles/byte	RAM/ROM:192/272bytes
Rijndael-128	6805	895 cycles/byte	RAM/ROM:50/540bytes
	8051	199 cycles/byte	RAM/ROM:49/1016bytes
	ARM	180 cycles/byte	RAM/ROM:16/3900bytes
SHACAL-1	6805	3362 cycles/byte	RAM/ROM:118/379bytes
SHA-1	6805	1051 cycles/byte	RAM/ROM:118/419bytes
GPS	6805	N/A	RAM/ROM:?/300bytes

Table 26. Results of our Khazad, MISTY1, and SAFER++ implementations.

	RAM	ROM (code + tables)	Cycles
Khazad	41(+16)	1227 (705 + 512)	4000
MISTY1	31	2682 (1530 + 1152)	5280
SAFER++	35(+16)	1345 (705 + 640)	3966

ROM are expressed in bytes. The "(+16)" means that 16 bytes must be added if the key is to be kept. The given number of cycles is for the encryption of an 8-byte block and the key schedule.

As one can see, SAFER++ (64-bit block) has comparable performance with Khazad and MISTY1 on a 8051-based CPU, whereas on desktop machines it is slower. This is due to the fact that most of the operations in SAFER++ are performed on 8-bit words, whereas Khazad and MISTY1 can be optimised using 32-bit words. However, the implementation of MISTY1 might be optimised further.

5.2.3 Implementation of (untweaked) ESIGN

SFLASH might have an advantage over other signature schemes only if it is faster on a low cost smartcard without any coprocessor. Two algorithms are candidates to compete with SFLASH: ESIGN and ECDSA. These three algorithms have similar performances on desktop computers, and a rough evaluation shows that they may have similar performances on low cost smartcards. ESIGN has been chosen for this evaluation because there are no IPR problems with optimised implementations of the algorithm.

Low cost smartcards usually have microprocessors either of the Intel 8051 family or of the Motorola 6805 family. The 8051 was chosen as the platform for our comparison. All evaluations are made on the simplest variant of the 8051.

ESIGN and SFLASH first compute the hash of the message. In both cases, this step uses three SHA-1s, so that the times are comparable. Therefore, we can omit this step and implement only the signature algorithm itself.

The implementation characteristics we are interested in are the memory requirements, the speed, and the code size. RAM is clearly the most sensitive resource on a smartcard. A basic 8051 processor has a bank of 128 bytes of internal memory, which is the fastest RAM available to programs. If we need more memory, we have to use external RAM, which is much slower and quite expensive. Therefore, we shall be especially attentive to RAM usage, and try to use as little RAM as possible. The SFLASH parameters have fixed size, thus we do not have decisions to make about their implementation. For ESIGN there is some flexibility concerning the parameters sizes, but we do not use this feature as our implementation is designed to support a fixed key size. We choose the recommended size of 1152 bits for the modulus and $e = 8$ for the exponent. (This is the smallest exponent that fulfills the security requirements, as recommended by the designers in their original (untweaked) submission.)

The first thing to note when implementing ESIGN is that the parameters sizes forces us to use external RAM, for example the modulus will not fit in the internal RAM. The main steps of the ESIGN signature algorithm are two (very different) modular exponentiations. The first modular exponentiation is the computation of $r^8 \bmod n$, where n is a 144-byte modulus and r is a 96-byte value. This step is clearly the costliest one: its computation takes more than a third of the total time, due to the fact that we have to perform three squarings and three modular reductions on large numbers that have to be stored in external memory, as they cannot fit in the on-chip memory of the 8051. External RAM is costlier to handle, as there is only one pointer for accessing data located behind the first 256 bytes of the external RAM. The second main step is the modular exponentiation $r^7 \bmod p$, where p is a 48-byte prime. Here we have to perform two squarings, two multiplications and four modular reductions. The main difference from the previous modular exponentiation — as the values we have to handle are not so big — is that we can use the internal RAM, and only need some smaller external RAM. Thus, this step is not as costly as the first. In total, both modular exponentiations take about half of the complete time.

In Table 27 we give the characteristics of our ESIGN implementation. The performance of this implementation could be improved following a comment by Chung-Huang Yang. The computation of r^7 mod p can be replaced by one modular reduction (r^8 mod n) mod p (where r^8 mod n is already computed) and a multiplication by the modular inverse r^{-1} mod p.

An implementation of ESIGN on an 8-bit H8/300 based smartcard (Hitachi) has been done by Morita, Okamoto and Yang. The claimed results are that a signature generation takes 6.15 s, with a 1152-bit modulus and $e = 1024$, and a CPU running at 5 MHz. For comparison, our implementation would take 12 s to generate a signature with a CPU at the same speed. However, this should not be considered as a fair comparison, as the smartcards are different.

Table 27. Characteristics of the (untweaked) ESIGN and SFLASH (new version) smartcard implementations.

	Key size (bytes)	**Code size** (bytes)	**Memory size** (bytes)	**Signature** (CPU cycles)
ESIGN	336	3000	800	5 100 000
SFLASH (new version)		2000 + 1369	127	7 200 000

5.2.4 Implementation of SFLASH (new version)

Here, we give the details of an implementation of the SFLASH algorithm on a basic 8051 smartcard. We let K denote the field $\mathbb{F}_{2^7}$ and L denote the field $\mathbb{F}_{2^{259}}$, which is an extension field of degree 37 of K. These fields are described in the submission as quotients of the polynomial ring $\mathbb{F}_2[X]$ (resp. $K[X]$) by an ideal generated by an irreducible polynomial of degree 7 (resp. 37).

We implemented the steps 6–9 of the SFLASH signature algorithm, as described in the submission. The initial steps consist of transforming a message into an element of L by using a series of three applications of SHA-1. These steps (1–5) were neglected since the main goal of our implementation was to compare SFLASH (new version) with ESIGN and both have the same initial triple hashing.

The implementation was written in assembly. In the first attempt, we focused on fitting the signature to a basic 8051 without external memory, that is with only the basic 128 bytes of RAM. The significant data of this implementation are as follows:

– Code size: 2K + some extra tables (37 × 37 bytes)
– RAM: 127 bytes (contains registers, stack and 3 elements of L, namely 111 bytes)
– Run time for one signature: approximately 7.2 million cycles

Table 27 contains these characteristics for our SFLASH implementation. An independent implementation by Schlumberger-Sema gives around 223 000 CPU cycles (using external memory). The performance of the implementations by Schlumberger-Sema with and without optimisation of the multiplications using Gray codes is given in Table 28.

Table 28. Performance of optimised implementations of SFLASH (new version), made by the submitters.

CPU Type	**Using Gray Codes**	**CPU Cycles**	**Clock Cycles**	**Time (sec) at 3.57 MHz**	**Time (sec) at 10 MHz**	**RAM (bytes)**	**ROM (bytes)**
Intel 8051	no	319 968	3 839 616	1.075	0.384	473	2.5K
	yes	223 118	2 677 416	0.750	0.268	473	3.1K
Infineon 66	no		821 135	0.230	0.082	473	2.5K
	yes		586 605	0.164	0.059	473	3.1K

We describe the technical part of our implementation: Steps 6–9 of the SFLASH signature algorithm involve two affine maps of L viewed as a K-

vector space of dimension 37. These maps are described in the `GEN_AFF` function, which has addresses of the vectors `VAR_i` and `VAR_j` as input and computes $\texttt{VAR\_j} = \texttt{A}\ \texttt{VAR\_i} + \texttt{B}$, where `A` is a 37×37 matrix and `B` a 37×1 matrix (a column vector), both stored in the ROM.

The delicate part of the signature is the computation of $\texttt{A}^h$ (the modular exponentiation) in L. To do this, we follow suggestion (b) of the SFLASH description (whereas in the Schlumberger-Sema implementation they compute the minimal h, thus saving a lot of computation). To do this exponentiation, we need a linear map, a multiplication map in L (`MULTL`) and a squaring map in L (`SQUAREL`).

Note that we cannot use `MULTL` to square because we only use 128 bytes of RAM: `MULTL` multiplies two elements and writes the result into a third different one, changing two of the three variables of L that are stored in the RAM, whereas `SQUAREL` squares a variable and writes it into a different one, changing only one of the three variables. When performing the modular exponentiation we often need to keep the values of two variables, hence we need `SQUAREL`.

Since the linear map is the same as the affine map `GEN_AFF` with $B = 0$, we only introduced a switch (bit) to indicate whether the linear version will be called.

All these functions rely on a sub-function `MULTK` which performs the multiplication of two bytes (elements of K). This function was made SPA-resistant by using a table of logarithms and a table of exponentials to transform multiplications into sums. Thus we were able to lower the cycle count from 12.4 to 7.2 million with an additional cost of storing two 128-byte arrays in the ROM.

The more dramatic improvements in the cycle count will come from a better implementation of the modular exponentiation. In particular, if we can efficiently compute inverses in L, we can speed up things quite essentially.

On the other hand, the Schlumberger-Sema implementation uses a smaller representative of the exponent h and exploits (as we did) a quasi-periodicity of some loops.

5.2.5 GPS

The GPS submitters claim that GPS (and its tweak) were designed for smartcard applications, in view of speed and code size (only 300 bytes). We obtained very high speeds on smartcards; the card only needs to compute the answer, which on our platforms takes between 1 and 3 ms. If we use a much larger number for $|A|$, the speed for the answer is about the same and only the commitment and the verification are slower. (The commitment may take about 100–300 ms.)

A. Tables

Table 29. Host Computers

Platform	Location	Processor, Speed, Memory	Operating Systems	Compilers
PIII/Linux	TEC	Pentium III, 450 MHz, 256M	Linux 2.4.17	gcc 3.1.1, gcc3.3
	TEC	Pentium III, 600 MHz, 256M	Linux 2.4.17	gcc 2.95.2, gcc 3.0.3, gcc 3.1.1
	TEC	Pentium III, 933 MHz, 256M	Linux 2.4.17	gcc 2.95.2
	RHUL	Pentium III, 665 MHz	Linux 2.2.16	egcs 2.91.66
PIII/MS	TEC	Pentium III, 450 MHz	Win 2000	Visual C 6.0, gcc 2.95.3
	SAG	Pentium III, 850 MHz	Win 2000	gcc 2.95.3
	SAG	Pentium III, 850 MHz	Win 2000	Intel C++ 6.0, Visual C 6.0
	UCL	Pentium III, 850 MHz, 256M	Win 2000	gcc 2.95.3
PI/MMX	TEC	Pentium MMX, 133 MHz, 80M	Linux 2.2.9	gcc 2.7.2.3
Pentium4	TEC	Pentium 4, 1.8 GHz	Linux 2.4.0	gcc 2.95.2
	TEC	Pentium 4, 1.7 GHz	Linux 2.4.12	gcc 2.95.2, gcc 3.1.1
Pentium2	TEC	Pentium II, 350 MHz, 64M	Win 98	Visual C 6.0, gcc 2.95.3
Xeon	TEC	Pentium 4 Xeon, 1680 MHz	Linux 2.4.2	gcc 2.96, egcs 2.91.66
	KUL	Pentium 4 Xeon, 1500 MHz	Linux 2.4.7	gcc 2.96
486	TEC	i486DX, 33 MHz	Linux 2.0.30	gcc 2.7.2.3
Alpha EV6.7	TEC	21264A, 667 MHz	OSF1 V4.0, V5.1	Dec C V6.4, gcc 2.97
Sparc V9	TEC	Ultrasparc IIi, 400 MHz	Sun OS 5.8	SWC 5.1, gcc 3.0.4
	TEC	Ultrasparc IIi, 338 MHz	Sun OS 5.8	SWC 5.1, gcc 3.0.4
	UIB	Ultrasparc IIi, 450 MHz	Sun OS 5.8	SWC 5.2
	ENS	Ultrasparc IIi, 450 MHz, 2G	Sun OS 5.7	SWC 5.0
	ENS	Ultrasparc IIi, 333 MHz	Sun OS 5.8	gcc 3.2.1
	SAG	Ultrasparc IIi, 248 MHz	Sun OS 5.6	gcc 2.95.3
Mac	ENS	G4 (Power PC), 400 MHz	Mac OS 10.0.4	gcc 2.95.2
AMD	TEC	Duron, 1200 MHz, 128M	WinXP	Visual C 7.0 (.NET), gcc 2.95.3
	ENS	AthlonXP 1700+, 1467 MHz, 256M	Linux 2.4.18	gcc 2.96

Table 30. Legacy Block Ciphers Performance Results

Primitive Name	Key Size	Platform											
		PIII Claimed	**PIII/Linux**	**PIII/MS**	**PI/MMX**	**Pentium4**	**Pentium2**	**Xeon**	**486**	**Alpha**	**Sparc V9**	**Mac**	**AMD**
Idea	128	78.7/78.7/–	55/55/450	53/52/395	151/151/1738	100/100/472	59/59/802	100/100/448	133/133/941	55/56/399	88/88/599	53/53/494	55/55/434
Khazad	128	67/67/1206	39/40/765	38/38/823	72/73/2082	48/48/904	45/45/1023	45/45/898	87/87/2092	19/19/368	34/33/799	40/39/981	43/43/837
Khazad-tweak	128	67/67/1206	39/40/766	44/44/822	83/83/2084	45/45/916	45/45/1011	44/45/934	78/78/2182	19/19/373	33/33/808	41/41/996	43/43/861
Misty1	128	37.5/–/170(PII)	47/47/208	42/43/125	102/101/401	59/56/131	48/48/195	56/52/130	69/69/243	36/35/90	33/34/117	62/60/174	52/52/120
Safer++	128	169/160/1333	152/168/1504	143/161/1692	296/256/3570	154/171/3302	145/193/2199	156/171/1731	215/206/2678	85/87/1876	104/117/2173	148/179/2614	150/182/1344
CS-cipher	128	506/506/–(PI)	156/140/2686	166/149/2920	268/329/4994	164/133/5227	166/150/2922	137/131/6513	209/219/4521	115/117/3047	124/156/3350	162/187/4108	196/185/3535
Hierocrypt-L1	128	24.8/24.8/–	43/46/34K	48/51/36K	79/90/47K	69/68/51K	56/58/36K	70/72/55K	118/121/37K	23/28/37K	61/66/33K	67/71/33K	39/42/32K
Nimbus	128	66/–/24665	34/34/12K	36/34/12K	62/58/24K	83/82/31K	36/34/12K	83/82/32K	74/73/30K	19/19/7219	64/68/22K	59/59/19K	39/42/16K
Nush	128	22.5/–/64*	44/38/1415	47/36/1402	111/111/2424	44/26/1103	55/67/1389	44/25/1094	45/50/1223	30/21/942	30/26/2137	36/36/1202	27/25/1659
CAST-128	128		30/30/916	34/34/774	73/73/1486	39/39/1024	38/37/1016	40/39/1007	47/47/2163	36/37/608	32/33/615	37/37/1107	30/31/1005
DES	56		59/59/883	62/62/929	88/88/1624	61/61/816	62/62/932	60/60/819	87/86/1234	37/37/596	54/53/798	60/59/764	54/54/773
Triple-DES	168		154/155/2636	160/160/2744	253/252/4868	158/158/2477	161/160/2755	157/157/2453	504/530/3737	99/99/1814	126/126/2336	156/155/2341	141/141/2312
Kasumi	128		73/74/297	91/92/267	146/146/738	157/148/253	151/150/439	152/161/257	126/126/370	88/88/180	96/97/278	98/99/310	88/85/198
RC5 (12 rounds)	64		19/19/1202	24/24/1494	67/73/3302	30/30/1420	24/24/1504	30/30/1420	39/44/2673	28/25/1109	23/22/1555	19/25/1404	19/19/1236
Skipjack	80		114/120/16	133/150/11	165/166/38	108/107/21	150/152/10	110/107/20	100/100/28	74/78/24	53/54/32	172/133/10	149/144/8.4

The performances given are those for encryption/decryption/key setup, where the encryption and decryption times are measured in cycles/byte and the key setup time is measured in cycles.
* The key schedule claims are for encryption only. Our figures also include decryption subkeys.

Table 31. Normal and High Block Ciphers Performance Results

Primitive	Key	Platform											
Name	**Size**	**PIII Claimed**	**PIII/Linux**	**PIII/MS**	**PI/MMX**	**Pentium4**	**Pentium2**	**Xeon**	**486**	**Alpha**	**Sparc V9**	**Mac**	**AMD**
Camellia	128	36/36/263 (PII)	35/35/313	37/37/334	80/80/825	64/63/453	38/38/426	64/64/444	65/60/561	36/35/287	47/47/352	31/31/383	31/31/274
	192	30.8/30/222 (Asm)	45/45/420	49/48/434	105/105/1166	86/86/546	49/49/533	82/85/537	105/105/788	48/47/407	61/61/539	41/41/578	41/41/386
	256	30.9/30/266 (Asm)	45/45/429	49/49/447	105/105/1207	86/87/555	49/49/554	83/85/559	105/105/796	48/47/453	60/60/542	41/41/578	41/41/393
Camellia 2nd impl	128		35/35/285	37/37/265	82/82/676	72/73/415	37/37/342	72/72/408	62/65/412	44/44/276	50/46/293	32/32/336	32/32/235
	192		45/45/390	49/48/395	107/107/1036	96/96/516	49/49/514	95/95/513	96/101/698	58/58/416	65/60/508	41/41/541	40/41/346
	256		45/45/401	49/48/407	107/107/1075	96/96/539	49/49/526	95/95/533	96/101/701	58/58/441	65/60/514	41/41/541	42/42/361
RC6	128	16/16/– (P-pro)	17/17/1054	24/25/1040	84/85/3094	36/37/2056	24/25/1060	37/36/2050	86/86/2392	27/24/1509	69/68/1432	28/32/1106	17/17/947
(20 rounds)	192		18/17/1175	30/25/1258	84/85/3154	37/37/2141	32/25/1291	37/37/2162	86/86/2904	27/24/1725	117/117/1901	28/32/1115	17/18/1039
	256	16/16/– (P-pro)	18/17/1168	25/25/1262	84/85/3134	37/37/2153	25/25/1291	37/37/2061	86/86/2505	26/24/1727	69/69/1921	28/32/1149	17/18/1032
Safer++	128	40/76/2333	50/63/1333	46/60/1464	104/117/3154	47/58/2918	46/63/1480	47/53/1550	69/77/2035	30/44/1677	35/43/1939	46/74/2266	41/52/1192
	256	57/101/3227	68/88/1805	63/84/1865	144/165/4407	65/83/3998	64/89/1887	65/79/2085	98/109/2820	42/62/2257	49/73/2590	62/104/3226	57/73/1611
Anubis	128	36.8/36.8/4527	37/37/3550	37/37/3727	80/80/9189	35/35/3750	37/37/4389	35/35/3669	55/56/7895	25/25/2990	33/33/4885	36/36/5015	37/37/4687
	160	39.3/39.3/5709	40/40/4519	39/39/4927	85/85/11K	37/37/4845	39/39/5570	37/37/4751	61/64/10K	27/27/3950	35/35/6329	38/38/6540	37/37/6119
	192	41.6/41.6/8008	43/43/5857	42/42/6993	91/91/15K	40/39/6054	42/41/7692	40/40/5949	64/65/12K	29/29/4803	37/37/8088	40/40/8233	42/41/7601
	224	43.8/43.8/9576	45/45/7356	44/44/8546	97/97/18K	41/41/7338	44/44/9251	42/42/7330	67/68/15K	31/31/5715	40/40/9989	41/42/9940	43/44/9387
	256	46.3/46.3/11264	48/48/8876	47/47/10K	102/102/22K	44/44/8902	46/46/11K	44/45/8844	71/71/18K	33/33/8234	42/42/12K	43/43/11K	46/46/11K
	288	48.5/48.5/11291	50/50/10K	49/49/12K	108/108/26K	46/46/10K	49/49/12K	47/47/10K	75/76/21K	35/35/9954	45/45/14K	44/45/13K	48/48/13K
	320	50.8/50.8/15169	53/53/12K	51/51/13K	113/113/30K	48/48/11K	51/51/15K	50/49/11K	82/86/24K	36/37/10K	47/47/16K	47/47/16K	49/49/15K
Grandcru	128	2812/4062/20K*	1250/1518/89K	1294/1686/92K	3121/4177/214K	1634/2274/124K	1731/2024/120K	1675/2136/123K	2171/2646/161K	1028/1278/72K	1574/2166/117K	1798/2129/130K	1612/2106/114K
Hierocrypt-3	128	37/63/370*	51/64/125K	50/56/131K	97/134/169K	54/63/195K	49/56/131K	57/63/202K	135/174/137K	33/39/131K	59/78/118K	74/93/121K	52/61/100K
	192	44/781/386*	60/75/147K	59/67/154K	114/158/199K	63/75/229K	59/66/155K	67/74/239K	158/202/161K	38/46/153K	71/87/140K	85/107/142K	60/72/124K
	256	50/88/468*	69/86/170K	67/76/178K	130/182/230K	73/85/266K	67/76/178K	77/85/276K	183/235/185K	44/52/176K	80/103/161K	96/122/166K	69/83/147K
Noekeon-Dir	128	32/32/0	56/58/31	89/91/28	146/148/81	71/72/31	89/91/34	67/99/31	114/117/57	72/76/28	55/54/15	52/62/11	77/75/27
Noekeon-Ind	128	32/32/525	44/45/634	43/44/732	89/85/1546	64/66/948	52/57/1027	63/66/877	67/68/1049	41/42/635	37/36/546	31/32/515	36/36/620
Nush	128	21/21/112	23/20/2528	28/25/2579	34/30/4385	31/20/1982	28/25/2556	31/20/1978	30/30/2196	23/15/1305	24/21/3900	35/28/2100	22/20/3077
	256	21/21/–	23/20/2545	28/25/2595	34/30/4418	31/20/2000	28/25/2585	31/20/1999	30/30/2229	23/15/1322	24/21/3932	36/28/2110	22/19/3088
Q	128	36/36/500*	54/107/993	59/140/984	96/142/1923	51/231/986	59/144/1022	51/231/1032	80/131/1417	35/165/760	47/87/856	76/187/1319	48/167/954
	256	AMD-ATHLON	60/119/1154	64/156/1104	106/158/2184	58/256/1153	64/160/1193	56/257/1147	89/147/1605	39/184/852	54/98/1014	85/208/1583	54/185/1066
SC2000	128	23.9/25.18/488*	38/41/2424	70/42/2756	66/68/11K	60/60/4661	78/50/3031	64/64/4676	214/237/7713	32/31/3307	29/31/3866	32/32/3026	33/37/2718
	192	27.3/28.7/525*	43/46/2823	81/48/3073	75/77/12K	68/69/5856	90/57/3357	74/75/5798	310/325/8800	36/36/3701	34/35/4862	35/36/3407	45/41/3023
	256	27.5/32.8/526*	43/46/2860	81/48/3089	75/77/12K	68/69/5909	90/57/3386	73/74/5841	278/326/8932	37/37/3730	34/36/4865	35/36/3377	40/41/3044
Mars	128		31/30/2520	36/35/2354	116/112/2882	82/72/3492	43/37/2635	81/74/3306	133/119/2072	33/29/2924	48/49/2863	32/32/2269	29/32/2411
	192		31/30/2530	36/35/2337	117/116/2951	82/72/3480	43/37/2639	81/74/3294	124/119/2087	33/29/2929	48/49/2872	32/32/2290	29/32/2412
	256		31/30/2525	36/35/2336	116/112/2991	82/73/3599	43/37/2654	81/74/3337	138/113/2098	33/29/2934	49/49/2904	32/32/2311	29/32/2429
Rijndael	128		25/26/504	23/23/497	56/56/1461	24/25/689	22/23/612	24/25/711	48/48/1805	17/17/493	21/21/453	29/28/1011	30/31/500
	192		30/31/601	27/27/552	67/67/1713	28/29/820	27/27/714	27/28/839	56/58/2173	20/21/449	25/25/463	32/32/1178	35/36/511
	256		34/35/949	32/32/775	78/78/2159	32/35/1327	31/31/878	32/33/1363	64/64/2571	24/24/663	28/30/750	37/37/1405	39/41/708
Seed	128		45/45/409	51/50/421	78/78/949	63/63/401	51/51/447	69/70/385	76/77/758	40/40/340	34/34/353	36/36/403	41/41/393
Serpent	128		68/80/1577	59/57/1276	99/127/3072	154/172/2235	73/77/1872	153/162/2230	284/303/2952	54/41/1383	57/53/1563	50/51/1398	60/64/1675
	192		68/80/1578	59/57/1272	99/127/3053	155/171/2232	73/76/1874	152/161/2226	277/299/2952	54/41/1381	57/53/1577	50/49/1423	60/65/1681
	256		68/80/1566	59/57/1257	99/127/3041	154/171/2231	73/76/1884	153/161/2235	241/308/3003	54/41/1376	57/53/1575	50/51/1420	60/65/1686
Twofish	128		28/26/10K	28/29/15K	56/55/19K	51/49/8871	31/33/11K	51/48/12K	38/37/14K	20/20/8661	28/28/14K	35/35/14K	30/33/19K
	192		28/26/12K	30/30/17K	56/55/25K	52/49/11K	31/33/13K	52/48/14K	37/40/18K	20/20/12K	28/27/18K	35/35/19K	29/33/26K
	256		28/25/16K	28/29/20K	56/55/32K	52/49/13K	31/33/17K	51/48/16K	38/38/22K	20/20/17K	28/27/22K	35/35/23K	30/33/32K

The performances given are those for encryption/decryption/key setup, where the encryption and decryption times are measured in cycles/byte and the key setup time is measured in cycles.
* The key schedule claims are for encryption only. Our figures also include decryption subkeys.

Table 32. Block ciphers with 160- and 256-bit blocks

Primitive	Key	Block	Platform											
Name	**Size**	**Size**	**PIII Claimed**	**PIII/Linux**	**PIII/MS**	**PI/MMX**	**Pentium4**	**Pentium2**	**Xeon**	**486**	**Alpha**	**Sparc V9**	**Mac**	**AMD**
SHACAL-1	512	160	140/116.5/3200	29/26/75	27/26/447	59/56/198	59/36/177	30/26/64	57/37/173	40/41/880	30/27/44	30/27/136	23/23/337	27/24/54
SHACAL-2	512	256	112.5/115/2800	44/43/737	44/46/748	82/82/1490	51/58/933	44/46/888	51/54/857	59/60/1677	30/31/671	37/37/1169	29/29/809	38/40/597
RC6 (20 rounds)	256	256		59/58/10K	85/83/12K	178/176/31K	132/123/21K	85/83/12K	133/120/21K*	153/154/20K	12/11/2716	98/100/15K	121/120/12K	58/60/12K
Rijndael	192	256		33/34/1785	33/32/1882	77/78/4828	35/35/2512	33/31/2145	35/35/2557	–/–/–	24/24/1818	30/29/2159	–/–/–	50/46/2291
	256	256		33/35/2298	34/33/2362	76/78/6126	35/36/3172	33/32/2705	34/36/3252	–/–/–	24/24/2326	31/32/2584	–/–/–	52/47/2985

The performances given are those for encryption/decryption/key setup, where the encryption and decryption times are measured in cycles/byte and the key setup time is measured in cycles.

* On Alpha, RC6 with 256-bit blocks is more than twice faster than RC6 with 128-bit blocks, although on other processors the contrary holds.

Table 33. Stream Ciphers Performance Results

Primitive	Key	IV	Platform											
Name	Size	Size	PIII Claimed	PIII/Linux	PIII/MS	PI/MMX	Pentium4	Pentium2	Xeon	486	Alpha	Sparc V9	Mac	AMD
BMGL (m=16)	128			1475/2340	1484/2336	2869/5487	1992/2960	1624/2614	1917/2960	3753/3556	1126/1345	1583/2485	1701/2576	1867/2141
2nd impl	128			424/183K	416/184K	896/268K	706/345K	441/194K	668/332K	1734/211K	384/122K	570/224K	–/–	504/172K
Snow	128		6.76/2000	6.7/1179	6.5/1328	15/3101	8.6/1674	8.3/1349	7.7/1707	10/2284	5.6/997	6.7/1770	7.0/1490	8.1/1465
	256		6.75/2000	6.7/1206	6.4/1352	15/3216	8.7/1735	8.3/1382	7.8/1753	10/2302	5.6/1027	6.6/1791	7.0/1497	8.0/1491
Sober-t16	128		42.6/1581	40/1033	44/1188	77/2363	45/1164	62/1640	46/1146	50/1651	38/979	41/1404	42/1402	46/1207
	256		42.6/2062	40/1456	44/1604	77/3173	46/1610	55/2247	46/1526	50/2192	38/1307	41/1894	43/1900	46/1577
Sober-t32	128		24.5/1348	20/838	21/913	36/1617	27/944	26/1119	28/962	38/1749	18/759	22/1260	20/1187	18/846
	256		24.5/2564	20/1025	21/1119	37/1979	27/1140	26/1364	27/1146	38/2131	18/948	22/1508	20/1410	18/1021
Leviathan	128			10/77K	10/79K	23/126K	10/76K	13/80K	10/74K	17/70K	12/25K	14/130K	14/79K	10/98K
	192			10/77K	10/79K	23/127K	10/76K	13/80K	10/74K	17/70K	12/75K	14/129K	14/79K	10/98K
	256			10/77K	10/79K	23/127K	10/76K	13/80K	10/74K	17/70K	12/25K	14/129K	14/79K	10/98K
Lili	128		1200/–	827/46	921/43	1296/145	987/59	918/53	966/59	1040/78	566/34	665/70	717/65	841/44
RC4	128			7.3/2659	8.0/2704	17/6855	20/2568	7.8/2824	20/2879	15/5781	12/3010	13/2667	14/3024	11/2600
BMGL with IV	128	128		1474/2344/20	1504/2326/17	2869/5410/47	2002/2943/20	1618/2461/16	1899/2943/20	3675/3501/38	1124/1328/28	1571/2600/158	1615/2457/20	1861/2126/15
2nd impl	128	128		423/178K/20	414/187K/17	897/268K/47	710/335K/20	445/184K/16	675/333K/20	1690/204K/39	382/122K/29	579/220K/152	–/–/–	501/170K/15
Snow	128	64		6.7/15/612	6.2/12/690	15/41/1612	8.3/19/846	8.3/11/689	7.5/19/857	10/35/1186	5.6/14/520	6.7/51/880	7.1/11/763	8.0/14/638
with IV	256	64		6.7/44/612	6.2/16/689	15/82/1651	8.7/59/846	8.3/32/689	7.6/60/865	10/53/1229	5.6/18/541	6.6/72/889	7.1/20/774	8.0/38/680
Sober-t16	128	128		40/1158/1025	44/1253/1186	77/2636/2113	46/1287/1113	60/1714/1534	46/1299/1078	51/1747/1475	38/1031/911	41/1556/1326	43/1556/1380	46/1281/1127
with IV	256	128		40/1583/1025	44/1675/1186	77/3495/2113	46/1760/1092	55/2343/1530	46/1665/1061	51/2298/1474	38/1371/907	40/1985/1350	43/2031/1385	46/1668/1128
Sober-t32	128	128		20/995/995	21/1121/1176	36/2086/1724	27/1110/1126	26/1398/1200	27/1157/1166	38/1904/1344	18/850/843	23/1389/1400	20/1378/1361	18/925/1007
with IV	256	128		20/1176/994	21/1353/1180	36/2434/1728	27/1330/1129	26/1677/1205	27/1382/1175	39/2281/1333	19/1006/856	23/1651/1423	20/1598/1361	18/1106/1006
Scream-0	128	128		6.6/3603/1124	6.7/3651/1141	12/6917/2224	12/7196/2225	6.7/3684/1143	12/7150/2276	7.9/4653/1400	7.5/2944/991	10/5958/1606	10/6406/1910	6.8/4205/1281
Scream-F	128	128		7.5/3598/1125	7.6/3661/1141	13/6917/2224	12/7300/2221	7.6/3684/1143	13/7139/2266	8.9/4653/1404	10/2944/992	12/5995/1608	11/6419/1900	8.6/4204/1259
Scream-S	128	128		6.9/40K/1193	7.1/40K/1186	12/53K/2217	12/47K/2434	7.1/51K/1231	12/48K/2332	7.9/39K/1431	7.5/47K/995	9.9/28K/1678	8.7/48K/1481	7.2/49K/1383
Seal-3.0	160	32		5.9/158K/17	6.0/172K/12	15/304K/32	5.9/335K/16	6.0/178K/18	5.5/337K/15	7.7/208K/24	4.7/137K/10	3.8/190K/8.7	4.7/135K/6.2	5.8/141K/12

The performances given in the 1st part of the table are those for key stream generation and key setup, where the key stream generation time is measured in cycles/byte and the key setup time is measured in cycles.
For the stream ciphers with initial value (IV) setup in the 2nd part of the table, the performances given are those for key stream generation/key setup/IV setup, where the IV setup time is measured in cycles.

Table 34. Hash Functions Performance Results

Primitive	Hash	Platform											
Name	**Size**	**PIII Claimed**	**PIII/Linux**	**PIII/MS**	**PI/MMX**	**Pentium4**	**Pentium2**	**Xeon**	**486**	**Alpha**	**Sparc V9**	**Mac**	**AMD**
Whirlpool	512	133/–/–	79/59/5534	82/47/5657	190/260/13K	108/130/8559	97/51/6804	112/125/8794	460/122/30K	35/14/2400	91/22/6266	133/82/8818	104/42/7012
2nd impl (MMX)	512		46/63/3199	73/50/5026	177/244/11K	60/132/4495	83/76/5992	60/125/4468	N/A	N/A	N/A	N/A	99/51/6573
MD4	128		4.7/15/440	4.5/12/411	6.9/44/639	6.4/34/600	4.7/13/444	6.4/29/586	4.2/36/601	6.3/11/467	6.8/10/515	7.0/14/518	4.7/8.3/419
MD5	128		7.2/15/602	6.8/12/558	8.9/44/768	9.4/34/799	7.2/12/611	9.4/29/785	5.8/36/704	10.0/11/691	10/10/746	10/14/752	7.5/10/599
RIPEMD	160		18/16/1339	16/14/1207	33/54/2363	26/36/1929	23/15/1651	26/36/1921	27/38/3741	24/11/1697	24/12/1751	20/17/1384	21/12/1493
SHA-0	160		15/16/1011	12/14/896	30/54/1769	23/36/1651	16/15/1008	23/36/1438	51/39/1933	9.8/11/653	12/12/853	12/16/829	12/12/796
SHA-1	160		15/16/1024	13/14/929	27/54/1830	25/20/1497	16/15/1086	25/20/1460	17/39/1500	11/10/745	13/12/955	9.7/16/872	12/12/825
SHA-2	256		39/44/2736	39/20/2643	71/82/4804	40/118/3159	40/34/2860	39/112/3042	60/50/4271	29/10/2021	34/34/2409	29/32/2052	34/39/2369
	384		83/157/11K	74/101/9907	187/432/25K	122/292/16K	84/244/11K	131/280/17K	176/207/24K	16/30/2333	65/118/9018	93/206/12K	71/105/9672
	512		83/156/11K	74/101/9940	187/433/25K	122/292/16K	84/244/11K	131/280/17K	176/207/24K	16/30/2348	65/117/9027	93/206/12K	71/106/9752
Tiger	192		21/16/1542	24/14/1744	41/60/2966	27/27/2015	24/13/1778	26/26/1895	73/39/5137	5.2/10/390	23/14/1551	28/19/1871	20/11/1449
BCHASH-Rijndael	128		49/15/1712	45/14/1588	90/42/3078	77/19/2579	45/13/1605	76/19/2673	214/32/6963	33/8.1/1179	54/82/1997	59/22/2043	47/11/1650
	256		112/44/3775	116/20/3913	241/93/8082	152/118/5254	123/35/4118	150/109/5288	–/–/–	97/10/3204	171/133/5583	–/–/–	149/40/4889

The performances given are those for hash/initialization/initialization+finalization, where the hash time is measured in cycles/byte and the initialization and initialization+finalization times are measured in cycles.

Table 35. Message Authentication Codes Performance Results

Primitive Name	Key Size	MAC Size	Platform											
			PIII Claimed	PIII/Linux	PIII/MS	PI/MMX	Pentium4	Pentium2	Xeon	486	Alpha	Sparc V9	Mac	AMD
Ttmac	160	160	95.8/–/–	21/27/1564	19/21/1304	33/86/2479	40/50/2675	21/30/1582	37/47/2614	23/59/1906	25/16/1691	26/47/1769	21/55/1453	21/14/1526
Umac-16	128	64		6.0/52K/53K	5.8/52K/53K	22/109K/111K	6.2/76K/77K	6.1/53K/54K	6.1/76K/78K	21/72K/75K	4.5/48K/49K	20/72K/74K	12/71K/73K	6.2/74K/75K
Umac-32	128	64	≈ 2/–/–	2.9/54K/55K	2.7/53K/55K	6.3/150K/152K	6.7/76K/77K	2.5/54K/55K	6.6/78K/81K	8.1/94K/98K	1.1/46K/47K	10/74K/76K	6.6/72K/74K	1.9/70K/72K
HMAC-Whirlpool	512	512		72/5078/24K	73/5163/24K	178/12K/59K	98/7161/33K	86/6351/29K	103/7563/35K	462/31K/151K	36/2488/12K	97/6698/31K	136/9409/44K	100/6810/32K
HMAC-MD4	512	128		4.7/638/1961	4.5/640/1881	6.9/1059/3315	6.4/770/2396	4.7/682/2013	6.4/772/2385	4.2/850/2365	6.3/603/2186	6.9/934/2531	6.7/948/2682	4.7/609/1880
HMAC-MD5	512	128		7.3/804/2634	6.8/799/2470	8.9/1186/3822	9.4/963/3186	7.2/853/2705	9.4/968/3175	5.8/955/2766	10/845/3107	10/1150/3440	10/1204/3607	7.4/817/2709
HMAC-RIPE-MD	512	160		18/1571/5608	16/1440/5045	33/2969/10K	27/2083/7656	23/1952/7060	26/2095/7682	29/4318/14K	24/1793/7028	24/2102/10K	19/1872/6018	21/1793/6556
HMAC-SHA-0	512	160		15/1380/4619	12/1184/3905	30/2749/8858	23/1915/7065	16/1454/4955	23/1922/6723	54/6839/39K	9.8/908/3044	12/1267/3926	12/1395/4092	13/1196/3830
HMAC-SHA-1	512	160		15/1346/4697	13/1211/4029	27/2566/8550	25/1987/6702	16/1445/5255	24/1989/6722	17/2004/6872	11/934/3402	13/1361/4558	9.8/1249/3947	12/1162/3947
HMAC-SHA-2	512	256		39/2899/11K	40/2967/11K	71/5460/20K	40/3129/12K	40/3017/11K	39/3186/11K	60/4568/17K	29/2162/8364	34/2919/10K	29/2387/8478	33/2479/9469
	512	384		84/625/34K	75/599/30K	188/1410/75K	124/996/50K	84/660/34K	132/806/53K	178/871/74K	16/193/7261	65/686/28K	92/745/36K	72/488/29K
	512	512		84/614/34K	75/584/30K	188/1399/75K	124/993/51K	84/654/34K	132/801/53K	178/859/73K	16/195/7268	65/681/28K	92/722/37K	72/487/29K
HMAC-Tiger	512	192		21/1758/6526	24/1924/7117	41/3495/12K	28/2307/8489	24/1964/7182	26/2145/7836	73/6421/25K	5.2/500/1776	24/2130/7050	29/2513/8382	20/1654/6037
CBCMAC-Rijndael	128	128		26/616/2056	24/548/1919	58/1633/4833	26/958/2789	24/694/1661	27/960/3407	55/2220/11K	20/523/1636	22/649/2958	30/1135/3151	31/513/1124
CBCMAC-DES	64	64		61/973/2924	62/980/3017	91/1778/4848	72/1009/1932	62/971/1560	69/1036/3198	91/1393/8338	39/628/2153	56/902/2723	61/887/2713	54/747/1283
CBCMAC-Shacal	512	160		31/829/2886	28/846/2745	63/1870/5926	67/751/2473	31/877/1577	74/828/4725	43/1579/5472	37/984/3575	34/1017/3588	25/1008/2774	29/535/1237

The performances given are those for MAC/key setup/key setup+finalization, where the MAC time is measured in cycles/byte and the key setup and key setup+finalization times are measured in cycles.

Table 36. Asymmetric Encryption Performance Results

Primitive	Key	Platform											
Name	**Size**	**PIII Claimed**	**PIII/Linux**	**PIII/MS**	**PI/MMX**	**Pentium4**	**Pentium2**	**Xeon**	**486**	**Alpha**	**Sparc V9**	**Mac**	**AMD**
EPOC-2	1152		30M/7989K/366M	30M/8024K/569M	47M/10M/471M	83M/18M/1150M	25M/6235K/389M	23M/5866K/357M	–/–/–	–/–/–	–/–/–	–/–/–	18M/4654K/299M
RSA-OAEP	1024		2026K/42M/1654M	2583K/52M/2214M	4876K/97M/3916M	3028K/62M/2616M	2572K/53M/1935M	2981K/61M/2587M	4638K/90M/3776M	–/–/–	3872K/77M/3106M	2255K/46M/1956M	2289K/48M/2024M
ACE encrypt			57M/25M/3194M	–/–/–	52M/22M/2783M	63M/27M/3416M	–/–/–	51M/20M/3237M	90M/38M/4724M	–/–/–	268M/113M/13G	–/–/–	33M/14M/1748M
EPOC1			6312K/11M/46M	6433K/11M/47M	10M/17M/75M	18M/30M/123M	7599K/9853K/59M	18M/30M/122M	19M/32M/128M	1343K/2376K/10M	66M/106M/421M	–/–/–	4155K/6930K/31M
EPOC2			8254K/11M/46M	8477K/12M/48M	12M/19M/75M	24M/32M/123M	8538K/10M/47M	23M/32M/122M	23M/34M/128M	2112K/2461K/10M	75M/114M/418M	–/–/–	5823K/7270K/31M
EPOC3			8279K/4882K/46M	8554K/5158K/48M	12M/7689K/75M	23M/11M/123M	7981K/3895K/43M	23M/11M/121M	23M/12M/128M	2089K/1067K/10M	73M/40M/418M	–/–/–	5808K/2728K/31M
PSEC1			7952K/6322K/2306	9703K/7341K/–	9836K/8338K/2169	13M/10M/4776	19M/8373K/5295	12M/10M/4696	12M/10M/2268	7136K/6280K/3612	22M/20M/5017	–/–/–	7480K/6087K/2993
PSEC2			7703K/6176K/2380	–/–/–	9646K/8347K/2162	12M/10M/4710	10M/8197K/4534	12M/9959K/4538	12M/11M/2326	7040K/6197K/3742	22M/20M/5061	–/–/–	7301K/5757K/2992
PSEC3			7952K/76K/2370	–/–/–	9864K/84K/2154	13M/108K/4716	10M/72K/4546	13M/111K/4471	12M/121K/2228	7147K/40K/3838	22M/73K/4980	–/–/–	7495K/76K/2918

The performances given are those for encryption/decryption/key generation. All of them are measured in cycles.

Table 37. Signature Performance Results

Primitive	Key	Platform											
Name	**Size**	**PIII Claimed**	**PIII/Linux**	**PIII/MS**	**PI/MMX**	**Pentium4**	**Pentium2**	**Xeon**	**486**	**Alpha**	**Sparc V9**	**Mac**	**AMD**
ECDSA-GF(2^p)	160		4775K/6085K/4669K	4639K/5950K/4494K	7838K/9947K/7692K	6100K/7862K/5936K	4620K/5950K/4536K	6261K/8098K/6080K	8543K/11M/8481K	2523K/3212K/2466K	6745K/8680K/6656K	5480K/7000K/5600K	4025K/5251K/3909K
ECDSA-GF(2^{163})	163		5061K/6809K/4852K	5051K/6652K/4932K	8223K/10M/7966K	5922K/7981K/5669K	5162K/6912K/4970K	6161K/8217K/5840K	7507K/10M/7317K	3590K/4791K/3465K	4811K/6435K/4631K	4880K/6560K/4711K	4578K/6153K/4363K
KCDSA-GF(2^p)	160		4724K/6085K/4688K	4596K/5869K/4524K	7785K/9894K/7737K	6024K/7769K/5984K	4410K/5740K/4582K	6139K/7993K/6129K	8533K/10M/8448K	2512K/3184K/2482K	6770K/8630K/6744K	5640K/7280K/5651K	4013K/5149K/3933K
KCDSA-GF(2^{163})	163		5024K/6692K/4888K	5034K/6716K/4962K	8157K/10M/7999K	5939K/7854K/5707K	5141K/6912K/5020K	6094K/8068K/5892K	7431K/10M/7326K	3562K/4674K/3470K	4680K/6300K/4617K	4800K/6280K/4776K	4530K/6068K/4400K
Esign	1152		586K/167K/368M	604K/169K/460M	847K/319K/799M	996K/253K/545M	525K/190K/400M	527K/259K/483M	1221K/366K/863M	416K/85K/174M	1035K/300K/536M	890K/250K/688M	405K/148K/293M
RSA-PSS	1024		42M/2029K/1334M	52M/2562K/1443M	97M/4894K/3833M	62M/3019K/1937M	53M/2562K/1438M	62M/2979K/1929M	90M/4623K/–	–/–/–	77M/3520K/2994M	46M/2240K/1825M	48M/2288K/1353M
SFLASHv2			1385K/328K/731M	–/–/–	5106K/765K/2929M	–/–/–	1761K/344K/734M	1299K/269K/737M	–/–/–	1353K/143K/540M	1885K/312K/928M	1860K/289K/848M	2030K/385K/1020M
ACE sign			24M/19M/7313M	–/–/–	22M/16M/7314M	26M/20M/9645M	–/–/–	25M/19M/8790M	34M/28M/9710M	–/–/–	95M/84M/25G	–/–/–	14M/7113K/5485M
ECDSA			952K/2538K/805K	1211K/3207K/1023K	1667K/4589K/1491K	1971K/5415K/1758K	1236K/3251K/1042K	1463K/3870K/1241K	1446K/3758K/1194K	–/–/386K	809K/2083K/670K	1164K/3024K/948K	997K/2663K/841K
Esign (e=8)	1152		2046K/370K/104M	2099K/370K/113M	2362K/577K/169M	4434K/936K/269M	2074K/296K/95M	4107K/938K/265M	2861K/997K/279M	978K/113K/32M	5069K/2965K/872M	–/–/–	1818K/275K/74M
FLASH			2050K/291K/1092M	2721K/308K/1395M	6362K/973K/3220M	2406K/280K/1116M	7067K/–/2106M	2294K/244K/1129M	5425K/753K/3021M	1506K/197K/732M	3132K/394K/1058M	2416K/426K/854M	3296K/412K/1456M
QUARTZ			5500M/121K/1555M	6104M/115K/1826M	12G/315K/4444M	6261M/144K/3167M	12G/220K/4257M	7214M/136K/2823M	19G/347K/5926M	2855M/129K/1111M	5760M/191K/4703M	–/–/–	6406M/123K/2028M
SFLASH			1700K/195K/786M	2303K/259K/952M	5239K/640K/3676M	2387K/395K/1192M	2330K/269K/943M	2308K/337K/1132M	4183K/489K/2799M	1309K/164K/465M	2767K/567K/3753M	1800K/241K/735M	2910K/212K/1016M
RSA-PSS			56M/981K/2182M	62M/1208K/2388M	106M/2089K/4005M	82M/1587K/3206M	115M/1255K/3412M	76M/697K/2967M	95M/1924K/3623M	40M/779K/1558M	132M/2424K/4968M	48M/968K/1862M	53M/1171K/2078M

The performances given are those for signature/verification/key generation. All of them are measured in cycles.

Table 38. Identification Performance Results

Primitive Name	Key Size	Platform											
		PIII Claimed	**PIII/Linux**	**PIII/MS**	**PI/MMX**	**Pentium4**	**Pentium2**	**Xeon**	**486**	**Alpha**	**Sparc V9**	**Mac**	**AMD**
GPS			312M	326M	439M	757M	249M	236M	–	71M	–	–	183M
			2864K	2860K	5333K	9201K	2899K	2676K	–	811K	–	–	2138K
			4691K	4707K	8541K	14M	4571K	4202K	–	1056K	–	–	3340K
			328	297	389	491	309	343	–	144	–	–	241
			5475K	5482K	9699K	16M	5194K	4755K	–	1200K	–	–	3796K

The performances given are those for parameter generation/key generation/commitment/answer/verification. All of them are measured in cycles.

Table 39. Legacy Block Ciphers Performance Results by Processor and Compiler

Primitive	Key	Platform, Processor and Compiler									
		PIII/Linux	PIII/Linux	PIII/Linux	PIII/Linux	PIII/Linux	PIII/Linux	PIII/Linux	PIII/Win	PIII/Win	PIII/Win
Name	Size	Coppermine	Coppermine	Coppermine	Tualatin	Katmai	Katmai	Coppermine			
		gcc 2.95.2	gcc 3.1.1	gcc 3.0.3	gcc 2.96	gcc 3.1.1	gcc 3.3	gcc-egcs	Visual C 6.0	Intel C	gcc 2.95.3
Idea	128	61/61/878	57/56/450	58/58/492	60/60/1231	56/56/669	55/55/690	67/68/811	64/66/585	53/52/395	59/59/851
Khazad	128	45/46/834	39/40/765	40/40/807	43/44/813	40/41/814	40/41/829	46/46/884	38/38/823	41/41/853	45/45/823
Khazad-tweak	128	49/49/841	39/40/766	42/42/808	44/44/880	40/41/906	40/41/917	46/46/978	46/46/825	44/44/892	45/45/822
Misty1	128	47/48/257	47/47/209	47/47/210	50/49/296	48/48/208	48/48/243	49/48/340	61/62/195	42/43/125	47/47/243
Safer++	128	152/196/1835	178/169/1504	170/169/1505	279/316/1979	177/168/1526	181/169/1943	184/245/1952	143/161/2600	185/215/1692	158/193/2050
Cs-cipher	128	163/150/2851	184/191/2691	156/140/2873	392/434/12K	183/190/2686	294/193/2779	172/155/4837	414/432/5077	411/433/3208	166/149/2920
Hierocrypt-L1	128	54/57/36K	44/47/34K	48/51/36K	52/54/48K	44/47/34K	43/46/34K	53/54/66K	65/70/48K	48/51/52K	54/58/36K
Nimbus	128	34/34/12K	38/41/13K	38/42/13K	37/44/14K	38/41/13K	36/40/13K	37/36/13K	42/41/16K	41/42/17K	36/34/12K
Nush	128	95/66/1438	48/46/1422	44/38/1476	116/78/1497	48/46/1417	46/47/1415	99/73/1452	55/70/1402	47/36/1472	97/67/1489
CAST-128	128	37/38/1021	31/30/933	31/32/916	34/38/1085	30/30/938	31/30/943	52/58/992	48/48/1081	34/34/774	38/37/1009
DES	56	60/60/922	59/59/886	59/59/905	62/62/1024	59/59/883	59/60/910	61/61/969	72/72/952	68/68/960	62/62/929
Triple-DES	168	161/160/2753	155/156/2656	158/158/2726	162/162/3107	154/155/2636	157/156/2728	182/181/2899	186/187/2862	182/183/2858	160/160/2744
Kasumi	128	90/88/410	74/75/339	73/74/297	93/90/406	124/125/345	124/126/333	96/95/391	143/141/565	97/97/267	91/92/436
RC5 (12 Rounds)	64	24/24/1508	20/19/1235	20/19/1308	21/21/2604	19/19/1202	35/37/1394	35/38/1605	33/34/1711	35/31/1494	24/24/1500
Skipjack	80	160/149/18	115/121/16	121/120/16	166/183/22	114/120/16	115/121/17	191/197/18	256/409/11	133/150/13	150/152/17

		PII/Win	PII/Win	PIV/Linux	PIV/Linux	Xeon	Xeon	Xeon	AMD Duron	AMD Duron	AMD Athlon
				Northwood	Northwood	Model 1	Model 0	Model 0	Win XP	Win XP	Linux
		Visual C 6.0	gcc 2.95.3	gcc 2.95.2	gcc 3.1.1	gcc 3.2	egcs 2.91.66	gcc 2.96	VC 7.0 (.NET)	gcc 2.95.3	gcc 2.96
Idea	128	68/67/802	59/59/1061	105/105/472	100/100/478	100/100/459	118/120/521	103/102/448	67/67/517	57/58/435	55/55/434
Khazad	128	47/47/1098	45/45/1023	52/52/1102	48/48/904	48/48/898	54/54/1175	45/45/963	48/46/837	45/45/866	43/43/904
Khazad-tweak	128	49/48/1105	45/45/1011	50/50/1089	45/45/916	46/46/934	50/50/1129	44/45/1026	51/51/889	45/45/861	43/43/948
Misty1	128	72/68/195	48/48/244	59/56/147	62/56/131	58/53/130	61/60/148	56/52/143	59/60/120	52/52/174	55/53/151
Safer++	128	145/203/2578	158/193/2199	198/278/3896	154/171/3302	156/171/1731	178/345/4598	168/187/3886	150/233/2322	187/208/1999	182/182/1344
Cs-cipher	128	642/448/5114	166/150/2922	323/133/5227	164/161/6312	143/169/7454	231/137/8629	137/131/6513	239/239/4522	248/189/3841	196/185/3535
Hierocrypt-L1	128	65/68/48K	56/58/36K	69/68/51K	69/69/57K	70/72/57K	76/72/61K	76/72/55K	39/42/44K	48/51/34K	43/47/32K
Nimbus	128	41/40/16K	36/34/12K	83/82/31K	93/99/34K	92/93/33K	83/82/32K	89/98/33K	39/42/20K	45/46/16K	45/51/16K
Nush	128	55/71/1389	97/67/1435	44/26/1113	44/26/1103	44/25/1094	48/29/1276	47/29/1118	34/27/1707	34/30/1716	27/25/1659
CAST-128	128	48/47/1107	38/37/1016	42/42/1054	39/39/1024	40/39/1007	88/93/1194	43/42/1093	36/35/1073	33/33/1192	30/31/1005
DES	56	73/71/943	62/62/932	73/73/879	61/61/816	60/60/819	73/73/954	61/61/829	68/69/856	54/54/782	55/54/773
Triple-DES	168	184/184/2822	161/160/2755	192/192/2631	158/158/2477	158/158/2453	387/382/2817	157/157/2517	189/181/2565	141/141/2312	143/144/2337
Kasumi	128	211/209/544	151/150/439	162/150/396	157/148/253	192/163/257	212/213/383	152/161/260	123/122/246	132/131/319	88/85/198
RC5 (12 Rounds)	64	32/33/1778	24/24/1504	32/31/2376	30/30/1420	30/30/1420	38/54/1863	31/32/3399	26/27/2203	26/24/1236	19/19/1477
Skipjack	80	354/480/10	150/152/19	243/327/23	108/107/21	110/107/20	240/267/22	159/176/25	560/864/8.4	182/173/15	149/144/14

		Alpha	Alpha	Sun/Sparc V9	Sun/Sparc	Sun/Sparc V9	Sun/Sparc V9	Sun/Sparc V9	Sun/Sparc V9	Sun	Sun
				450MHz	248MHz	338MHz	338MHz	400MHz	400MHz	450MHz	333MHz
		cc	gcc 2.97	gcc 3.0.4+3.2.1	gcc 2.95.3	cc	gcc 3.0.4	cc	gcc 3.0.4	cc	gcc 3.2.1
Idea	128	55/56/647	57/57/399	88/88/600	88/88/600	185/188/697	90/90/619	179/180/658	88/88/599	186/189/678	91/91/672
Khazad	128	25/25/384	19/19/368	35/36/971	39/39/1096	35/35/832	35/35/983	34/33/799	34/35/957	36/36/921	36/37/1002
Khazad-tweak	128	25/26/385	19/19/373	36/36/1019	39/39/1132	35/35/834	35/35/1034	33/33/808	34/35/991	36/36/921	39/40/1148
Misty1	128	47/49/135	36/35/90	51/52/159	56/56/150	35/36/125	54/55/151	33/34/117	53/53/146	44/47/121	56/56/158
Safer++	128	163/302/1968	85/87/1876	108/117/3013	116/142/3141	151/178/2334	111/120/3128	147/171/2253	104/117/3004	116/158/2173	106/120/3081
Cs-cipher	128	374/362/3887	115/117/3047	139/160/4054	124/156/4400	131/168/4456	145/166/4168	126/161/4275	141/162/4061	132/165/3350	136/157/4666
Hierocrypt-L1	128	23/28/39K	35/34/37K	63/68/33K	64/72/34K	64/73/42K	64/69/35K	62/70/41K	61/66/33K	62/66/38K	65/73/33K
Nimbus	128	19/19/7219	36/36/13K	77/76/25K	78/76/25K	66/72/24K	79/78/26K	64/68/22K	77/76/25K	65/71/23K	79/78/25K
Nush	128	31/27/942	30/21/1212	31/26/2177	30/27/2137	38/34/2865	31/27/2230	37/32/2744	31/26/2167	37/30/2813	33/29/2342
CAST-128	128	43/42/608	36/37/662	38/38/992	38/38/1067	34/35/658	40/41/1305	32/33/615	38/38/1217	34/34/674	39/40/1030
DES	56	37/37/603	39/39/596	68/68/888	63/63/899	55/56/842	65/65/913	54/54/819	63/63/897	56/53/798	64/65/904
Triple-DES	168	99/99/1873	101/101/1814	170/168/2641	171/171/2698	131/130/2527	182/181/2731	126/126/2455	172/171/2654	140/140/2336	176/175/2735
Kasumi	128	104/104/180	88/88/253	97/97/289	100/98/278	99/101/381	100/101/298	96/98/365	97/98/289	106/106/356	107/108/344
RC5 (12 Rounds)	64	28/27/1109	28/25/1392	23/23/1709	24/24/1555	28/31/1613	24/23/1772	27/30/1574	23/22/1726	30/30/1663	25/25/2078
Skipjack	80	396/421/32	74/78/24	55/55/32	60/58/32	90/92/117	55/56/33	88/90/105	53/54/32	127/128/119	55/57/32

The performances given are those for encryption/decryption/key setup, where the encryption and decryption times are measured in cycles/byte and the key setup time is measured in cycles.

Table 40. Normal and High Block Ciphers Performance Results by Processor and Compiler

Primitive	Key	Platform, Processor and Compiler									
Name	**Size**	**PIII/Linux Coppermine gcc 2.95.2**	**PIII/Linux Coppermine gcc 3.1.1**	**PIII/Linux Coppermine gcc 3.0.3**	**PIII/Linux Tualatin gcc 2.96**	**PIII/Linux Katmai gcc 3.1.1**	**PIII/Linux Katmai gcc 3.3**	**PIII/Linux Coppermine gcc-egcs**	**PIII/Win Visual C 6.0**	**PIII/Win Intel C**	**PIII/Win gcc 2.95.3**
Camellia	128	38/37/339	35/35/336	35/35/313	35/35/318	35/35/415	35/35/403	40/40/366	65/65/411	47/47/334	37/37/337
	192	49/48/489	45/45/462	47/47/447	46/46/420	45/45/556	45/45/558	51/51/487	84/84/605	59/59/434	49/48/445
	256	49/48/497	45/45/472	47/47/468	46/46/429	45/45/571	45/45/574	52/51/496	84/84/629	59/59/447	49/49/454
Camellia 2nd impl	128	38/38/327	35/35/304	35/35/285	36/36/297	35/35/342	35/36/343	40/40/337	64/65/387	45/45/265	37/37/298
	192	48/49/471	45/45/446	45/45/390	46/47/410	45/45/527	46/45/523	53/53/465	84/85/594	55/56/395	49/48/426
	256	49/49/479	45/46/454	46/46/401	47/47/415	45/45/538	45/45/542	53/52/473	84/85/616	55/56/407	49/48/435
RC6 (20 rounds)	128	31/25/1264	18/17/1054	18/17/1109	26/22/1162	17/17/1063	31/29/1524	33/34/2030	37/33/2073	29/29/1740	24/25/1040
	192	37/25/1271	18/17/1176	18/17/1175	26/22/1333	18/17/1187	32/29/1537	33/34/2034	37/34/2076	30/30/1753	32/25/1258
	256	25/25/1270	18/17/1168	18/17/1184	25/22/1343	18/17/1180	32/29/1548	33/34/2059	36/34/2066	30/30/1758	25/25/1262
Safer++	128	52/65/1606	50/64/1341	51/63/1333	50/64/1750	50/64/1346	50/65/1717	51/66/1732	47/60/2258	46/78/1481	50/63/1464
	256	72/91/1823	69/91/1805	70/89/1815	68/88/2419	69/91/1808	69/90/2344	70/98/2241	65/84/3121	63/110/2047	68/89/1865
Anubis	128	38/38/4454	38/38/3611	40/40/3783	41/40/3845	37/37/3615	37/37/3550	40/40/4501	37/37/4325	43/43/3727	37/37/4527
	160	41/40/5775	41/40/4663	43/43/4801	43/43/5040	41/40/4668	40/40/4519	43/43/5821	40/40/5530	46/46/4927	39/39/5852
	192	43/43/7621	43/43/6043	45/45/6125	46/46/6955	43/43/6050	43/43/5857	46/46/7657	42/42/7843	49/49/6993	42/42/7630
	224	45/45/9631	46/46/7558	48/48/7873	49/49/8732	45/45/7524	45/45/7356	48/48/9445	44/44/9508	52/52/8546	44/44/9134
	256	48/48/11K	48/49/9076	51/51/9245	51/51/10K	48/48/9139	48/48/8876	51/51/11K	47/47/11K	55/55/10K	47/47/11K
	288	51/50/13K	52/52/10K	54/54/11K	54/54/12K	51/51/10K	50/50/10K	54/54/13K	49/49/13K	58/58/12K	49/49/12K
	320	53/53/15K	54/54/12K	57/57/12K	57/57/13K	53/53/12K	53/53/12K	57/56/15K	52/52/15K	61/61/13K	51/51/15K
Grandcru	128	1628/1999/113K	1255/1526/90K	1462/1721/105K	2234/2645/160K	1250/1518/89K	1365/1687/98K	2133/2743/150K	1856/2472/131K	1294/1686/92K	1732/2019/120K
Hierocrypt-3	128	58/78/130K	52/65/126K	51/64/130K	53/71/172K	51/65/125K	52/65/126K	57/75/240K	50/56/173K	52/61/191K	53/68/131K
	192	67/92/154K	60/76/149K	60/75/154K	64/84/204K	60/76/149K	61/76/147K	67/87/284K	59/67/204K	62/73/225K	62/80/154K
	256	77/106/178K	69/88/171K	69/86/178K	72/97/235K	69/87/170K	69/88/171K	77/102/328K	67/76/236K	70/83/260K	71/92/178K
Noekeon-Dir	128	88/91/31	60/59/34	92/95/34	116/115/35	59/59/34	56/58/34	91/92/31	137/137/58	116/117/28	89/91/34
Noekeon-Ind	128	53/56/1033	44/46/660	44/45/634	45/46/852	44/45/657	45/46/668	89/63/1172	75/75/1035	43/44/732	52/57/1016
Nush	128	28/25/2554	23/20/2559	23/21/2588	29/25/2606	23/20/2549	23/20/2568	28/25/2528	28/25/2579	30/25/2585	28/25/2584
	256	28/25/2584	23/20/2592	23/21/2634	29/25/2680	23/20/2573	23/20/2616	28/25/2545	28/25/2605	30/25/2595	28/25/2597
Q	128	59/142/993	54/112/1008	55/139/1022	69/157/1036	54/111/1010	54/107/1010	64/146/1071	61/146/1109	59/140/984	59/145/1026
	256	64/159/1174	61/124/1161	60/155/1155	77/181/1155	60/123/1158	61/119/1154	71/164/1229	67/163/1213	65/156/1104	64/162/1164
SC2000	128	72/42/2732	50/49/2424	38/41/2740	60/55/6738	59/58/2672	60/59/2534	77/50/3359	133/134/6988	138/138/3972	70/42/2756
	192	83/47/3047	56/57/3315	43/46/3055	70/62/7637	67/68/3582	68/68/2823	88/57/3766	152/153/7908	159/159/4443	81/48/3073
	256	83/47/3059	56/57/3327	43/46/3071	70/62/7660	67/67/3599	68/68/2860	88/57/3795	152/153/7955	159/159/4483	81/48/3089
Mars	128	44/37/2673	31/30/2536	31/34/2527	35/38/2571	31/30/2539	45/39/2547	51/47/2520	45/42/2902	36/35/2354	43/37/2628
	192	44/37/2658	31/30/2551	31/34/2530	36/38/2582	31/30/2550	45/39/2555	51/47/2530	45/42/2920	36/35/2337	43/37/2633
	256	44/37/2670	31/30/2549	31/34/2525	35/38/2571	31/30/2553	45/39/2548	52/47/2533	45/42/2929	36/35/2336	43/37/2644
Rijndael	128	28/29/902	26/26/504	25/26/523	27/27/792	26/26/621	26/26/630	28/30/986	23/23/497	24/29/546	28/28/910
	192	34/34/989	31/31/601	30/31/647	32/32/916	31/31/766	31/31/770	33/36/1168	27/27/552	29/34/587	32/33/1018
	256	38/39/1157	35/36/949	34/35/960	37/37/1153	35/35/1067	36/36/1077	39/41/1350	32/32/780	33/39/775	37/37/1199
Seed	128	50/50/441	45/45/409	48/48/417	50/50/470	45/45/423	45/46/422	55/55/492	57/57/446	51/50/421	51/51/465
Serpent	128	73/95/2027	70/81/1577	70/81/1925	84/82/2241	70/80/1878	70/80/1818	68/81/2307	84/77/1908	59/57/1276	73/86/1904
	192	73/95/2039	70/80/1578	70/81/1924	84/82/2223	70/80/1892	70/80/1820	68/81/2321	84/77/1931	59/57/1272	73/86/1919
	256	73/95/2032	70/80/1566	70/81/1917	84/82/2218	70/80/1880	70/81/1830	68/81/2306	84/77/1907	59/57/1257	73/86/1911
Twofish	128	36/36/17K	29/26/16K	28/29/17K	30/30/13K	29/26/11K	29/26/10K	35/36/23K	31/30/16K	28/29/15K	33/36/16K
	192	36/36/19K	29/26/18K	28/29/19K	30/30/18K	29/26/13K	29/26/12K	35/36/27K	31/30/17K	30/30/18K	33/36/19K
	256	36/36/22K	29/26/20K	28/29/22K	30/30/19K	29/25/17K	29/26/16K	35/36/33K	32/30/20K	28/29/20K	33/36/23K

The performances given are those for encryption/decryption/key setup, where the encryption and decryption times are measured in cycles/byte and the key setup time is measured in cycles.

Table 41. Normal and High Block Ciphers Performance Results by Processor and Compiler (continued)

Primitive	Key	Platform, Processor and Compiler									
		PII/Win	PII/Win	PIV/Linux	PIV/Linux	Xeon	Xeon	Xeon	AMD Duron	AMD Duron	AMD Athlon
Name	Size			Northwood	Northwood	Model 1	Model 0	Model 0	Win XP	Win XP	Linux
		Visual C 6.0	gcc 2.95.3	gcc 2.95.2	gcc 3.1.1	gcc 3.2	egcs 2.91.66	gcc 2.96	VC 7.0 (.NET)	gcc 2.95.3	gcc 2.96
Camellia	128	64/64/479	38/38/426	71/72/535	64/63/453	64/64/444	71/71/626	69/68/580	68/68/330	34/35/274	31/31/290
	192	84/84/672	49/49/533	96/96/815	86/86/546	82/85/537	95/95/819	90/89/633	90/91/456	44/44/386	41/41/392
	256	84/84/697	49/49/554	96/96/830	86/87/555	83/85/559	95/95/837	89/89/673	90/90/463	44/45/398	41/41/393
Camellia 2nd impl	128	65/65/405	37/37/342	72/73/525	88/90/415	88/90/408	72/72/590	95/93/432	68/69/316	35/35/265	32/32/235
	192	84/84/646	49/49/514	96/96/732	116/118/516	117/115/513	95/95/796	123/128/612	90/92/448	45/45/371	40/41/346
	256	84/84/676	49/49/526	96/96/790	114/116/539	113/115/533	95/95/799	125/131/656	90/92/455	45/45/377	42/42/361
RC6	128	36/33/2070	24/25/1060	48/45/2136	36/37/2056	37/37/2050	101/85/3740	37/36/2053	32/31/1790	22/23/947	17/17/1034
(20 rounds)	192	37/34/2029	32/25/1291	103/66/2164	37/37/2141	37/37/2188	102/86/4071	37/37/2162	32/31/1792	28/23/1056	17/18/1039
	256	35/33/1984	25/25/1291	39/45/2153	37/37/2206	37/37/2157	92/85/3885	37/37/2061	32/31/1800	23/23/1057	17/18/1032
Safer++	128	46/77/2240	50/63/1480	54/59/3249	47/58/2918	47/58/1550	48/61/4083	52/53/3420	52/75/2043	45/52/1608	41/65/1192
	256	64/108/3115	68/89/1887	74/84/4154	65/83/3998	65/83/2085	66/94/5713	71/79/4773	68/104/2835	63/73/1686	57/89/1611
Anubis	128	37/37/4389	37/37/4575	35/35/4998	36/36/3750	37/37/3793	52/52/5176	35/35/3669	46/46/5576	37/37/5552	37/37/4687
	160	39/39/5570	39/39/5899	37/37/6282	38/39/4845	39/39/5364	56/56/6730	37/37/4751	52/51/7554	37/37/7095	38/39/6119
	192	42/41/7859	42/42/7692	40/39/7695	41/41/6054	41/41/6170	59/59/7787	40/40/5949	61/63/11K	42/41/9499	42/42/7601
	224	44/44/9556	44/44/9251	41/41/9627	43/44/7338	44/44/7469	61/62/10K	42/42/7330	66/63/12K	46/49/12K	43/44/9387
	256	46/46/11K	47/47/11K	44/44/11K	46/46/8902	46/47/8844	67/67/11K	44/45/8919	71/72/16K	48/47/15K	46/46/11K
	288	49/49/13K	49/49/12K	46/46/12K	49/48/10K	50/49/10K	69/70/13K	47/47/10K	72/75/19K	53/54/18K	48/48/13K
	320	51/51/15K	52/52/15K	48/48/15K	51/51/11K	53/53/11K	74/73/15K	50/49/12K	71/71/22K	49/49/18K	51/51/15K
Grandcru	128	4208/5651/305K	1731/2024/120K	1860/2401/132K	1634/2274/124K	1675/2192/123K	2626/3113/183K	1738/2136/124K	2752/4145/192K	1640/2109/114K	1612/2106/115K
Hierocrypt-3	128	49/56/172K	54/68/131K	54/79/195K	61/63/205K	60/63/204K	57/76/219K	60/64/202K	59/71/100K	56/72/125K	52/61/116K
	192	59/66/201K	63/81/155K	63/93/229K	69/75/242K	69/74/241K	67/90/265K	70/76/239K	70/88/124K	66/84/148K	60/72/138K
	256	67/76/232K	71/93/178K	73/107/266K	80/85/277K	79/85/278K	77/101/303K	81/86/276K	89/105/147K	99/134/173K	69/83/159K
Noekeon-Dir	128	138/138/61	89/91/34	97/100/31	71/72/32	67/116/31	100/99/31	107/107/32	114/102/27	79/75/35	77/78/35
Noekeon-Ind	128	74/75/1131	52/57/1027	97/100/1494	64/66/948	63/66/974	102/100/1559	64/67/877	57/58/853	42/49/800	36/36/620
Nush	128	28/25/2556	28/25/2581	37/21/2005	31/20/1982	31/20/1978	37/21/1999	32/20/2005	24/21/3077	25/22/3134	22/20/3079
	256	28/25/2585	28/25/2613	37/21/2019	31/20/2000	31/20/1999	37/21/2027	32/20/2015	24/21/3104	25/22/3156	22/19/3088
Q	128	60/144/1128	59/145/1022	53/254/986	51/231/1028	51/231/1032	55/256/1032	51/251/1079	69/185/1236	54/189/971	48/167/954
	256	67/160/1228	64/162/1193	58/283/1153	58/256/1161	57/257/1147	60/297/1251	56/279/1165	69/205/1413	58/199/1110	54/185/1066
SC2000	128	139/139/7106	78/50/3031	96/70/5188	60/60/4661	64/64/4676	123/79/6364	77/81/5146	33/55/10K	63/37/2718	40/43/3182
	192	159/159/8008	90/57/3357	111/80/5856	68/69/6612	74/75/6552	143/89/7149	90/91/5798	45/65/10K	72/41/3023	46/48/3566
	256	159/159/8100	90/57/3386	111/81/5909	68/69/6669	73/74/6624	141/89/7240	90/91/5841	40/65/9344	72/41/3044	46/48/3593
Mars	128	45/41/2871	43/37/2635	138/160/3628	82/72/3492	81/74/3469	148/161/3306	89/102/3494	39/38/2623	45/38/2435	29/32/2411
	192	45/42/2891	43/37/2639	138/161/3635	82/72/3480	81/74/3456	146/162/3294	88/102/3410	40/38/2627	45/38/2445	29/32/2412
	256	45/41/2893	43/37/2654	137/165/3667	82/73/3599	81/74/3458	148/164/3368	88/102/3337	40/39/2692	45/38/2440	29/32/2429
Rijndael	128	22/23/612	28/28/973	26/26/1329	24/25/689	24/25/711	33/33/1525	24/25/1228	32/35/500	31/32/852	30/31/748
	192	27/27/714	32/33/1208	31/30/1367	28/29/820	28/29/839	39/38/1831	27/28/1388	39/42/511	36/38/1107	35/36/833
	256	31/31/878	37/37/1412	35/35/1651	32/38/1327	32/33/1363	43/46/2209	32/33/1546	47/46/708	41/42/1431	39/41/1091
Seed	128	56/56/447	51/51/456	63/63/467	69/69/401	69/70/385	84/81/778	84/83/638	52/51/410	42/42/406	41/41/393
Serpent	128	83/77/1872	73/86/2139	168/199/2819	154/172/2235	153/170/2230	157/162/3078	153/174/2664	68/64/1731	62/80/1675	60/77/1691
	192	83/76/1874	73/86/2148	167/199/2826	155/171/2232	152/169/2226	158/161/3111	153/174/2706	68/65/1731	62/79/1681	60/77/1704
	256	83/76/1884	73/86/2139	167/199/2828	154/171/2231	153/171/2235	157/161/3106	153/173/2650	68/65/1707	63/79/1703	60/77/1686
Twofish	128	31/33/11K	34/36/12K	66/57/12K	51/49/8871	52/49/12K	57/51/20K	51/48/15K	37/33/24K	41/47/24K	30/34/19K
	192	31/33/13K	34/36/15K	66/56/15K	52/49/11K	52/48/14K	56/51/23K	52/48/17K	37/33/27K	41/47/30K	29/34/26K
	256	31/33/17K	34/36/20K	65/57/21K	52/49/13K	52/48/16K	57/51/28K	51/48/19K	37/33/32K	41/46/37K	30/34/32K

The performances given are those for encryption/decryption/key setup, where the encryption and decryption times are measured in cycles/byte and the key setup time is measured in cycles.

Table 42. Normal and High Block Ciphers Performance Results by Processor and Compiler (continued)

Primitive	Key	Platform, Processor and Compiler									
		Alpha	Alpha	Sun/Sparc V9	Sun/Sparc	Sun/Sparc V9	Sun/Sparc V9	Sun/Sparc V9	Sun/Sparc V9	Sun	Sun
Name	Size			450MHz	248MHz	338MHz	338MHz	400MHz	400MHz	450MHz	333MHz
		cc	gcc 2.97	gcc 3.0.4+3.2.1	gcc 2.95.3	cc	gcc 3.0.4	cc	gcc 3.0.4	cc	gcc 3.2.1
Camellia	128	48/48/287	36/35/320	49/49/375	55/55/352	57/57/759	54/54/405	47/47/557	50/50/376	53/53/712	51/51/381
	192	64/64/512	48/47/407	72/72/539	74/74/556	71/73/958	70/69/585	61/61/808	66/66/546	64/64/828	75/75/540
	256	64/64/525	48/47/453	72/72/544	74/74/563	72/73/978	70/71/569	60/60/849	66/65/542	65/64/817	74/74/564
Camellia 2nd impl	128	51/51/276	44/44/311	50/46/327	55/56/293	60/59/664	53/51/335	52/52/535	50/48/306	53/52/568	51/47/323
	192	66/64/504	58/58/416	72/72/529	74/74/508	78/77/891	70/63/559	65/67/772	66/60/518	68/71/829	68/64/521
	256	66/64/521	58/58/441	72/72/521	74/73/528	79/78/910	72/65/549	65/67/799	65/60/514	68/72/807	75/75/574
RC6 (20 rounds)	128	27/24/1509	27/27/1664	69/68/1465	70/68/1581	120/123/1599	71/69/1508	116/115/1537	70/68/1432	120/121/1725	69/68/1582
	192	27/24/1725	27/27/1905	174/167/6946	176/171/7267	121/124/2062	178/175/7274	117/117/1901	170/169/6856	119/121/2023	190/183/7651
	256	26/24/1727	27/27/1906	70/70/2317	69/69/2207	121/123/2070	72/72/2378	95/95/1921	70/70/2315	99/100/2047	71/70/2609
Safer++	128	33/93/1723	30/44/1677	47/70/2665	48/65/2817	36/45/2076	50/74/2800	35/43/2002	48/71/2695	37/49/1939	47/66/2758
	256	43/129/2425	42/62/2257	67/99/3656	68/91/3842	52/75/2839	69/102/3790	49/73/2720	68/100/3588	52/74/2590	67/93/3685
Anubis	128	29/29/2990	25/25/3854	33/33/7064	37/37/6250	38/38/5054	34/34/6611	37/37/4885	33/33/6341	35/34/5011	36/36/6188
	160	31/30/3950	27/27/4993	35/35/8874	38/38/7525	40/40/6614	36/37/8510	38/38/6329	36/36/8275	37/37/6573	38/39/7659
	192	32/32/4803	29/29/6274	37/37/11K	40/41/9652	43/43/8424	39/40/10K	42/41/8088	38/38/10K	39/39/8337	41/41/10K
	224	34/34/5715	31/31/7755	40/40/12K	43/43/11K	45/45/10K	41/42/12K	43/43/9989	40/40/12K	42/42/11K	43/43/12K
	256	36/36/8234	33/33/9237	42/42/16K	45/45/14K	48/48/12K	44/44/15K	46/46/12K	42/43/14K	44/44/14K	46/46/14K
	288	37/37/9954	35/35/10K	51/51/18K	50/49/16K	49/49/14K	46/46/18K	47/48/14K	45/45/17K	47/47/16K	49/49/17K
	320	39/39/10K	36/37/12K	53/53/21K	51/51/19K	54/53/16K	49/49/21K	52/52/16K	47/47/20K	50/49/19K	50/50/20K
Grandcru	128	1326/1754/93K	1028/1278/72K	1760/2247/125K	1867/2312/139K	1895/2719/133K	1809/2321/129K	1815/2638/130K	1762/2250/125K	2029/2896/139K	1574/2166/117K
Hierocrypt-3	128	33/39/131K	34/39/136K	81/116/128K	80/99/123K	62/81/162K	90/117/126K	59/78/153K	87/113/122K	62/82/138K	82/118/118K
	192	38/46/153K	39/46/161K	103/132/147K	93/111/145K	75/90/192K	96/140/149K	71/87/183K	94/134/140K	80/88/163K	101/130/141K
	256	44/52/176K	44/52/185K	109/162/170K	109/130/168K	83/106/218K	120/156/171K	80/103/208K	117/153/169K	84/108/217K	109/145/161K
Noekeon-Dir	128	108/110/40	72/76/28	77/76/15	74/76/25	73/62/15	79/78/15	70/60/15	76/75/15	55/54/16	85/87/16
Noekeon-Ind	128	46/46/647	41/42/635	37/36/547	40/39/648	49/49/802	38/37/559	47/47/780	38/36/546	40/40/703	39/39/598
Nush	128	23/15/1305	27/16/2145	31/26/3985	35/26/3900	28/28/5277	31/27/4081	27/27/5127	31/26/3975	24/21/4915	29/29/4272
	256	23/15/1322	27/16/2161	30/26/4001	35/26/3932	28/28/5138	32/27/4080	27/27/5002	31/26/4000	24/21/4985	29/29/4285
Q	128	35/174/764	42/165/760	63/180/895	55/206/880	52/150/888	65/190/896	51/146/856	64/181/886	47/87/930	63/204/980
	256	39/195/861	47/184/852	70/201/1021	61/230/1014	56/167/1058	73/212/1039	54/162/1017	70/202/1021	56/98/1150	69/229/1132
SC2000	128	59/63/4053	32/31/3307	35/32/4955	31/32/4885	31/33/8004	37/33/5101	29/31/7733	36/32/4977	31/31/7587	48/52/3866
	192	66/71/4491	36/36/3701	41/37/5483	37/37/5461	35/37/8948	42/38/5676	34/36/8721	41/37/5532	35/35/8823	55/60/4862
	256	66/71/4515	37/37/3730	41/37/5523	37/37/5574	35/38/9013	42/38/5757	34/36/8720	41/37/5549	35/36/8638	55/61/4865
Mars	128	33/29/2924	37/33/3189	48/49/2865	51/52/2863	77/76/3152	50/51/2987	74/74/2975	49/50/2884	74/73/2925	55/55/3127
	192	33/29/2929	37/33/3185	48/49/2872	51/52/2884	76/76/3102	50/51/2984	75/74/3004	49/50/2891	73/73/2947	55/56/3153
	256	33/29/2934	37/33/3191	49/49/2904	51/52/2931	77/76/3140	50/51/3054	75/73/3040	49/50/2928	73/72/2998	55/56/3179
Rijndael	128	17/17/493	19/19/756	31/29/684	31/31/705	24/21/557	31/32/673	22/21/532	30/31/650	21/22/453	29/30/655
	192	20/21/449	22/22/847	36/35/783	38/36/724	26/26/568	37/37/770	25/25/542	37/36/734	25/28/463	35/36/796
	256	24/24/663	25/25/1044	42/41/1122	43/43/1192	29/31/779	44/43/1231	28/30/750	43/42/1186	29/32/753	43/42/1245
Seed	128	44/44/358	40/40/340	35/35/381	36/36/447	39/38/399	39/39/483	36/36/368	36/36/453	34/34/353	39/37/396
Serpent	128	54/41/1383	67/54/1904	67/61/1563	69/65/1626	66/55/1716	70/63/1635	63/53/1645	68/61/1586	57/53/1708	66/61/1728
	192	54/41/1381	67/54/2082	67/61/1577	69/65/1639	65/55/1720	70/63/1658	63/53/1665	68/61/1606	57/54/1716	66/61/1737
	256	54/41/1376	67/54/1900	67/61/1575	69/65/1645	65/55/1719	70/63/1638	64/53/1654	69/61/1601	57/53/1710	66/61/1739
Twofish	128	24/24/8661	20/20/10K	40/41/15K	39/40/14K	29/28/15K	43/44/16K	28/28/14K	40/42/15K	28/28/15K	38/39/15K
	192	24/24/12K	20/20/13K	40/42/19K	41/40/18K	29/29/19K	42/44/20K	29/28/18K	41/42/19K	28/27/18K	38/39/20K
	256	24/24/23K	20/20/17K	40/41/24K	40/40/22K	30/29/23K	42/42/26K	28/28/22K	43/42/24K	28/27/23K	38/39/24K

The performances given are those for encryption/decryption/key setup, where the encryption and decryption times are measured in cycles/byte and the key setup time is measured in cycles.

Table 43. Block Ciphers with 160- and 256-Bit Blocks by Processor and Compiler

Primitive	Key	Block	Platform, Processor and Compiler									
Name	Size	Size	PIII/Linux Coppermine gcc 2.95.2	PIII/Linux Coppermine gcc 3.1.1	PIII/Linux Coppermine gcc 3.0.3	PIII/Linux Tualatin gcc 2.96	PIII/Linux Katmai gcc 3.1.1	PIII/Linux Katmai gcc 3.3	PIII/Linux Coppermine gcc-egcs	PIII/Win Visual C 6.0	PIII/Win Intel C	PIII/Win gcc 2.95.3
SHACAL-1	512	160	29/26/78	34/27/445	29/27/78	31/34/515	34/27/75	34/27/498	30/31/439	44/65/839	27/31/715	30/26/447
SHACAL-2	512	256	45/46/745	46/45/737	45/47/765	46/48/891	46/45/871	46/45/885	44/43/793	60/59/1187	46/47/748	44/46/753
RC6 (20 rounds)	256	256	85/89/13K	61/58/10K	60/62/11K	86/85/17K	61/58/10K	59/59/11K	96/88/15K	120/120/17K	125/118/12K	85/83/12K
Rijndael	192	256	34/39/2426	36/35/1785	35/35/2609	37/38/2811	36/34/2001	36/35/1999	33/39/2476	33/32/1882	40/39/2068	34/39/2454
	256	256	33/37/3053	36/36/2298	35/35/3255	36/35/3575	36/36/2538	36/36/2560	33/37/3146	34/33/2362	35/36/2798	34/37/3105
			PII/Win Visual C 6.0	PII/Win gcc 2.95.3	PIV/Linux Northwood gcc 2.95.2	PIV/Linux Northwood gcc 3.1.1	Xeon Model 1 gcc 3.2	Xeon Model 0 egcs 2.91.66	Xeon Model 0 gcc 2.96	AMD Duron Win XP VC 7.0 (.NET)	AMD Duron Win XP gcc 2.95.3	AMD Athlon Linux gcc 2.96
SHACAL-1	512	160	45/56/64	30/26/80	63/36/177	59/37/457	57/37/423	81/62/402	62/37/173	37/37/54	27/25/58	28/24/318
SHACAL-2	512	256	60/58/1277	44/46/888	55/60/933	51/58/1015	52/59/1075	79/64/930	51/54/857	56/52/861	38/40/597	40/40/633
RC6 (20 rounds)	256	256	125/125/18K	85/83/12K	190/171/23K	132/123/21K	133/120/21K	173/174/25K	133/133/23K	101/101/18K	78/79/15K	58/60/12K
Rijndael	192	256	33/31/2145	34/39/2558	42/48/4735	35/35/2512	35/35/2557	42/47/3778	35/37/3841	50/46/2291	50/49/2722	50/50/2430
	256	256	33/32/2705	34/38/3254	35/44/5982	38/36/3172	34/36/3252	35/43/4797	58/36/4834	53/47/2985	52/56/3561	53/53/3089
			Alpha cc	Alpha gcc 2.97	Sun/Sparc V9 450MHz gcc 3.0.4+3.2.1	Sun/Sparc 248MHz gcc 2.95.3	Sun/Sparc V9 338MHz cc	Sun/Sparc V9 338MHz gcc 3.0.4	Sun/Sparc V9 400MHz cc	Sun/Sparc V9 400MHz gcc 3.0.4	Sun 450MHz cc	Sun 333MHz gcc 3.2.1
SHACAL-1	512	160	35/27/60	30/27/44	34/32/288	33/31/259	37/27/152	35/33/146	36/27/141	34/32/136	30/27/257	34/32/289
SHACAL-2	512	256	37/33/671	30/31/706	37/41/1169	37/42/1191	39/38/1267	39/43/1212	38/37/1237	37/41/1176	38/38/1251	37/42/1184
RC6 (20 rounds)	256	256	12/11/2716	12/11/5511	100/100/19K	98/100/18K	–/–/–	103/103/20K	–/–/–	101/100/19K	–/–/–	110/110/15K
Rijndael	192	256	24/24/1818	–/–/–	46/47/3001	44/45/3052	32/30/2273	47/50/3086	31/29/2159	46/49/2957	30/30/2301	47/55/2759
	256	256	24/24/2326	–/–/–	45/56/3629	47/45/3801	33/33/2698	47/51/3763	31/32/2584	46/49/3610	33/32/2815	48/54/3441

The performances given are those for encryption/decryption/key setup, where the encryption and decryption times are measured in cycles/byte and the key setup time is measured in cycles.

Table 44. Stream Ciphers Performance Results by Processor and Compiler

Primitive	Key	IV	Platform, Processor and Compiler									
			PIII/Linux	PIII/Linux	PIII/Linux	PIII/Linux	PIII/Linux	PIII/Linux	PIII/Linux	PIII/Win	PIII/Win	PIII/Win
Name	Size	Size	Coppermine	Coppermine	Coppermine	Tualatin	Katmai	Katmai	Coppermine			
			gcc 2.95.2	gcc 3.1.1	gcc 3.0.3	gcc 2.96	gcc 3.1.1	gcc 3.3	gcc-egcs	Visual C 6.0	Intel C	gcc 2.95.3
BMGL (m=16)	128		1674/2455	1501/2355	1483/2460	1854/7234	1492/2541	1475/2541	2834/2340	1641/3258	1484/2762	1631/2336
2nd impl	128		540/193K	426/190K	463/214K	533/203K	437/188K	424/183K	510/205K	438/184K	416/189K	506/195K
Snow	128		8.2/1319	6.8/1199	6.7/1179	11/1981	6.8/1207	6.8/1224	7.8/1451	7.5/1757	6.5/1424	8.3/1328
	256		8.2/1333	6.8/1243	6.7/1206	11/2059	6.8/1264	6.8/1285	7.8/1461	7.5/1845	6.4/1444	8.3/1352
Sober-t16	128		61/1753	40/1074	40/1119	68/1635	40/1070	42/1033	67/1720	49/1414	44/1188	62/1635
	256		61/2379	40/1546	40/1604	67/2236	40/1546	42/1456	67/2263	49/1993	44/1604	62/2234
Sober-t32	128		21/923	22/842	20/838	25/900	26/1061	25/1068	21/949	24/1105	23/1057	21/913
	256		21/1138	22/1025	20/1032	26/1118	25/1296	25/1267	21/1152	24/1351	23/1293	21/1119
Leviathan	128		13/78K	10/79K	12/78K	12/113K	10/79K	10/79K	16/77K	15/112K	10/108K	13/79K
	192		12/78K	10/78K	12/78K	12/113K	10/78K	10/79K	16/77K	15/111K	10/108K	13/79K
	256		13/78K	10/79K	12/78K	12/113K	10/78K	10/79K	16/77K	15/112K	10/108K	13/79K
Lili	128		914/60	935/46	964/51	1103/72	921/46	936/47	827/65	1056/43	959/43	921/58
RC4	128		7.4/2659	7.3/3015	7.3/2885	8.1/3698	7.3/2879	7.3/3195	7.6/2963	8.2/3340	10/3469	8.0/2704
BMGL with IV	128	128	1679/2437/20	1506/2344/20	1485/2457/20	1872/7087/57	1494/2421/20	1474/2436/20	2848/2393/20	1779/3237/17	1504/2753/18	1620/2326/21
2nd impl	128	128	538/193K/20	427/183K/20	460/209K/20	526/203K/23	438/181K/20	423/178K/20	510/204K/20	439/187K/17	414/191K/18	507/195K/21
Snow	128	64	8.2/17/680	6.9/15/620	6.7/17/612	11/46/916	6.9/15/620	6.9/15/624	7.6/17/730	7.5/12/782	6.2/12/705	8.3/17/690
with IV	256	64	8.2/51/680	6.9/44/621	6.7/50/612	11/56/947	6.9/44/621	6.9/45/625	7.6/50/729	7.5/34/831	6.2/16/719	8.2/49/689
Sober-t16	128	128	61/2005/1759	40/1167/1033	40/1240/1052	67/1950/1649	40/1178/1028	42/1158/1025	67/1908/1740	49/1498/1332	44/1253/1186	61/1772/1575
with IV	256	128	61/2688/1759	40/1627/1034	40/1712/1052	68/2572/1642	40/1659/1029	42/1583/1025	67/2500/1740	49/2061/1332	44/1675/1186	61/2353/1577
Sober-t32	128	128	21/1126/1202	21/995/995	20/1005/995	26/1142/1151	26/1279/1002	25/1253/1012	21/1149/1158	24/1253/1282	23/1198/1285	21/1121/1176
with IV	256	128	21/1355/1200	22/1176/994	20/1189/994	25/1380/1153	26/1499/995	25/1481/1011	21/1379/1157	24/1504/1285	23/1439/1290	21/1353/1180
Scream-0	128	128	6.6/3603/1124	10/4059/1671	6.8/3732/1169	6.7/3845/1233	9.9/3983/1652	10/4071/1683	6.9/3603/1126	10.0/5072/2167	7.5/4002/1163	6.7/3651/1141
Scream-F	128	128	7.5/3604/1125	11/4022/1659	7.9/3732/1169	10/3854/1234	10/3990/1645	10/4018/1681	10/3598/1126	12/5077/2169	8.0/4008/1161	7.6/3661/1141
Scream-S	128	128	7.0/51K/1236	9.8/40K/1723	6.9/49K/1193	7.3/61K/1253	9.7/40K/1713	10/41K/1740	7.1/56K/1224	10/48K/2285	7.5/40K/1186	7.1/52K/1230
Seal-3.0	160	32	5.9/176K/18	6.2/161K/18	6.7/166K/18	7.9/180K/20	6.2/158K/18	6.2/161K/18	8.7/187K/17	6.9/227K/21	7.1/172K/12	6.0/183K/18

The performances given in the 1st part of the table are those for key stream generation and key setup, where the key stream generation time is measured in cycles/byte and the key setup time is measured in cycles. For the stream ciphers with initial value (IV) setup in the 2nd part of the table, the performances given are those for key stream generation/key setup/IV setup, where the IV setup time is measured in cycles.

Table 45. Stream Ciphers Performance Results by Processor and Compiler (continued)

Primitive	Key	IV	Platform, Processor and Compiler									
			PII/Win	PII/Win	PIV/Linux	PIV/Linux	Xeon	Xeon	Xeon	AMD Duron	AMD Duron	AMD Athlon
Name	Size	Size			Northwood	Northwood	Model 1	Model 0	Model 0	Win XP	Win XP	Linux
			Visual C 6.0	gcc 2.95.3	gcc 2.95.2	gcc 3.1.1	gcc 3.2	egcs 2.91.66	gcc 2.96	VC 7.0 (.NET)	gcc 2.95.3	gcc 2.96
BMGL (m=16)	128		1644/3310	1624/2614	2140/4596	1992/2960	2009/2960	2501/4011	1917/4042	1945/2777	2128/2141	1867/2160
2nd impl	128		441/211K	519/194K	895/350K	706/345K	707/332K	904/356K	668/362K	504/541K	592/185K	530/172K
Snow	128		9.1/1823	8.3/1349	9.2/2725	8.6/1674	8.7/1707	9.3/2707	7.7/2601	8.3/1465	8.1/1530	12/1554
	256		8.9/1908	8.3/1382	9.2/2920	8.7/1735	8.7/1753	9.4/2777	7.8/2664	8.3/1491	8.0/1597	12/1622
Sober-t16	128		64/1672	62/1640	55/1891	45/1164	46/1146	56/1933	52/1346	46/1249	52/1264	46/1207
	256		55/2259	62/2247	55/2778	46/1610	46/1526	56/2856	52/1726	46/1646	52/1682	46/1577
Sober-t32	128		27/1247	26/1119	27/1098	28/944	28/962	29/1119	28/1112	21/1009	19/846	18/862
	256		27/1539	26/1364	27/1290	28/1140	28/1146	29/1307	27/1294	21/1226	21/1116	18/1021
Leviathan	128		15/119K	13/80K	16/76K	10/79K	10/80K	18/74K	13/98K	14/103K	11/98K	10/99K
	192		15/119K	13/80K	15/76K	10/79K	10/80K	18/74K	12/98K	14/103K	11/98K	10/99K
	256		15/120K	13/80K	16/76K	10/80K	10/80K	18/74K	12/98K	14/103K	11/98K	10/99K
Lili	128		1104/53	918/59	1023/76	987/59	966/59	1361/78	1087/61	869/44	855/53	841/50
RC4	128		7.8/3484	8.1/2824	20/2568	20/4680	20/4553	20/4675	20/2879	11/3211	11/2600	12/2675
BMGL with IV	128	128	1822/3263/16	1618/2461/21	2136/4791/20	2002/2943/21	2020/2943/20	2505/4136/20	1899/3993/20	2103/2913/15	2059/2127/20	1861/2126/19
2nd impl	128	128	445/184K/16	520/195K/21	918/353K/20	710/335K/21	706/333K/20	890/344K/20	675/368K/21	501/566K/15	593/170K/20	530/174K/18
Snow	128	64	9.0/11/772	8.3/17/689	9.2/27/1270	8.3/19/846	8.5/19/857	9.5/24/1168	7.5/23/933	8.5/14/638	8.0/15/679	13/15/725
with IV	256	64	9.0/32/819	8.3/50/689	9.2/59/1327	8.7/73/846	8.8/76/865	9.4/60/1176	7.6/82/913	8.5/38/809	8.0/45/680	13/44/758
Sober-t16	128	128	60/1714/1534	61/1808/1583	55/1700/1630	46/1287/1113	46/1299/1078	56/1908/1739	52/1502/1325	46/1309/1202	51/1411/1236	46/1281/1127
with IV	256	128	55/2343/1530	61/2425/1583	55/2497/1619	46/1760/1092	46/1665/1061	57/2736/1765	52/1891/1280	46/1684/1199	51/1831/1238	46/1668/1128
Sober-t32	128	128	27/1410/1302	26/1398/1200	27/1242/1271	28/1110/1126	28/1157/1166	28/1338/1270	27/1243/1258	21/1063/1196	20/1163/1190	18/925/1007
with IV	256	128	27/1696/1307	26/1677/1205	27/1446/1274	27/1330/1129	28/1382/1175	29/1566/1264	27/1422/1241	21/1280/1196	20/1315/1187	18/1106/1006
Scream-0	128	128	9.0/5453/2124	6.7/3684/1143	12/7700/2225	14/7196/2544	14/7206/2531	12/7736/2276	12/7150/2379	10/5295/1651	6.8/4403/1281	7.1/4205/1330
Scream-F	128	128	12/5391/2124	7.6/3684/1143	12/7700/2221	16/7300/2541	16/7260/2531	13/7735/2266	13/7139/2381	11/5295/1649	8.6/4403/1259	9.6/4204/1330
Scream-S	128	128	10/74K/2150	7.1/51K/1231	12/70K/2434	14/47K/2792	14/48K/2809	12/73K/2553	12/57K/2332	10/49K/1855	7.2/54K/1383	7.2/56K/1414
Seal-3.0	160	32	6.8/220K/22	6.0/178K/18	5.9/376K/18	6.9/335K/16	7.0/339K/15	13/374K/18	5.5/337K/16	5.9/300K/12	5.8/159K/16	5.9/141K/17

The performances given in the 1st part of the table are those for key stream generation and key setup, where the key stream generation time is measured in cycles/byte and the key setup time is measured in cycles. For the stream ciphers with initial value (IV) setup in the 2nd part of the table, the performances given are those for key stream generation/key setup/IV setup, where the IV setup time is measured in cycles.

Table 46. Stream Ciphers Performance Results by Processor and Compiler (continued)

Primitive	Key	IV	Platform, Processor and Compiler									
Name	Size	Size	Alpha cc	Alpha gcc 2.97	Sun/Sparc V9 450MHz gcc 3.0.4+3.2.1	Sun/Sparc 248MHz gcc 2.95.3	Sun/Sparc V9 338MHz cc	Sun/Sparc V9 338MHz gcc 3.0.4	Sun/Sparc V9 400MHz cc	Sun/Sparc V9 400MHz gcc 3.0.4	Sun 450MHz cc	Sun 333MHz gcc 3.2.1
BMGL (m=16)	128		1126/1345	1382/1653	1693/2713	1861/2485	1643/2648	1731/2812	1583/2557	1665/2717	1739/2913	1691/2694
2nd impl	128		384/126K	429/122K	641/224K	676/235K	647/297K	664/244K	630/282K	640/236K	570/278K	699/230K
Snow	128		10/997	5.6/1242	7.2/1787	7.5/1826	8.2/1857	7.4/1951	7.7/1770	7.0/1838	6.7/1876	7.5/1823
	256		10/1027	5.6/1285	7.1/1801	7.5/1944	8.1/1905	7.6/2032	7.8/1791	7.1/1916	6.6/1882	7.4/1856
Sober-t16	128		79/1551	38/979	42/1538	47/1599	43/1565	44/1543	41/1495	42/1503	47/1656	45/1404
	256		79/2033	38/1307	43/2046	47/2090	42/2089	44/2067	41/2022	42/2004	46/2200	46/1894
Sober-t32	128		34/795	18/759	24/1501	25/1594	23/1303	24/1560	22/1260	24/1512	25/1355	24/1539
	256		34/957	18/948	24/1787	25/1898	23/1572	25/1859	22/1508	24/1795	25/1625	24/1794
Leviathan	128		12/25K	13/71K	15/135K	15/130K	16/165K	15/140K	16/156K	14/135K	15/156K	15/134K
	192		12/75K	13/83K	14/134K	15/129K	17/165K	15/140K	16/158K	15/134K	16/156K	15/133K
	256		12/25K	13/70K	14/133K	15/129K	17/163K	15/139K	16/158K	14/134K	16/156K	15/137K
Lili	128		581/47	566/34	757/71	672/70	691/73	798/73	665/71	760/71	757/75	720/71
RC4	128		12/3010	12/3243	14/3045	14/3070	13/3359	14/2714	13/3212	14/2667	13/3342	14/3539
BMGL with IV	128	128	1124/1328/44	1382/1687/28	1688/2770/160	1846/2600/158	1571/3092/192	1744/2861/186	1582/2979/170	1687/2752/169	1729/2938/196	1668/2691/174
2nd impl	128	128	382/126K/44	439/122K/29	679/225K/152	662/233K/160	657/304K/192	679/250K/185	623/275K/180	639/220K/162	579/280K/185	704/239K/182
Snow	128	64	10/28/520	5.6/14/579	7.2/52/918	7.6/56/903	8.0/55/940	7.4/56/995	7.6/52/880	7.0/52/947	6.7/51/900	7.2/51/963
with IV	256	64	10/30/541	5.6/18/610	7.2/74/890	7.6/80/973	8.2/78/954	7.4/86/1001	7.7/72/889	7.0/79/935	6.6/77/899	7.3/74/974
Sober-t16	128	128	79/1603/1489	38/1031/911	43/1672/1434	47/1746/1531	42/1703/1456	45/1714/1496	41/1643/1415	42/1657/1438	42/1798/1580	45/1556/1326
with IV	256	128	79/2059/1498	38/1371/907	43/2185/1459	47/2264/1505	42/2240/1464	45/2251/1485	40/2171/1410	42/2161/1434	47/2332/1548	45/1985/1350
Sober-t32	128	128	34/856/944	18/850/843	23/1609/1528	25/1741/1617	23/1449/1445	24/1673/1551	23/1389/1400	25/1596/1518	24/1482/1488	25/1734/1502
with IV	256	128	34/1034/948	19/1006/856	23/1899/1508	25/2037/1619	24/1712/1480	24/1965/1560	23/1651/1423	24/1901/1508	24/1758/1500	24/1973/1520
Scream-0	128	128	9.5/5780/1597	7.5/2944/991	10/5958/1606	10/6038/1672	11/6422/1725	10/6415/1704	11/6249/1660	10/6106/1634	11/7417/1932	13/6639/2299
Scream-F	128	128	10/5786/1586	12/2944/992	12/5995/1608	12/6045/1699	13/6514/1834	12/6396/1680	13/6250/1752	12/6104/1627	13/7470/1710	15/6693/2301
Scream-S	128	128	9.6/75K/1481	7.5/47K/995	9.9/44K/1712	10/42K/1678	12/30K/2007	10/48K/1752	11/28K/1892	10/46K/1691	13/29K/2166	13/41K/2255
Seal-3.0	160	32	10/137K/11	4.7/153K/10	3.8/206K/8.7	4.5/215K/10	5.2/199K/13	4.0/214K/9.6	5.0/193K/12	3.8/206K/8.8	5.2/190K/13	3.9/209K/9.4

The performances given in the 1st part of the table are those for key stream generation and key setup, where the key stream generation time is measured in cycles/byte and the key setup time is measured in cycles.
For the stream ciphers with initial value (IV) setup in the 2nd part of the table, the performances given are those for key stream generation/key setup/IV setup, where the IV setup time is measured in cycles.

Table 47. Hash Functions Performance Results by Processor and Compiler

Primitive	**Hash**	**Platform, Processor and Compiler**									
Name	**Size**	**PIII/Linux Coppermine gcc 2.95.2**	**PIII/Linux Coppermine gcc 3.1.1**	**PIII/Linux Coppermine gcc 3.0.3**	**PIII/Linux Tualatin gcc 2.96**	**PIII/Linux Katmai gcc 3.1.1**	**PIII/Linux Katmai gcc 3.3**	**PIII/Linux Coppermine gcc-egcs**	**PIII/Win Visual C 6.0**	**PIII/Win Intel C**	**PIII/Win gcc 2.95.3**
Whirlpool	512	83/64/5859	79/60/5557	79/59/5534	82/82/5771	89/62/6098	89/60/6122	86/65/6109	82/54/5657	90/47/6323	84/65/5946
2nd impl (MMX)	512	72/63/4869	46/64/3199	72/65/4884	78/83/5273	54/63/3762	54/63/3764	77/70/5201	130/83/8590	95/50/6298	73/71/5026
MD4	128	4.7/16/481	4.8/15/442	4.8/15/449	4.7/18/493	4.8/15/440	4.7/15/447	4.7/15/441	4.8/14/467	4.5/12/411	4.7/15/444
MD5	128	7.2/16/645	7.3/15/611	7.5/15/602	7.5/18/665	7.3/15/608	7.5/15/626	7.3/15/610	7.2/14/612	6.8/12/558	7.3/15/611
RIPEMD	160	23/16/1716	18/16/1345	22/16/1568	24/19/1748	18/16/1339	18/16/1347	22/16/1580	23/16/1675	16/14/1207	24/17/1704
SHA-0	160	20/16/1148	15/16/1025	15/16/1011	16/19/1097	15/16/1020	16/16/1024	16/16/1121	19/16/1100	12/14/896	16/17/1003
SHA-1	160	15/16/1195	15/16/1029	16/16/1053	16/19/1149	15/16/1024	15/16/1049	15/16/1187	19/16/1152	13/14/929	16/17/1083
SHA-2	256	40/45/2829	40/44/2755	40/47/2803	44/49/3068	40/44/2747	39/44/2736	41/45/2891	56/36/3816	39/20/2643	40/45/2850
	384	83/168/11K	87/157/11K	87/158/11K	89/195/12K	86/252/11K	104/254/13K	110/157/14K	80/144/10K	74/101/9907	84/162/11K
	512	83/168/11K	87/156/11K	87/158/11K	89/196/12K	86/252/11K	104/253/13K	110/157/14K	80/144/10K	74/101/9940	84/162/11K
Tiger	192	24/18/1722	22/16/1548	21/16/1542	21/19/1624	21/16/1547	21/16/1589	24/16/1761	25/15/1822	24/14/1745	24/18/1744
BCHASH-Rijndael	128	63/15/2188	49/15/1712	50/15/1721	59/18/2072	49/15/1723	50/15/1757	64/15/2182	45/14/1588	50/15/1686	58/15/2029
	256	143/45/4778	112/45/3775	146/47/4854	164/49/5514	112/45/3882	112/44/3878	155/45/5231	116/35/3913	128/20/4159	143/45/4785

		PII/Win Visual C 6.0	**PII/Win gcc 2.95.3**	**PIV/Linux Northwood gcc 2.95.2**	**PIV/Linux Northwood gcc 3.1.1**	**Xeon Model 1 gcc 3.2**	**Xeon Model 0 egcs 2.91.66**	**Xeon Model 0 gcc 2.96**	**AMD Duron Win XP VC 7.0 (.NET)**	**AMD Duron Win XP gcc 2.95.3**	**AMD Athlon Linux gcc 2.96**
Whirlpool	512	115/51/7870	97/68/6804	113/130/8905	108/132/8559	112/125/8794	123/127/9665	114/127/8851	104/42/7012	106/51/7328	109/49/7341
2nd impl (MMX)	512	146/83/9666	83/76/5992	97/132/7350	60/132/4495	60/125/4468	103/132/7652	99/129/7380	127/73/8335	102/54/6707	99/51/6573
MD4	128	4.8/13/462	4.7/15/444	6.4/34/639	6.5/34/600	6.4/34/598	6.4/29/611	6.5/33/586	4.8/8.3/419	4.7/10/443	4.8/11/423
MD5	128	7.2/12/611	7.3/15/613	9.4/34/828	9.5/34/799	9.4/34/794	9.4/29/786	9.5/33/785	7.5/11/599	8.0/10/653	8.0/11/631
RIPEMD	160	23/15/1651	24/17/1706	46/36/3203	26/36/1929	26/36/1921	35/36/2507	40/36/2866	21/12/1493	22/13/1628	23/12/1623
SHA-0	160	19/15/1089	16/17/1008	30/36/1828	23/36/1651	23/36/1619	27/36/1813	25/36/1438	12/13/1383	13/13/870	13/12/796
SHA-1	160	19/15/1136	16/17/1086	30/20/1903	25/20/1497	25/20/1487	26/30/2125	26/20/1460	14/13/1361	13/13/910	12/12/825
SHA-2	256	56/34/3799	40/46/2860	51/118/3817	40/120/3159	39/112/3042	53/114/4105	48/116/3510	46/39/3135	34/42/2425	34/41/2369
	384	118/244/15K	84/247/11K	122/294/16K	130/292/17K	135/280/18K	144/285/19K	131/282/17K	80/105/10K	71/112/9672	74/106/10K
	512	118/244/16K	84/247/11K	122/294/16K	130/292/17K	135/280/18K	145/284/19K	131/282/17K	88/106/17K	71/112/9752	74/106/10K
Tiger	192	27/13/1959	24/18/1778	27/27/2046	28/27/2015	29/27/2025	34/27/2449	26/26/1895	24/11/1912	21/14/1608	20/11/1449
BCHASH-Rijndael	128	45/13/1605	59/16/2085	131/19/4385	77/19/2579	76/20/2673	99/19/3445	87/19/2868	47/11/1650	70/14/2384	60/13/2008
	256	123/35/4118	146/45/4914	260/118/8815	152/120/5254	150/109/5288	276/114/9268	204/116/7000	149/40/4889	163/43/5432	156/41/5151

		Alpha cc	**Alpha gcc 2.97**	**Sun/Sparc V9 450MHz gcc 3.0.4+3.2.1**	**Sun/Sparc 248MHz gcc 2.95.3**	**Sun/Sparc V9 338MHz cc**	**Sun/Sparc V9 338MHz gcc 3.0.4**	**Sun/Sparc V9 400MHz cc**	**Sun/Sparc V9 400MHz gcc 3.0.4**	**Sun 450MHz cc**	**Sun 333MHz gcc 3.2.1**
Whirlpool	512	36/14/2485	37/15/2567	171/22/11K	184/31/12K	114/87/7281	183/32/12K	109/82/7000	173/30/11K	124/89/8371	176/28/11K
2nd impl (MMX)	512	84/15/5476	35/15/2400	143/24/9450	146/33/9711	102/88/6552	149/33/9842	97/82/6266	142/32/9551	91/91/6541	149/32/9808
MD4	128	7.1/12/585	6.3/11/467	7.1/11/526	7.6/11/529	7.1/10/526	7.3/11/542	6.8/10/515	7.1/11/528	7.2/16/531	7.1/11/536
MD5	128	10/12/812	10.0/11/691	10/11/795	10/11/754	10/10/767	10/11/787	10/10/746	10/11/750	10/17/782	10/11/763
RIPEMD	160	25/12/1817	24/11/1697	24/15/1751	25/15/1776	27/12/1868	25/13/1798	26/12/1811	24/13/1758	27/17/1839	24/13/1797
SHA-0	160	11/12/859	9.8/11/653	13/16/952	14/15/915	12/12/886	13/13/979	12/12/859	13/13/952	13/17/853	15/13/1015
SHA-1	160	11/10/883	11/11/745	13/13/1061	15/15/1211	14/12/1021	13/13/1119	14/12/955	13/13/1096	16/14/1095	13/13/1103
SHA-2	256	31/10/2117	29/19/2021	36/46/2411	37/49/2520	35/149/2834	37/47/2492	34/137/2725	36/45/2409	34/153/2745	37/34/2467
	384	17/30/2357	16/46/2333	100/118/13K	98/135/12K	72/283/9842	102/219/13K	69/268/9541	100/213/13K	65/287/9018	107/212/14K
	512	17/30/2374	16/46/2348	100/117/13K	98/137/12K	71/284/9863	102/220/13K	69/273/9560	100/213/13K	65/288/9027	160/324/21K
Tiger	192	5.4/10/407	5.2/10/390	28/17/1682	23/17/1551	28/14/1787	26/16/1696	26/14/1691	25/16/1643	32/18/1990	46/21/2871
BCHASH-Rijndael	128	33/8.1/1179	49/13/1676	65/82/2289	73/84/2596	63/98/2063	78/100/2701	61/83/1997	66/88/2383	54/100/2007	65/94/2340
	256	97/10/3204	–/–/–	221/133/7417	224/133/7688	182/150/6011	229/150/7635	171/134/5583	223/133/7456	187/152/6248	237/149/7935

The performances given are those for hash/initialization/initialization+finalization, where the hash time is measured in cycles/byte and the initialization and initialization+finalization times are measured in cycles.

Table 48. Message Authentication Codes Performance Results by Processor and Compiler

Primitive	Key	MAC	Platform, Processor and Compiler									
Name	Size	Size	PIII/Linux Coppermine gcc 2.95.2	PIII/Linux Coppermine gcc 3.1.1	PIII/Linux Coppermine gcc 3.0.3	PIII/Linux Tualatin gcc 2.96	PIII/Linux Katmai gcc 3.1.1	PIII/Linux Katmai gcc 3.3	PIII/Linux Coppermine gcc-egcs	PIII/Win Visual C 6.0	PIII/Win Intel C	PIII/Win gcc 2.95.3
Ttmac	160	160	21/30/1564	22/29/1577	23/29/1660	26/40/1842	22/27/1573	22/30/1585	21/32/1599	29/34/1976	19/21/1304	21/30/1576
Umac-16	128	64	6.2/52K/53K	6.0/56K/57K	6.2/55K/56K	6.4/53K/55K	–/–/–	6.0/57K/58K	6.6/57K/58K	5.8/71K/72K	6.2/52K/53K	6.1/52K/53K
Umac-32	128	64	3.1/55K/56K	3.0/59K/60K	3.1/59K/60K	3.2/54K/55K	–/–/–	2.9/59K/60K	3.3/60K/61K	2.7/64K/69K	3.2/53K/55K	3.2/54K/55K
HMAC-Whirlpool	512	512	72/5078/24K	74/5195/24K	72/5129/24K	78/6544/27K	83/5935/28K	86/6076/29K	77/5381/25K	81/5875/27K	90/6407/30K	73/5163/24K
HMAC-MD4	512	128	4.7/638/1991	4.8/663/1976	4.7/676/2020	4.8/1753/3175	4.7/693/2024	4.7/684/2028	4.7/642/1961	4.8/867/2247	4.5/640/1881	4.7/656/1985
HMAC-MD5	512	128	7.3/804/2677	7.3/838/2653	7.5/853/2699	7.5/1940/3873	7.3/866/2685	7.5/868/2750	7.3/808/2634	7.2/1018/2833	6.8/799/2470	7.3/828/2655
HMAC-RIPE-MD	512	160	23/1863/6910	18/1571/5608	22/1781/6451	24/2992/8141	18/1594/5623	18/1593/5646	22/1782/6526	23/2069/7100	16/1440/5045	24/1907/6987
HMAC-SHA-0	512	160	20/1695/5925	16/1380/4697	15/1398/4619	16/2530/5960	15/1413/4936	16/1428/4971	16/1393/5159	19/1788/5394	12/1184/3905	16/1414/4828
HMAC-SHA-1	512	160	15/1390/5237	15/1346/4697	16/1422/4786	17/2550/6145	15/1367/4825	15/1362/4825	15/1376/5200	19/1829/5597	13/1211/4029	16/1437/5207
HMAC-SHA-2	512	256	42/3087/12K	40/2958/11K	40/3018/11K	44/4304/13K	43/3378/12K	39/2899/11K	41/3106/11K	56/4188/15K	40/3020/11K	40/2967/11K
	512	384	84/626/34K	87/638/35K	87/641/35K	89/1706/37K	87/702/35K	104/694/42K	111/625/44K	80/745/32K	75/730/30K	84/599/34K
	512	512	84/615/34K	88/629/35K	87/638/35K	89/1699/37K	87/700/35K	104/694/42K	111/614/44K	80/744/32K	75/727/30K	84/584/34K
HMAC-Tiger	512	192	24/1867/7116	21/1766/6655	21/1758/6729	21/2881/7700	21/1787/6664	21/1789/6526	25/1965/7251	25/2214/7803	25/1974/7210	24/1924/7117
CBCMAC-Rijndael	128	128	29/1024/3151	28/616/2084	26/623/2056	28/880/2654	28/744/2308	28/752/2347	29/1082/3174	24/548/1919	33/595/2425	29/1007/2946
CBCMAC-DES	64	64	61/1000/3012	62/973/2924	61/973/2956	64/1131/3330	61/981/2931	62/1009/2998	62/1047/3099	74/980/3178	70/988/3099	62/1005/3017
CBCMAC-Shacal	512	160	31/829/2886	36/840/3043	32/856/2936	34/1090/3531	35/1027/3150	36/1017/3060	33/957/3143	46/1097/3978	28/856/2745	31/846/2916

			PII/Win Visual C 6.0	PII/Win gcc 2.95.3	PIV/Linux Northwood gcc 2.95.2	PIV/Linux Northwood gcc 3.1.1	Xeon Model 1 gcc 3.2	Xeon Model 0 egcs 2.91.66	Xeon Model 0 gcc 2.96	AMD Duron Win XP VC 7.0 (.NET)	AMD Duron Win XP gcc 2.95.3	AMD Athlon Linux gcc 2.96
Ttmac	160	160	28/32/1959	21/30/1582	41/57/2894	40/50/2675	39/47/2671	47/55/3254	37/53/2614	25/14/1754	21/21/1526	22/20/1596
Umac-16	128	64	6.4/73K/69K	6.1/53K/54K	6.4/76K/77K	6.2/79K/81K	6.1/76K/78K	6.5/80K/82K	7.2/85K/88K	7.7/74K/75K	6.2/76K/77K	6.9/74K/75K
Umac-32	128	64	2.5/69K/70K	3.1/54K/55K	6.7/76K/77K	6.7/79K/81K	6.6/78K/81K	7.0/81K/82K	7.4/86K/89K	2.2/70K/72K	1.9/77K/78K	1.9/75K/76K
HMAC-Whirlpool	512	512	116/8120/39K	86/6351/29K	103/7466/34K	98/7161/33K	106/7592/36K	103/7563/35K	104/7654/35K	104/7232/34K	102/6908/33K	100/6810/32K
HMAC-MD4	512	128	4.8/885/2295	4.7/682/2013	6.4/807/2529	6.5/770/2396	6.4/786/2385	6.4/836/2490	6.5/772/2397	4.8/821/2056	4.7/637/1944	4.7/609/1880
HMAC-MD5	512	128	7.2/1032/2889	7.4/853/2705	9.4/998/3305	9.5/963/3186	9.4/987/3175	9.5/1027/3288	9.5/968/3187	7.4/985/2736	8.0/848/2782	8.0/817/2709
HMAC-RIPE-MD	512	160	23/2054/7060	24/1952/7071	47/3385/12K	27/2083/7656	26/2095/7682	35/2753/10K	40/3036/11K	21/1928/6556	22/1812/6673	23/1793/6657
HMAC-SHA-0	512	160	19/1798/5442	16/1454/4955	30/2413/8320	23/1915/7065	23/1922/7099	27/2230/7969	25/2041/6723	16/2206/6517	13/1254/4033	13/1196/3830
HMAC-SHA-1	512	160	20/1849/5661	16/1445/5255	30/2497/8701	25/1987/6702	24/1989/6722	26/2157/8421	26/2106/6883	14/1687/5695	13/1204/4141	12/1162/3947
HMAC-SHA-2	512	256	56/4172/15K	40/3017/11K	51/3927/15K	40/3129/12K	39/3186/11K	53/4132/16K	48/3634/14K	47/3558/13K	33/2479/9469	34/2564/9602
	512	384	118/815/47K	84/660/34K	124/1041/50K	131/996/53K	136/829/55K	145/842/59K	132/806/53K	80/688/32K	72/512/29K	74/488/30K
	512	512	118/804/48K	84/654/34K	124/1041/51K	131/993/53K	136/827/55K	145/820/59K	132/801/53K	80/689/32K	72/504/29K	74/487/30K
HMAC-Tiger	512	192	27/2332/8184	24/1964/7182	28/2307/8489	29/2325/8515	28/2336/8540	34/2750/10K	26/2145/7836	24/2062/7188	22/1739/6523	20/1654/6037
CBCMAC-Rijndael	128	128	24/694/2102	29/1078/1661	40/1483/2789	26/958/3318	27/960/3407	35/1720/5347	27/1489/4432	33/513/1124	32/861/1512	31/819/2727
CBCMAC-DES	64	64	74/971/3175	62/973/1560	82/1009/1932	72/1061/3218	72/1036/3198	82/1175/3652	69/1041/3238	61/747/1301	54/766/1283	56/831/2581
CBCMAC-Shacal	512	160	46/1235/4065	31/877/1577	67/751/2473	86/864/5068	80/854/5037	90/1107/5932	74/828/4725	38/594/1438	31/535/1237	29/605/2372

			Alpha cc	Alpha gcc 2.97	Sun/Sparc V9 450MHz gcc 3.0.4+3.2.1	Sun/Sparc 248MHz gcc 2.95.3	Sun/Sparc V9 338MHz cc	Sun/Sparc V9 338MHz gcc 3.0.4	Sun/Sparc V9 400MHz cc	Sun/Sparc V9 400MHz gcc 3.0.4	Sun 450MHz cc	Sun 333MHz gcc 3.2.1
Ttmac	160	160	25/16/1691	28/35/1952	26/57/1796	27/52/1790	30/57/2067	27/58/1851	29/53/1993	26/57/1769	28/47/1903	26/61/1849
Umac-16	128	64	7.8/105K/107K	4.5/48K/49K	28/76K/79K	20/74K/76K	51/79K/83K	22/85K/89K	45/72K/74K	21/78K/81K	46/82K/86K	29/78K/80K
Umac-32	128	64	1.2/104K/106K	1.1/46K/47K	10/75K/78K	10/74K/76K	–/–/–	11/83K/85K	–/–/–	10/78K/80K	–/–/–	10/79K/80K
HMAC-Whirlpool	512	512	37/2488/13K	36/2710/12K	145/9890/48K	147/9982/49K	104/7148/34K	154/10K/51K	97/6698/31K	146/10K/48K	98/7279/34K	150/10K/50K
HMAC-MD4	512	128	7.1/603/2332	6.3/759/2186	7.1/934/2678	7.7/953/2657	7.0/972/2615	7.3/974/2763	6.9/936/2531	7.1/947/2714	7.3/979/2654	7.1/968/2768
HMAC-MD5	512	128	10/845/3258	10/990/3107	10/1197/3564	10/1159/3543	10/1189/3541	10/1247/3725	10/1150/3440	10/1215/3636	10/1195/3574	10/1224/3674
HMAC-RIPE-MD	512	160	25/1793/7189	24/1950/7028	24/2102/11K	25/2114/11K	27/2284/10K	25/2162/12K	26/2200/10K	24/2103/11K	27/2316/11K	24/2118/14K
HMAC-SHA-0	512	160	11/908/3581	9.8/981/3044	13/1439/4830	14/1458/4468	12/1297/3986	14/1480/5033	12/1267/3926	13/1432/4870	13/1280/4054	15/1601/5117
HMAC-SHA-1	512	160	11/934/3670	11/1073/3402	13/1448/4798	15/1500/5283	14/1408/4698	13/1495/4979	14/1361/4558	13/1453/4829	16/1443/5260	13/1482/4895
HMAC-SHA-2	512	256	31/2162/8462	29/2292/8364	36/2919/10K	37/2951/10K	35/3029/10K	37/3005/10K	34/2943/10K	36/2947/10K	34/2938/10K	37/2952/10K
	512	384	17/193/7280	16/412/7261	100/695/40K	98/686/39K	71/896/31K	103/804/41K	70/863/30K	100/785/40K	65/893/28K	105/788/42K
	512	512	17/195/7285	16/408/7268	100/693/40K	98/681/39K	71/880/31K	103/788/41K	69/835/30K	101/770/40K	65/873/28K	105/765/42K
HMAC-Tiger	512	192	5.4/500/1776	5.2/682/1933	28/2381/8132	24/2130/7050	28/2407/7963	26/2324/7628	27/2323/7749	25/2270/7407	33/2659/9073	31/2624/9143
CBCMAC-Rijndael	128	128	20/523/1636	21/812/2321	34/691/3592	34/839/3606	23/731/3151	36/699/3748	22/671/2979	32/676/3596	24/649/2958	33/727/3881
CBCMAC-DES	64	64	39/628/2154	46/638/2153	65/924/2991	66/969/3098	59/940/2843	70/971/3164	56/912/2723	68/934/3038	61/902/2946	69/937/3081
CBCMAC-Shacal	512	160	42/984/3575	37/1278/3953	36/1017/3629	36/1017/3588	40/1259/4027	37/1056/3837	39/1229/3904	36/1035/3633	34/1284/3762	37/1074/3851

The performances given are those for MAC/key setup/key setup+finalization, where the MAC time is measured in cycles/byte and the key setup and key setup+finalization times are measured in cycles.

Table 49. Asymmetric Encryption Performance Results by Processor and Compiler

Primitive	Key	Platform, Processor and Compiler									
Name	Size	PIII/Linux Coppermine gcc 2.95.2	PIII/Linux Coppermine gcc 3.1.1	PIII/Linux Coppermine gcc 3.0.3	PIII/Linux Tualatin gcc 2.96	PIII/Linux Katmai gcc 3.1.1	PIII/Linux Katmai gcc 3.3	PIII/Linux Coppermine gcc-egcs	PIII/Win Visual C 6.0	PIII/Win Intel C	PIII/Win gcc 2.95.3
EPOC-2	1152	31M/8224K/553M	31M/8262K/532M	31M/8255K/532M	31M/8076K/577M	31M/8246K/366M	31M/8306K/534M	30M/7989K/457M	–/–/–	–/–/–	30M/8024K/569M
RSA-OAEP	1024	2642K/57M/2415M	2058K/42M/1806M	2336K/48M/1654M	2964K/60M/2522M	2026K/42M/1736M	2087K/43M/1828M	2768K/57M/2460M	2588K/52M/2214M	2621K/56M/2272M	2583K/53M/2234M
ACE encrypt		62M/26M/3306M	63M/26M/3321M	60M/25M/3195M	57M/25M/3194M	–/–/–	–/–/–	–/–/–	–/–/–	–/–/–	–/–/–
EPOC1		6803K/11M/48M	6914K/11M/49M	6547K/11M/47M	6312K/11M/46M	–/–/–	–/–/–	–/–/–	–/–/–	–/–/–	6433K/11M/47M
EPOC2		8782K/12M/48M	8815K/12M/48M	8779K/12M/47M	8254K/11M/46M	–/–/–	–/–/–	–/–/–	–/–/–	–/–/–	8477K/12M/48M
EPOC3		8763K/5230K/48M	8936K/5403K/49M	8815K/5250K/48M	8279K/4882K/46M	–/–/–	–/–/–	–/–/–	–/–/–	–/–/–	8554K/5158K/48M
PSEC1		8876K/7240K/2747	8321K/6839K/2472	8387K/6693K/2306	7952K/6322K/2422	–/–/–	–/–/–	–/–/–	–/–/–	–/–/–	9703K/7341K/–
PSEC2		8654K/6897K/2724	8174K/6557K/2480	8194K/6417K/2380	7703K/6176K/2424	–/–/–	–/–/–	–/–/–	–/–/–	–/–/–	–/–/–
PSEC3		8838K/81K/2686	8291K/78K/2534	8339K/79K/2370	7952K/76K/2483	–/–/–	–/–/–	–/–/–	–/–/–	–/–/–	–/–/–
		PII/Win Visual C 6.0	PII/Win gcc 2.95.3	PIV/Linux Northwood gcc 2.95.2	PIV/Linux Northwood gcc 3.1.1	Xeon Model 1 gcc 3.2	Xeon Model 0 egcs 2.91.66	Xeon Model 0 gcc 2.96	AMD Duron Win XP VC 7.0 (.NET)	AMD Duron Win XP gcc 2.95.3	AMD Athlon Linux gcc 2.96
EPOC-2	1152	–/–/–	25M/6235K/389M	84M/18M/1152M	83M/18M/1150M	23M/5866K/357M	83M/18M/1141M	83M/18M/1211M	–/–/–	18M/4654K/299M	22M/6066K/402M
RSA-OAEP	1024	2576K/53M/2003M	2572K/53M/1935M	3521K/75M/3141M	3028K/62M/2616M	2981K/61M/2587M	3835K/79M/3336M	3246K/67M/2792M	2468K/49M/2073M	2383K/48M/2024M	2289K/48M/2027M
ACE encrypt		–/–/–	–/–/–	63M/27M/3426M	64M/27M/3416M	61M/26M/3237M	51M/20M/3402M	62M/26M/3314M	–/–/–	–/–/–	33M/14M/1748M
EPOC1		–/–/–	7599K/9853K/59M	18M/30M/124M	19M/30M/123M	19M/30M/122M	19M/30M/123M	18M/30M/122M	–/–/–	–/–/–	4155K/6930K/31M
EPOC2		–/–/–	8538K/10M/47M	24M/33M/123M	24M/32M/123M	23M/32M/122M	24M/33M/124M	24M/32M/122M	–/–/–	–/–/–	5823K/7270K/31M
EPOC3		–/–/–	7981K/3895K/43M	24M/11M/123M	23M/11M/123M	23M/11M/122M	24M/11M/123M	23M/11M/121M	–/–/–	–/–/–	5808K/2728K/31M
PSEC1		–/–/–	19M/8373K/5295	14M/11M/5729	13M/10M/4776	12M/10M/4696	14M/11M/5021	13M/10M/4842	–/–/–	–/–/–	7480K/6087K/2993
PSEC2		–/–/–	10M/8197K/4534	14M/10M/5727	12M/10M/4710	12M/10M/4780	14M/10M/4840	13M/9959K/4538	–/–/–	–/–/–	7301K/5757K/2992
PSEC3		–/–/–	10M/72K/4546	14M/128K/5784	13M/108K/4716	13M/111K/4561	14M/120K/7003	13M/112K/4471	–/–/–	–/–/–	7495K/76K/2918
		Alpha cc	Alpha gcc 2.97	Sun/Sparc V9 450MHz gcc 3.0.4+3.2.1	Sun/Sparc 248MHz gcc 2.95.3	Sun/Sparc V9 338MHz cc	Sun/Sparc V9 338MHz gcc 3.0.4	Sun/Sparc V9 400MHz cc	Sun/Sparc V9 400MHz gcc 3.0.4	Sun 450MHz cc	Sun 333MHz gcc 3.2.1
EPOC-2	1152	–/–/–	–/–/–	–/–/–	–/–/–	–/–/–	–/–/–	–/–/–	–/–/–	–/–/–	–/–/–
RSA-OAEP	1024	–/–/–	–/–/–	3892K/77M/3224M	3927K/77M/3225M	7131K/138M/5634M	4005K/80M/3281M	6853K/133M/5431M	3872K/77M/3106M	6198K/120M/5012M	3932K/78M/3280M
ACE encrypt		–/–/–	–/–/–	272M/114M/14G	–/–/–	304M/128M/16G	277M/117M/14G	283M/119M/14G	268M/113M/13G	–/–/–	275M/115M/14G
EPOC1		1584K/2834K/13M	1343K/2376K/10M	–/–/–	–/–/–	70M/114M/455M	–/–/–	66M/106M/421M	–/–/–	–/–/–	–/–/–
EPOC2		2467K/3001K/13M	2112K/2461K/10M	–/–/–	–/–/–	81M/123M/455M	–/–/–	75M/114M/418M	–/–/–	–/–/–	–/–/–
EPOC3		2456K/1282K/12M	2089K/1067K/10M	–/–/–	–/–/–	79M/43M/453M	–/–/–	73M/40M/418M	–/–/–	–/–/–	–/–/–
PSEC1		8303K/7438K/4116	7136K/6280K/3612	–/–/–	–/–/–	25M/23M/5756	–/–/–	22M/20M/5017	–/–/–	–/–/–	–/–/–
PSEC2		8238K/7383K/3857	7040K/6197K/3742	–/–/–	–/–/–	24M/22M/5746	–/–/–	22M/20M/5061	–/–/–	–/–/–	–/–/–
PSEC3		8348K/40K/4094	7147K/49K/3838	–/–/–	–/–/–	25M/86K/5537	–/–/–	22M/73K/4980	–/–/–	–/–/–	–/–/–

The performances given are those for encryption/decryption/key generation. All of them are measured in cycles.

Table 50. Signature Performance Results by Processor and Compiler

Primitive	Key	Platform, Processor and Compiler									
Name	Size	PIII/Linux Coppermine gcc 2.95.2	PIII/Linux Coppermine gcc 3.1.1	PIII/Linux Coppermine gcc 3.0.3	PIII/Linux Tualatin gcc 2.96	PIII/Linux Katmai gcc 3.1.1	PIII/Linux Katmai gcc 3.3	PIII/Linux Coppermine gcc-egcs	PIII/Win Visual C 6.0	PIII/Win Intel C	PIII/Win gcc 2.95.3
ECDSA-GF(2^p)	160	5326K/6808K/5215K	4968K/6387K/4860K	4775K/6085K/4669K	6344K/8047K/6243K	4871K/6359K/4779K	5277K/6810K/5181K	5947K/7575K/5832K	4639K/5950K/4494K	4689K/6005K/4662K	5413K/6984K/5330K
ECDSA-GF(2^{163})	163	5061K/6809K/4852K	5504K/7471K/5320K	5411K/7351K/5190K	6052K/8166K/5842K	5502K/7396K/5276K	5502K/7396K/5294K	5405K/7243K/5221K	5315K/7031K/5096K	5051K/6652K/4938K	5124K/6831K/4932K
KCDSA-GF(2^p)	160	5314K/6869K/5275K	4935K/6326K/4889K	4724K/6085K/4688K	6319K/8116K/6275K	4871K/6359K/4843K	5277K/6855K/5239K	5907K/7575K/5867K	4596K/5869K/4524K	4656K/5971K/4701K	5405K/6950K/5367K
KCDSA-GF(2^{163})	163	5024K/6692K/4888K	5462K/7291K/5342K	5387K/7291K/5248K	5953K/7902K/5774K	5457K/7261K/5304K	5502K/7216K/5331K	5362K/7176K/5252K	5281K/7005K/5127K	5034K/6716K/4977K	5090K/6831K/4962K
Esign	1152	671K/185K/458M	647K/178K/401M	648K/167K/368M	644K/246K/519M	682K/177K/452M	658K/221K/406M	586K/198K/497M	806K/209K/582M	705K/169K/568M	604K/187K/460M
RSA-PSS	1024	57M/2663K/1798M	43M/2060K/1343M	48M/2359K/1506M	60M/2961K/1874M	42M/2029K/1334M	43M/2090K/1354M	57M/2761K/1825M	52M/2581K/1443M	54M/2585K/1483M	53M/2562K/1982M
SFLASHv2		1555K/384K/1192M	1403K/348K/845M	−/−/−	1753K/328K/1060M	−/−/−	1576K/399K/916M	1385K/395K/731M	−/−/−	−/−/−	−/−/−
ACE sign		27M/19M/8201M	24M/19M/7313M	24M/19M/7617M	25M/20M/7436M	25M/19M/9800M	−/−/−	−/−/−	−/−/−	−/−/−	−/−/−
ECDSA		1320K/3456K/1116K	952K/2540K/805K	1008K/2701K/857K	983K/2617K/824K	958K/2538K/807K	−/−/−	−/−/−	−/−/−	−/−/−	1211K/3207K/1023K
Esign (e=8)	1152	2340K/405K/112M	2104K/395K/107M	2285K/393K/105M	2046K/370K/104M	2112K/401K/106M	−/−/−	−/−/−	−/−/−	−/−/−	2099K/370K/113M
FLASH		2251K/505K/2335M	2050K/443K/1869M	2411K/529K/2181M	2110K/291K/1092M	2423K/665K/1960M	2429K/668K/1967M	2844K/296K/1211M	−/−/−	−/−/−	2721K/308K/1395M
QUARTZ		6881M/129K/1788M	6072M/121K/1622M	6690M/126K/1652M	5500M/122K/1555M	7271M/154K/1781M	6954M/152K/1769M	7426M/142K/1901M	−/−/−	−/−/−	6104M/115K/1826M
SFLASH		1924K/274K/1153M	1724K/195K/793M	2084K/216K/1056M	1700K/207K/889M	1743K/207K/834M	1747K/208K/816M	2365K/284K/786M	−/−/−	−/−/−	2303K/259K/952M
RSA-PSS		64M/1252K/2440M	57M/1090K/2188M	61M/1180K/2357M	60M/981K/2317M	57M/1098K/2183M	56M/1101K/2182M	69M/1366K/2629M	−/−/−	−/−/−	62M/1208K/2388M

		PII/Win Visual C 6.0	PII/Win gcc 2.95.3	PIV/Linux Northwood gcc 2.95.2	PIV/Linux Northwood gcc 3.1.1	Xeon Model 1 gcc 3.2	Xeon Model 0 egcs 2.91.66	Xeon Model 0 gcc 2.96	AMD Duron Win XP VC 7.0 (.NET)	AMD Duron Win XP gcc 2.95.3	AMD Athlon Linux gcc 2.96
ECDSA-GF(2^p)	160	4620K/5950K/4536K	5530K/7087K/5411K	6332K/8190K/6172K	6100K/7862K/5936K	6281K/8098K/6080K	7513K/9656K/7452K	6261K/8261K/6150K	4025K/5251K/3909K	4464K/5762K/4354K	4612K/5940K/4505K
ECDSA-GF(2^{163})	163	5390K/7105K/5316K	5162K/6912K/4970K	5922K/7981K/5669K	6218K/8379K/5936K	6161K/8217K/5840K	6437K/8673K/6181K	6538K/8732K/6227K	4578K/6153K/4363K	4602K/6159K/4395K	5272K/7106K/5027K
KCDSA-GF(2^p)	160	4410K/5740K/4582K	5477K/7052K/5444K	6359K/8262K/6245K	6024K/7769K/5984K	6139K/7993K/6129K	7412K/9538K/7533K	6219K/8093K/6188K	4013K/5149K/3933K	4440K/5720K/4406K	4557K/5984K/4533K
KCDSA-GF(2^{163})	163	5355K/7105K/5348K	5141K/6912K/5020K	5939K/7854K/5707K	6108K/8116K/5973K	6094K/8068K/5892K	6345K/8505K/6230K	6454K/8572K/6290K	4530K/6068K/4400K	4572K/6135K/4448K	5236K/6967K/5087K
Esign	1152	807K/208K/584M	525K/190K/400M	1049K/281K/579M	996K/253K/545M	527K/259K/483M	1166K/312K/694M	957K/259K/618M	767K/206K/504M	405K/148K/293M	519K/154K/401M
RSA-PSS	1024	53M/2584K/1438M	53M/2562K/1983M	74M/3524K/2337M	62M/3019K/1937M	62M/2979K/1929M	78M/3815K/2462M	67M/3274K/2103M	49M/2454K/1353M	48M/2351K/1793M	48M/2288K/1502M
SFLASHv2		1837K/363K/734M	1761K/344K/1237M	−/−/−	−/−/−	1299K/269K/802M	1455K/299K/737M	−/−/−	2030K/406K/1020M	2121K/387K/1498M	2214K/385K/1449M
ACE sign		−/−/−	−/−/−	30M/20M/10G	26M/20M/9645M	25M/19M/8790M	28M/19M/10G	26M/19M/9822M	−/−/−	−/−/−	14M/7113K/5485M
ECDSA		−/−/−	1236K/3251K/1042K	2077K/5780K/1833K	1971K/5415K/1758K	1463K/3870K/1241K	2117K/5923K/1895K	1753K/4807K/1565K	−/−/−	1205K/3263K/1030K	997K/2663K/841K
Esign (e=8)	1152	−/−/−	2074K/296K/95M	4726K/954K/269M	4434K/936K/270M	4107K/938K/265M	4846K/951K/268M	4487K/939K/266M	−/−/−	−/−/−	1818K/275K/74M
FLASH		−/−/−	7067K/−/2106M	2882K/288K/1584M	2406K/280K/1116M	2336K/244K/1239M	3000K/293K/1137M	2294K/295K/1129M	−/−/−	3758K/412K/1612M	3296K/443K/1456M
QUARTZ		−/−/−	12G/220K/4257M	7378M/152K/3248M	6261M/144K/3167M	7214M/136K/2823M	10G/161K/3553M	7856M/147K/3281M	−/−/−	12G/123K/2133M	6406M/128K/2028M
SFLASH		−/−/−	2330K/269K/943M	2824K/431K/1499M	2387K/395K/1192M	2336K/337K/1202M	2842K/388K/1395M	2308K/363K/1132M	−/−/−	3332K/310K/1165M	2910K/212K/1016M
RSA-PSS		−/−/−	115M/1255K/3412M	84M/1615K/3243M	82M/1587K/3206M	76M/1473K/2967M	84M/697K/3670M	83M/1487K/3183M	−/−/−	57M/1171K/2202M	53M/1381K/2078M

		Alpha cc	Alpha gcc 2.97	Sun/Sparc V9 450MHz gcc 3.0.4+3.2.1	Sun/Sparc 248MHz gcc 2.95.3	Sun/Sparc V9 338MHz cc	Sun/Sparc V9 338MHz gcc 3.0.4	Sun/Sparc V9 400MHz cc	Sun/Sparc V9 400MHz gcc 3.0.4	Sun 450MHz cc	Sun 333MHz gcc 3.2.1
ECDSA-GF(2^p)	160	2523K/3212K/2466K	3646K/4702K/3574K	7515K/9540K/7417K	6745K/8680K/6656K	7706K/9734K/7597K	7571K/9497K/7436K	7480K/9520K/7362K	7280K/9320K/7207K	8100K/9990K/7805K	7592K/9657K/7522K
ECDSA-GF(2^{163})	163	3590K/4791K/3465K	4629K/6225K/4446K	7020K/9450K/6790K	4811K/6522K/4640K	5002K/6760K/4806K	7334K/9768K/7068K	4920K/6560K/4690K	7040K/9400K/6848K	4815K/6435K/4631K	7459K/10M/7289K
KCDSA-GF(2^p)	160	2512K/3184K/2482K	3646K/4679K/3615K	7515K/9540K/7462K	6770K/8630K/6744K	7807K/10M/7830K	−/−/−	7600K/9640K/7584K	7360K/9400K/7348K	7920K/10M/7907K	−/−/−
KCDSA-GF(2^{163})	163	3562K/4674K/3470K	4585K/6041K/4484K	7335K/9675K/7192K	4786K/6324K/4673K	5577K/7300K/5249K	−/−/−	4920K/6600K/4814K	7080K/9360K/6937K	4680K/6300K/4617K	−/−/−
Esign	1152	416K/85K/174M	700K/156K/381M	1053K/322K/562M	1035K/300K/536M	1270K/359K/605M	1090K/338K/575M	1246K/347K/591M	1050K/325K/560M	1257K/368K/623M	1073K/328K/578M
RSA-PSS	1024	−/−/−	−/−/−	77M/3858K/2994M	77M/3943K/3049M	136M/7098K/5351M	80M/3988K/−	132M/6840K/5166M	77M/3520K/3036M	120M/6120K/4707M	79M/3962K/3095M
SFLASHv2		1353K/246K/773M	1628K/143K/540M	1885K/312K/972M	2026K/357K/958M	2332K/413K/1042M	2009K/344K/1053M	2100K/364K/928M	1926K/324K/1001M	2139K/383K/942M	2034K/373K/1019M
ACE sign		−/−/−	−/−/−	95M/85M/25G	−/−/−	110M/95M/32G	97M/87M/26G	112M/90M/31G	102M/84M/26G	−/−/−	96M/85M/26G
ECDSA		−/−/386K	−/−/533K	−/−/−	−/−/−	912K/2347K/749K	1423K/3934K/1272K	809K/2083K/670K	1372K/3785K/1232K	−/−/−	−/−/−
Esign (e=8)	1152	1044K/128K/41M	978K/113K/32M	−/−/−	−/−/−	5578K/3218K/945M	−/−/−	5069K/2965K/872M	−/−/−	−/−/−	−/−/−
FLASH		1506K/242K/769M	1531K/197K/732M	3134K/394K/1089M	−/−/−	3358K/436K/1140M	3236K/409K/1370M	3135K/410K/1058M	3132K/394K/1330M	−/−/−	3374K/443K/1188M
QUARTZ		2855M/180K/1111M	2930M/129K/1131M	13G/191K/5958M	−/−/−	6246M/237K/5132M	14G/236K/6086M	5760M/224K/4703M	14G/227K/5893M	−/−/−	15G/225K/5926M
SFLASH		1309K/206K/558M	1324K/164K/465M	2767K/574K/3753M	−/−/−	3207K/671K/4388M	2894K/640K/4190M	2885K/627K/4085M	2784K/626K/3960M	−/−/−	2824K/567K/3930M
RSA-PSS		40M/794K/1558M	40M/779K/1560M	132M/2591K/5075M	−/−/−	147M/2424K/5609M	145M/2682K/5618M	134M/2619K/4968M	142M/2757K/5438M	−/−/−	133M/2642K/5110M

The performances given are those for signature/verification/key generation. All of them are measured in cycles.

Table 51. Identification Performance Results by Processor and Compiler

Primitive	Key	Platform, Processor and Compiler									
Name	Size	PIII/Linux Coppermine gcc 2.95.2	PIII/Linux Coppermine gcc 3.1.1	PIII/Linux Coppermine gcc 3.0.3	PIII/Linux Tualatin gcc 2.96	PIII/Linux Katmai gcc 3.1.1	PIII/Linux Katmai gcc 3.3	PIII/Linux Coppermine gcc-egcs	PIII/Win Visual C 6.0	PIII/Win Intel C	PIII/Win gcc 2.95.3
GPS		320M	322M	321M	312M	320M	322M	321M	–	–	326M
		3580K	3597K	3597K	3510K	3590K	3603K	2864K	–	–	2860K
		5689K	5730K	5730K	5574K	5714K	5737K	4691K	–	–	4707K
		328	337	337	339	333	338	338	–	–	297
		6467K	6495K	6495K	6367K	6472K	6512K	5475K	–	–	5482K

		PII/Win Visual C 6.0	PII/Win gcc 2.95.3	PIV/Linux Northwood gcc 2.95.2	PIV/Linux Northwood gcc 3.1.1	Xeon Model 1 gcc 3.2	Xeon Model 0 egcs 2.91.66	Xeon Model 0 gcc 2.96	AMD Duron Win XP VC 7.0 (.NET)	AMD Duron Win XP gcc 2.95.3	AMD Athlon Linux gcc 2.96
GPS		–	249M	760M	757M	236M	751M	749M	–	183M	237M
		–	2899K	9235K	9201K	2676K	9177K	9143K	–	2138K	2596K
		–	4571K	14M	14M	4202K	14M	14M	–	3340K	4092K
		–	309	508	491	343	470	437	–	241	322
		–	5194K	16M	16M	4755K	16M	16M	–	3796K	4634K

		Alpha cc	Alpha gcc 2.97	Sun/Sparc V9 450MHz gcc 3.0.4+3.2.1	Sun/Sparc 248MHz gcc 2.95.3	Sun/Sparc V9 338MHz cc	Sun/Sparc V9 338MHz gcc 3.0.4	Sun/Sparc V9 400MHz cc	Sun/Sparc V9 400MHz gcc 3.0.4	Sun 450MHz cc	Sun 333MHz gcc 3.2.1
GPS		–	71M	–	–	–	–	–	–	–	–
		–	811K	–	–	–	–	–	–	–	–
		–	1056K	–	–	–	–	–	–	–	–
		–	144	–	–	–	–	–	–	–	–
		–	1200K	–	–	–	–	–	–	–	–

The performances given are those for parameter generation/key generation/commitment/answer/verification. All of them are measured in cycles.

B. Figures

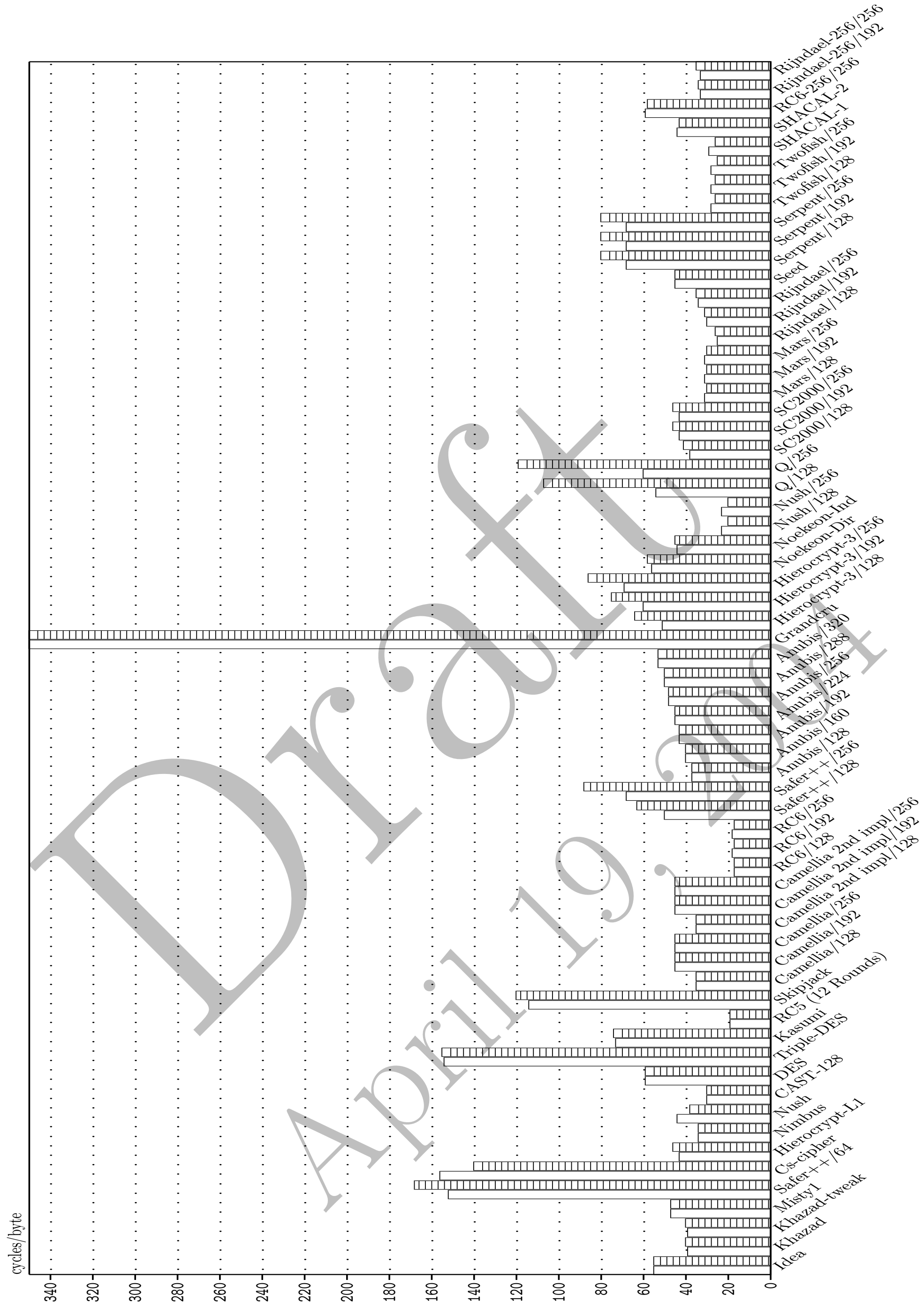


Fig. 2. Block Ciphers on PIII/Linux

□ encryption ▤ decryption

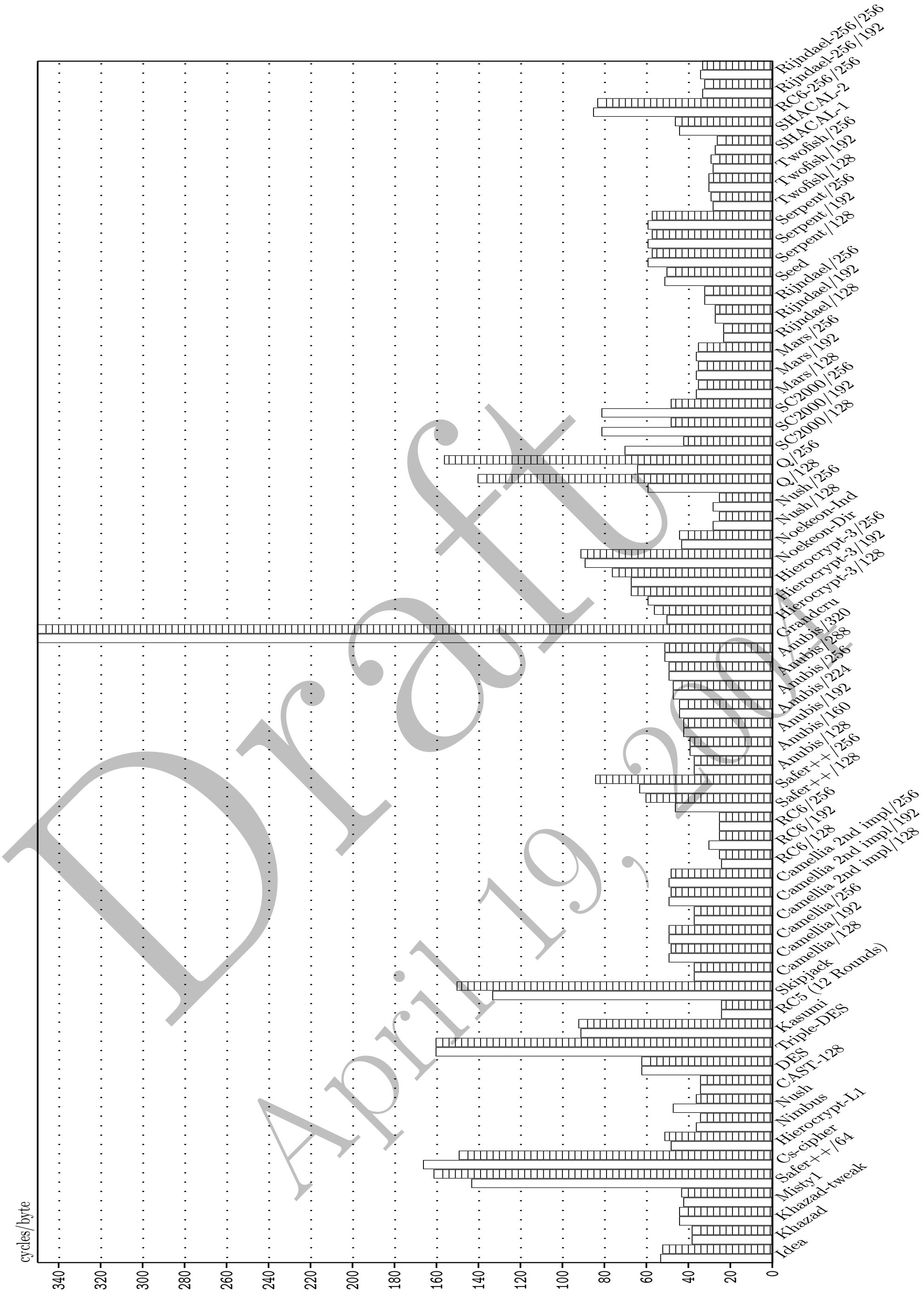


Fig. 3. Block Ciphers on PIII/MS

□ encryption ▤ decryption

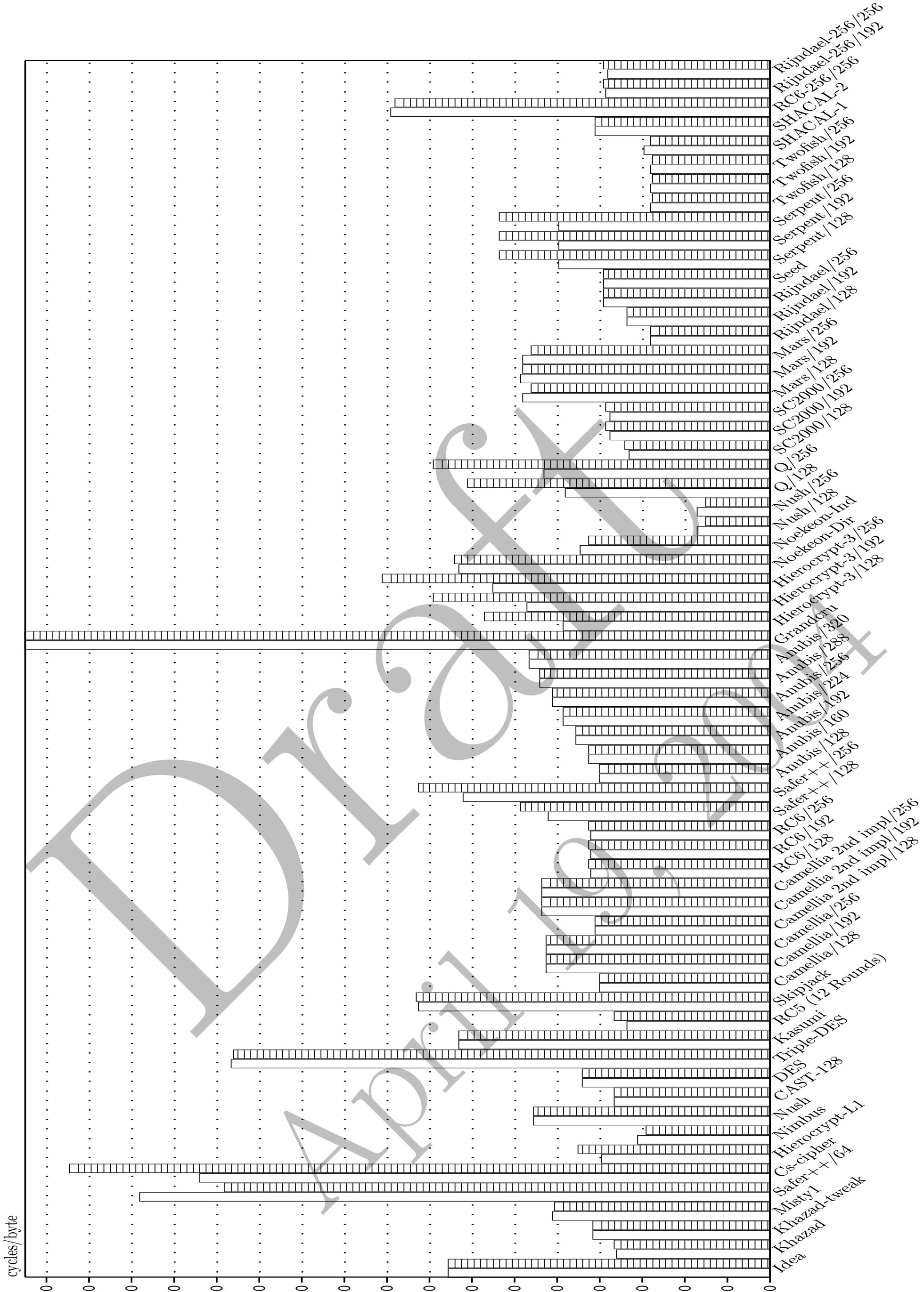


Fig. 4. Block Ciphers on PI/MMX

□ encryption ▤ decryption

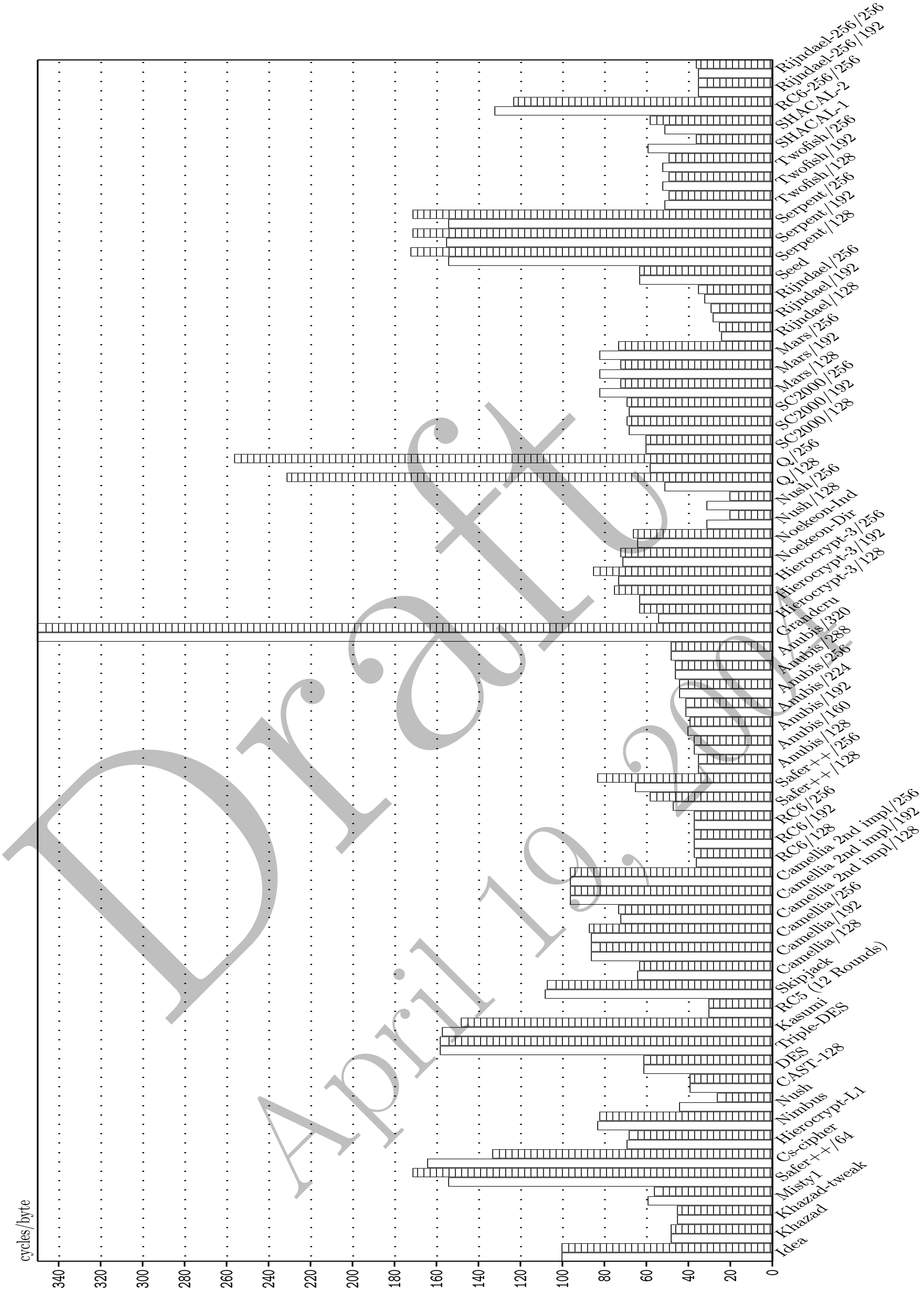


Fig. 5. Block Ciphers on Pentium4

□ encryption ▤ decryption

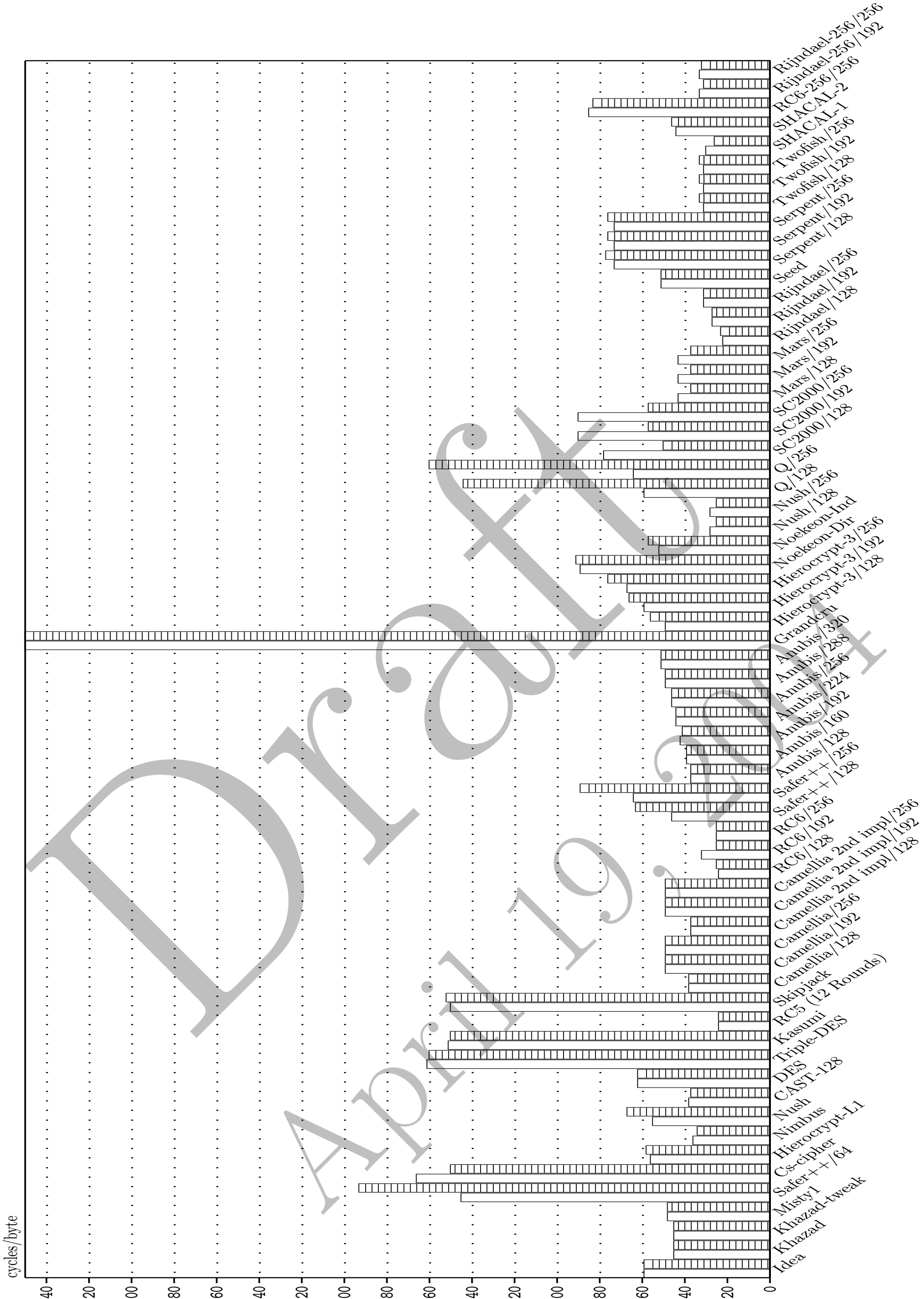


Fig. 6. Block Ciphers on Pentium2

□ encryption ▤ decryption

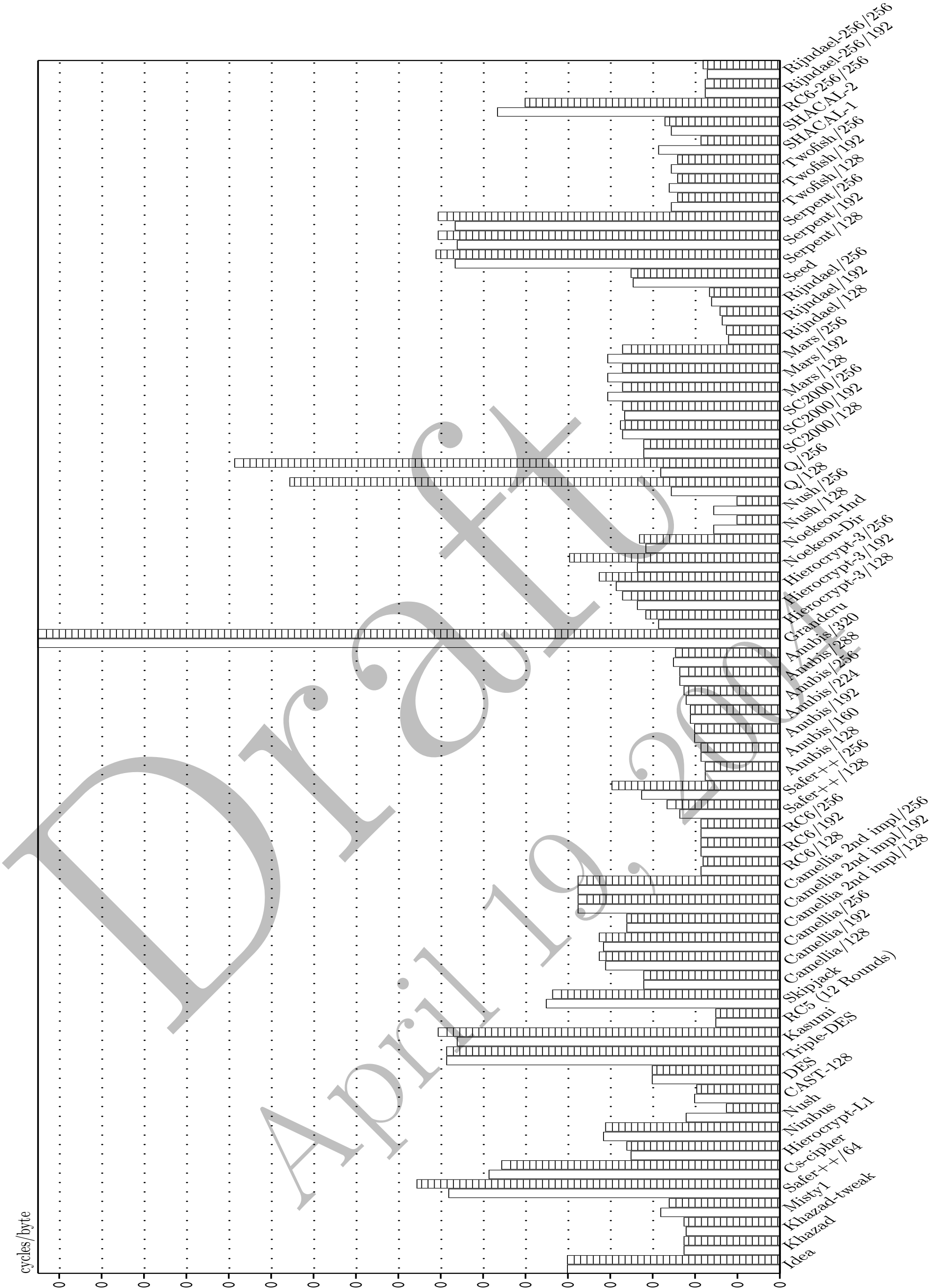


Fig. 7. Block Ciphers on Xeon

□ encryption ▤ decryption

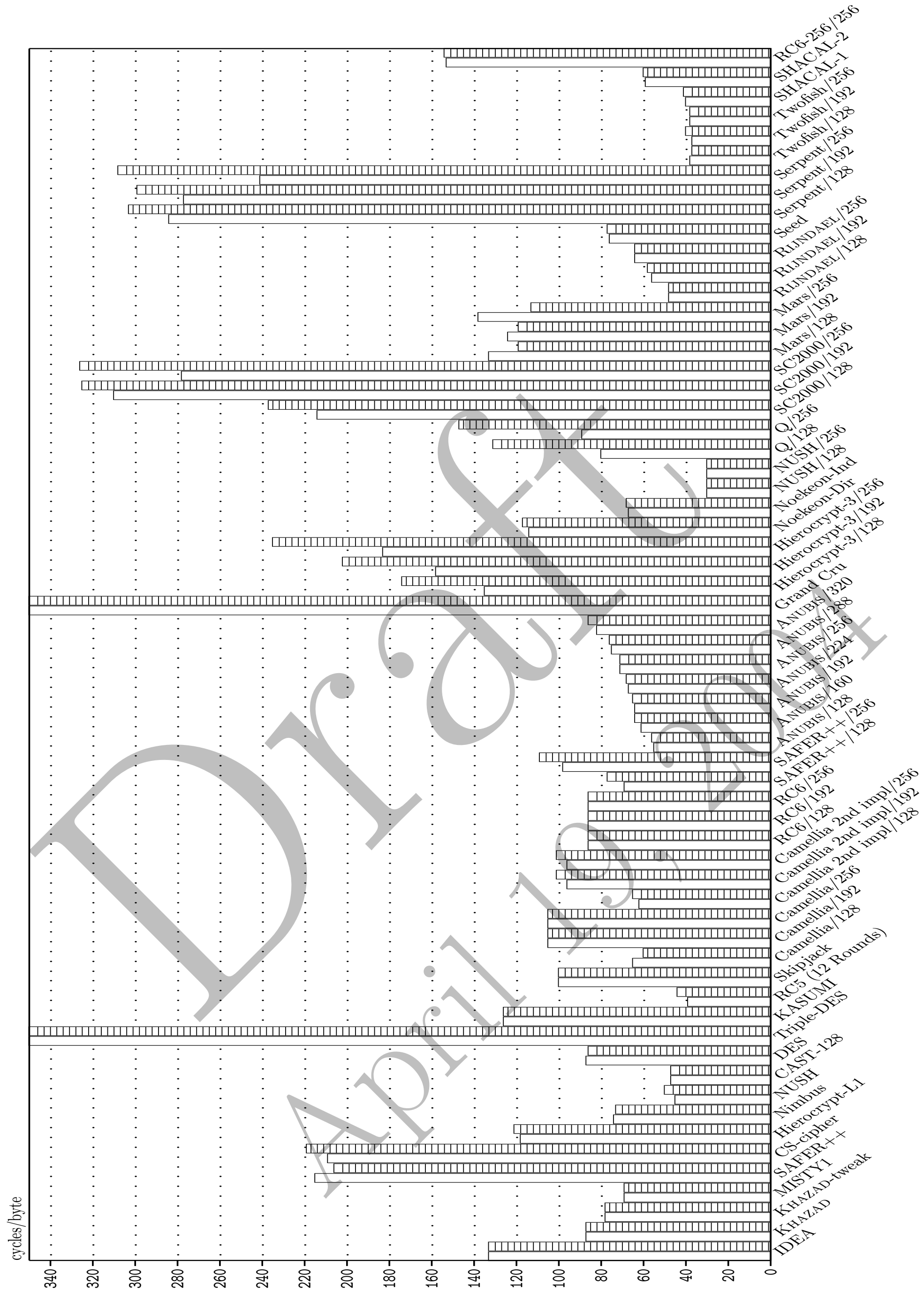


Fig. 8. Block Ciphers on 486

□ encryption ▤ decryption

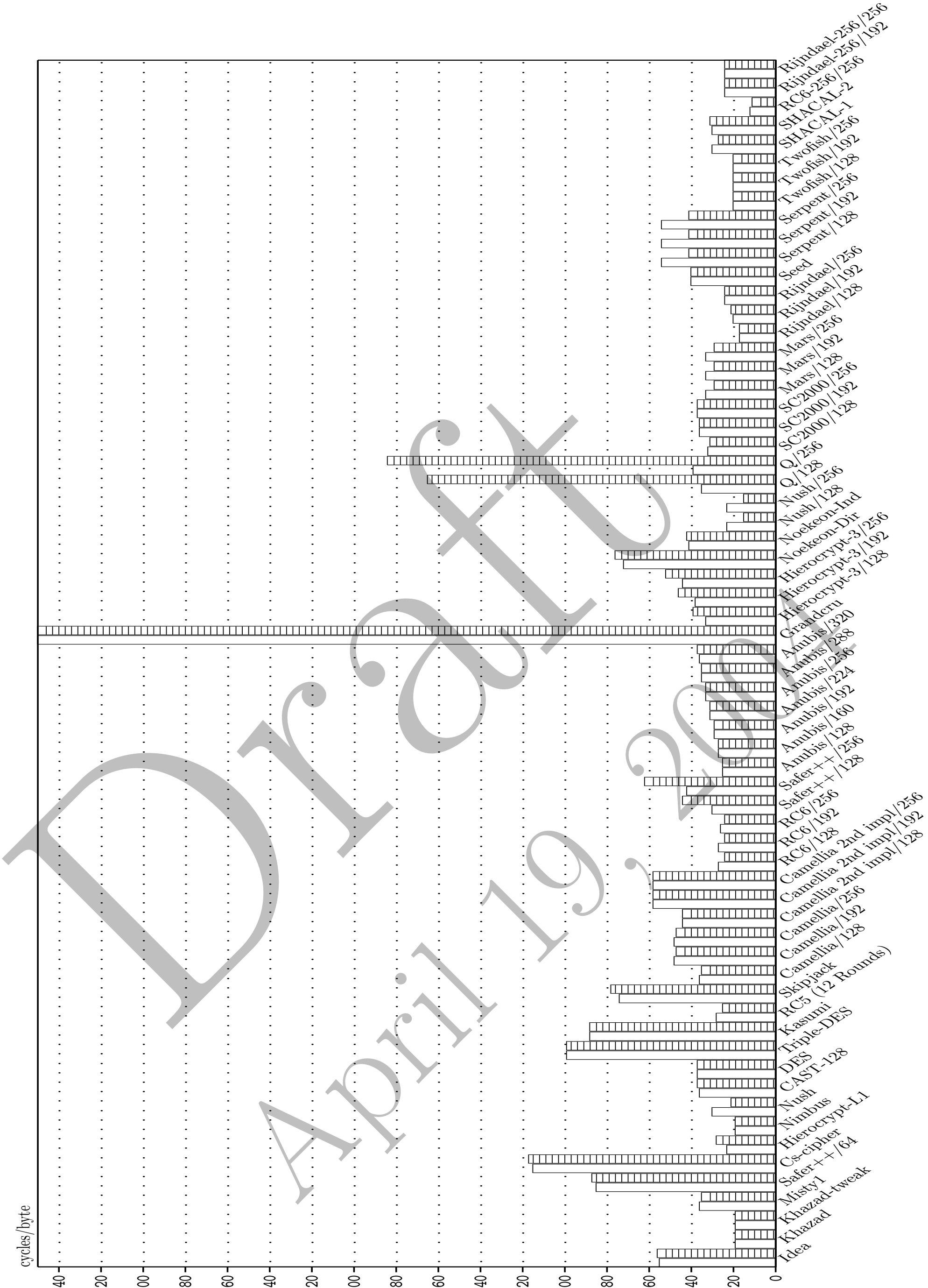


Fig. 9. Block Ciphers on Alpha

□ encryption ▤ decryption

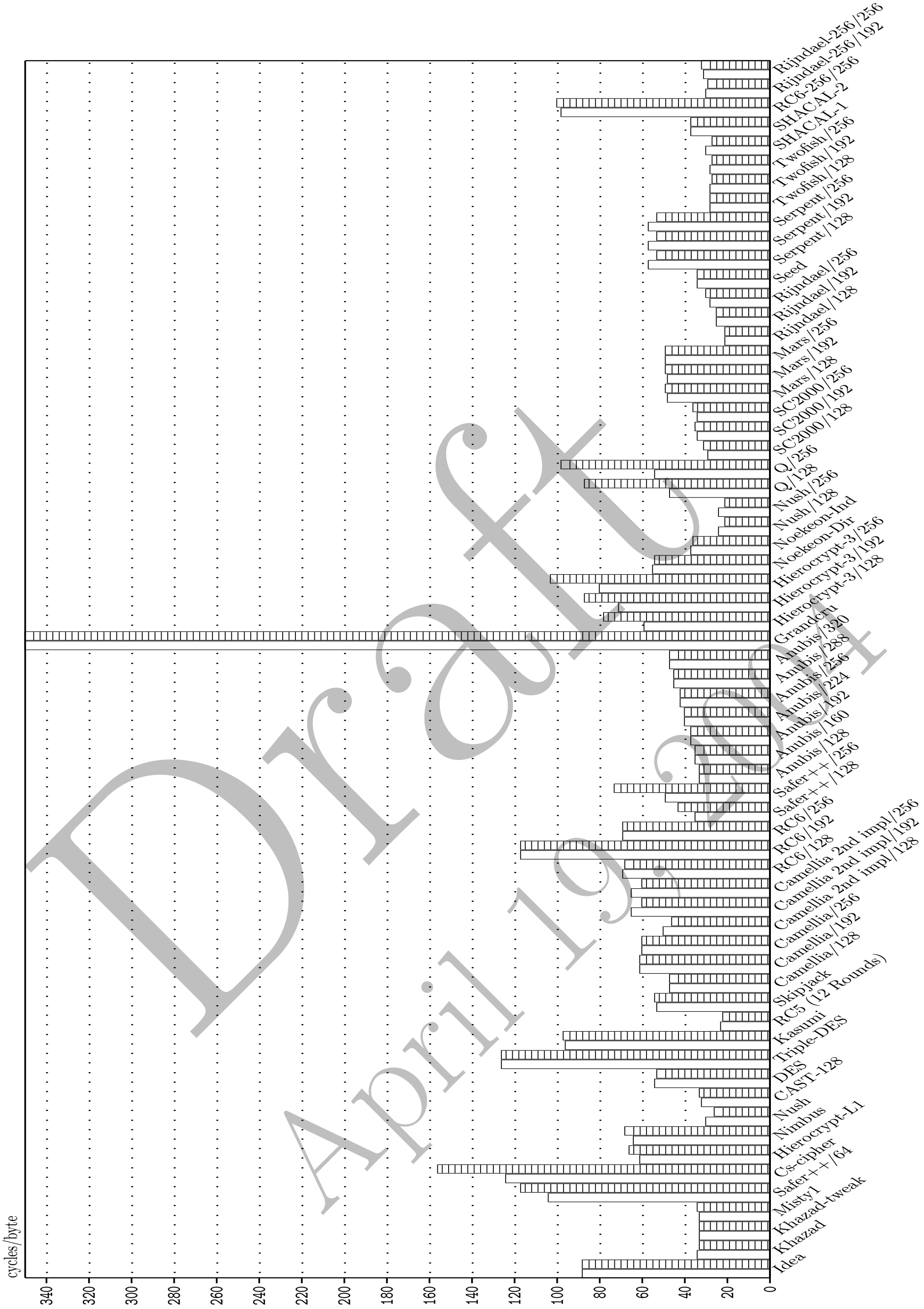


Fig. 10. Block Ciphers on Sparc V9

□ encryption ▤ decryption

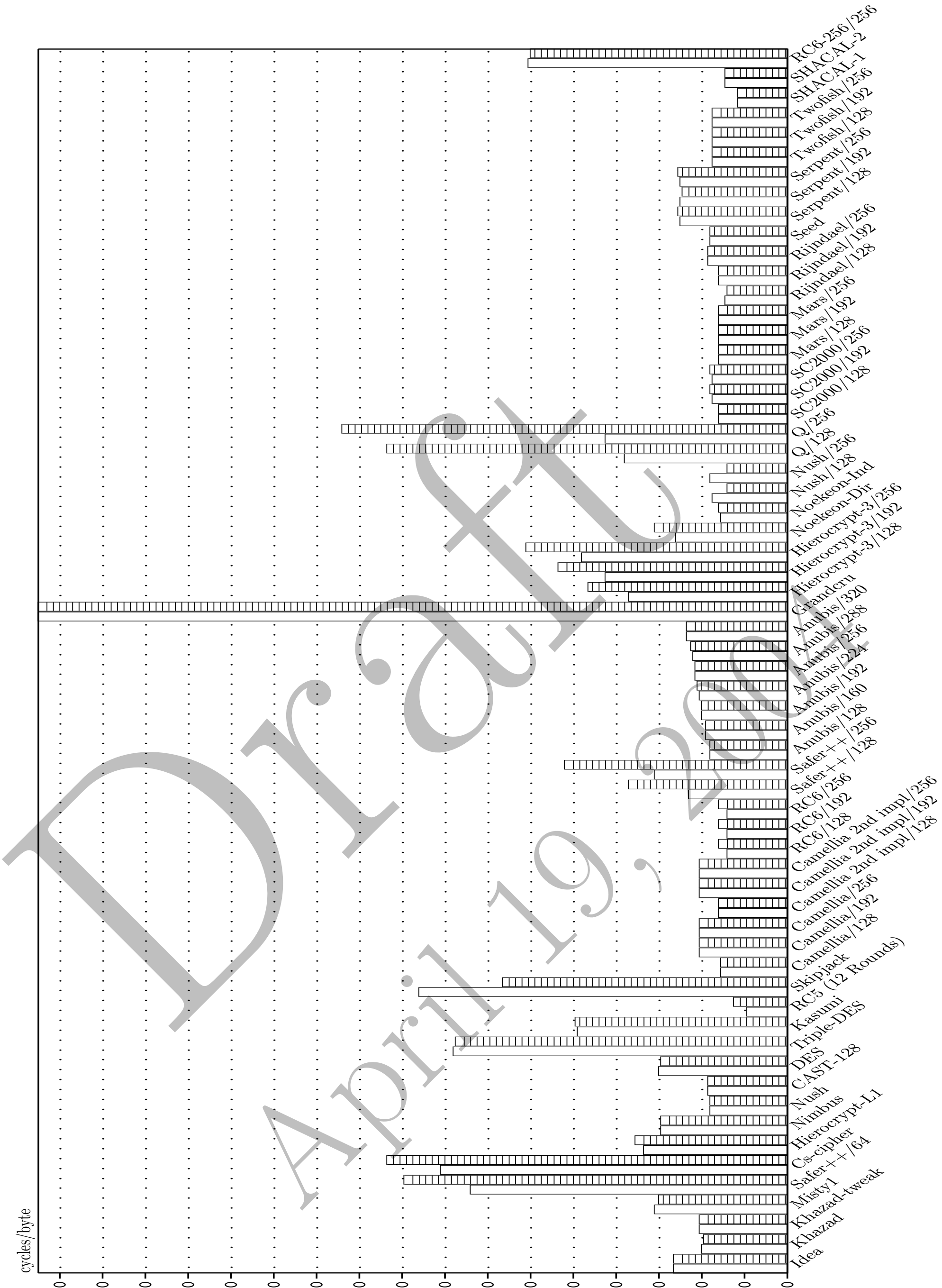


Fig. 11. Block Ciphers on Macintosh

□ encryption ▤ decryption

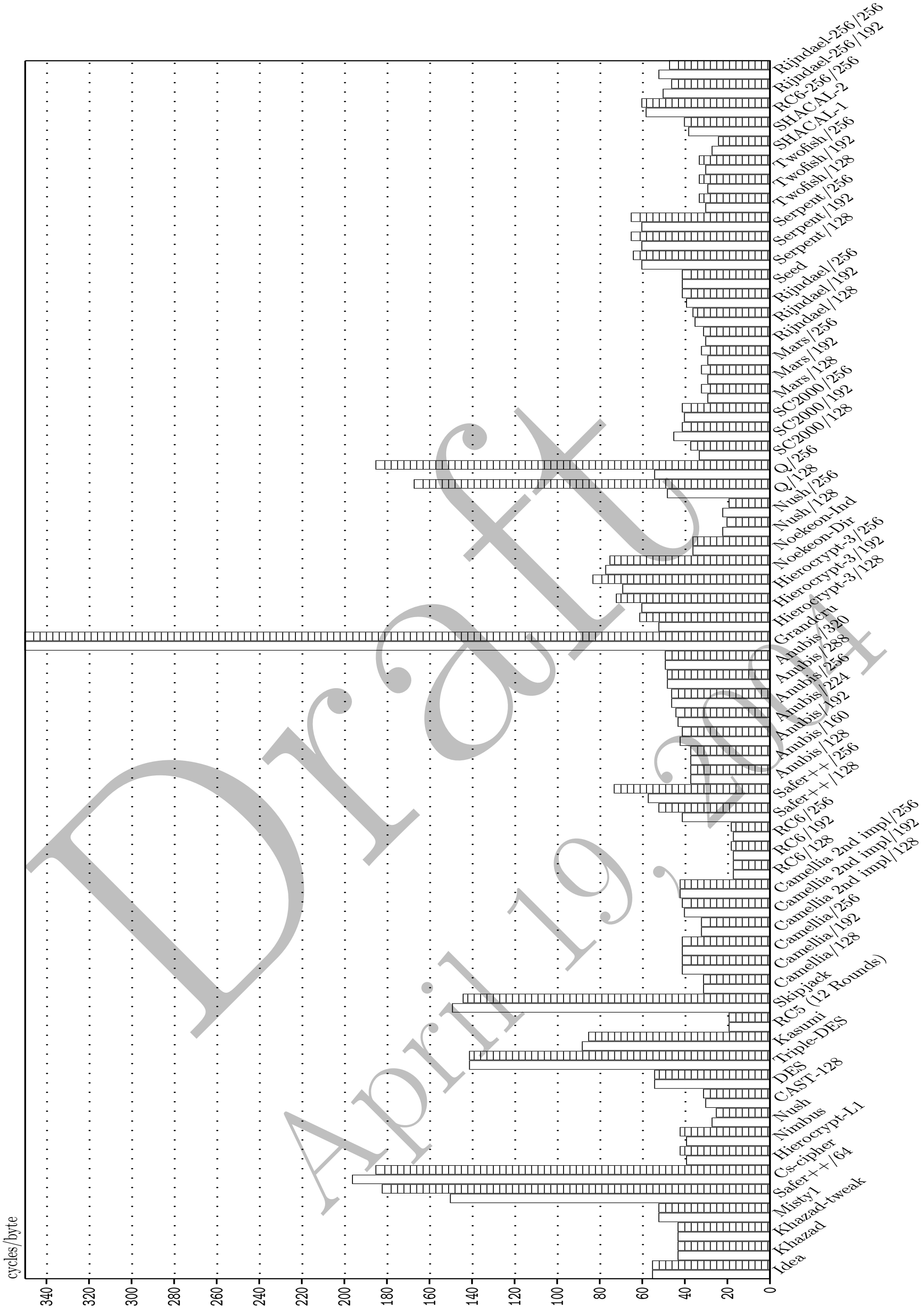


Fig. 12. Block Ciphers on AMD

□ encryption ▤ decryption

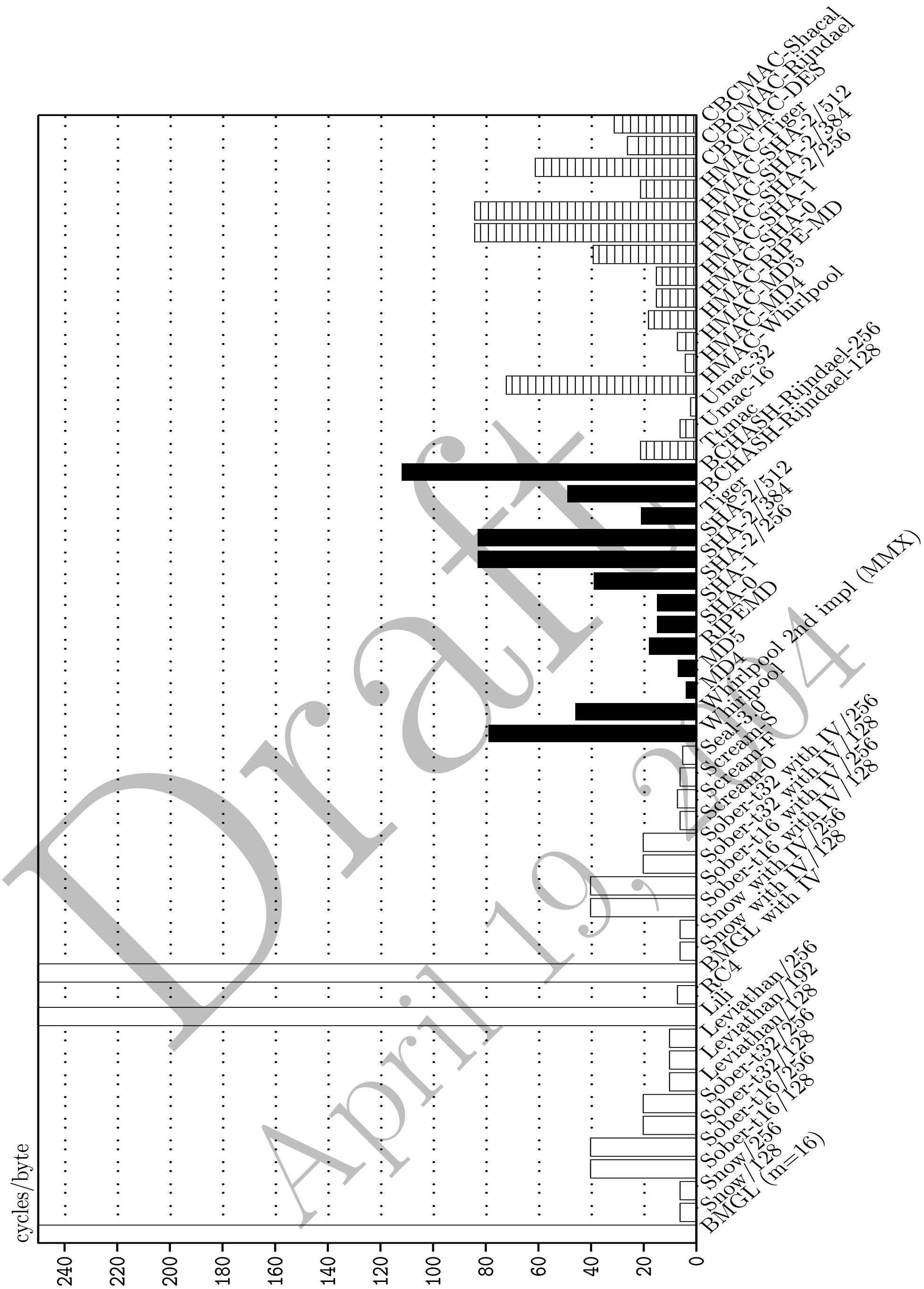


Fig. 13. Stream Ciphers, Hash Functions and MACs on PIII/Linux

□ stream ciphers ■ hash functions ▤ MACs

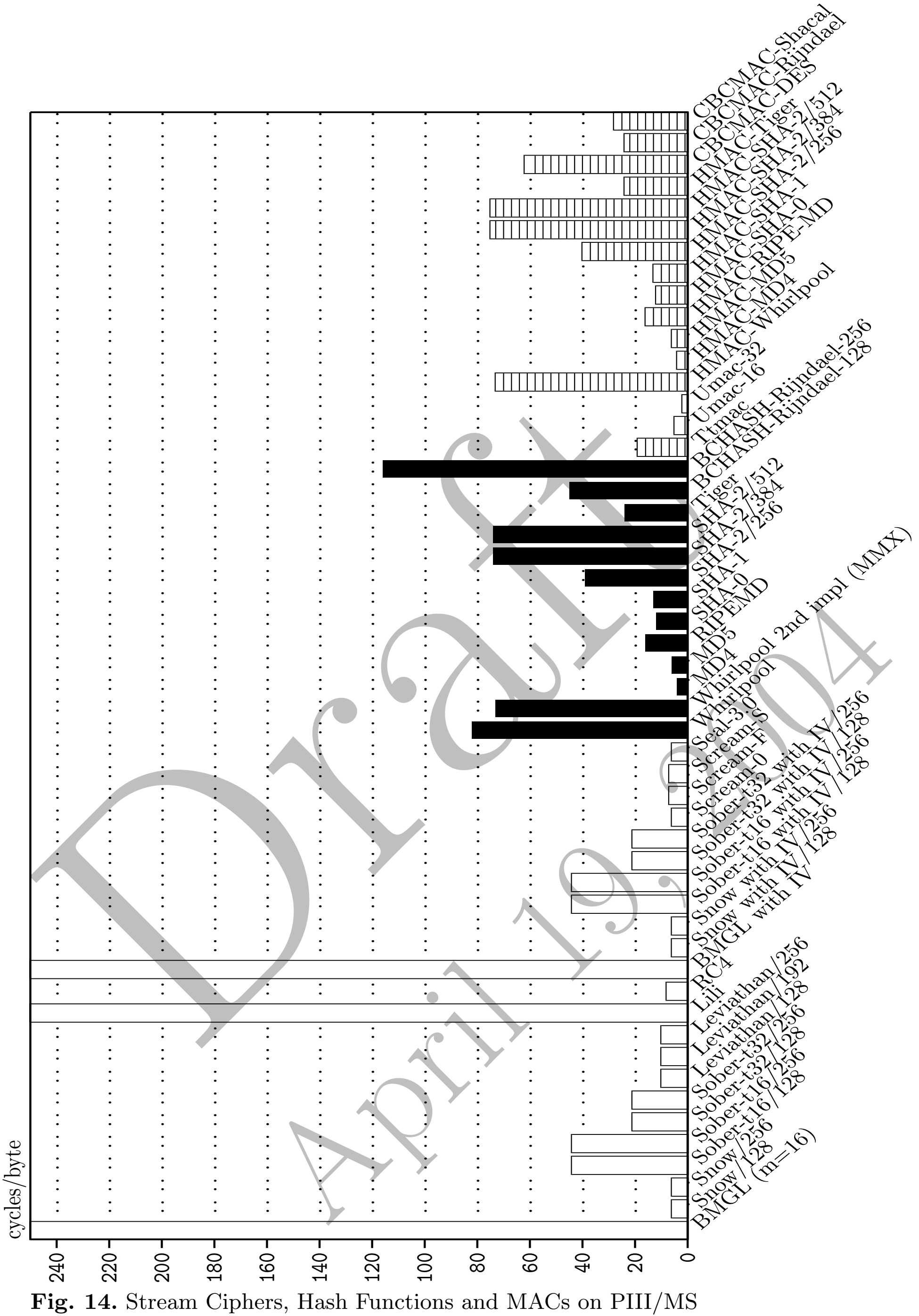


Fig. 14. Stream Ciphers, Hash Functions and MACs on PIII/MS

□ stream ciphers ■ hash functions ▤ MACs

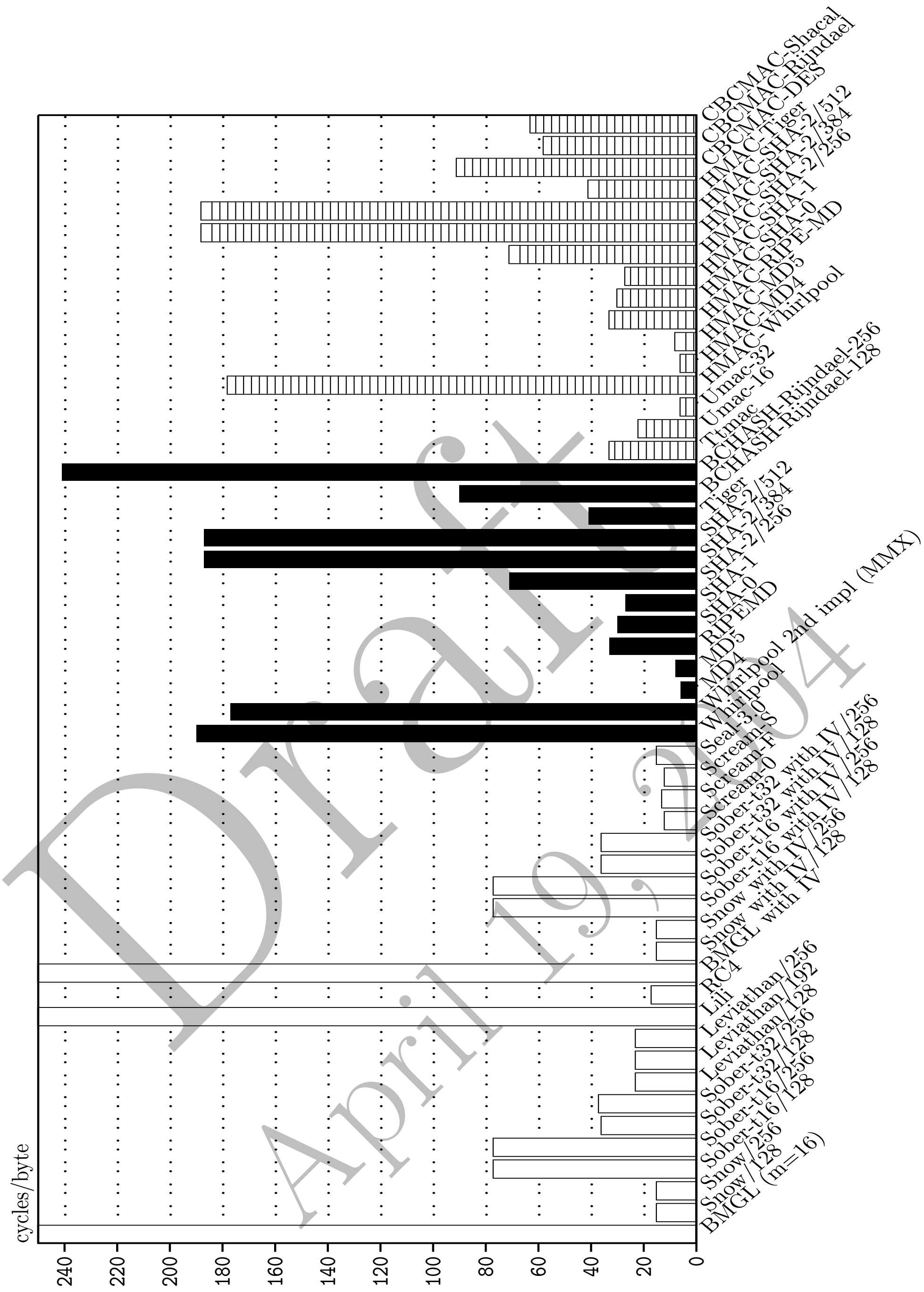


Fig. 15. Stream Ciphers, Hash Functions and MACs on PIII/MMX

☐ stream ciphers ■ hash functions ▤ MACs

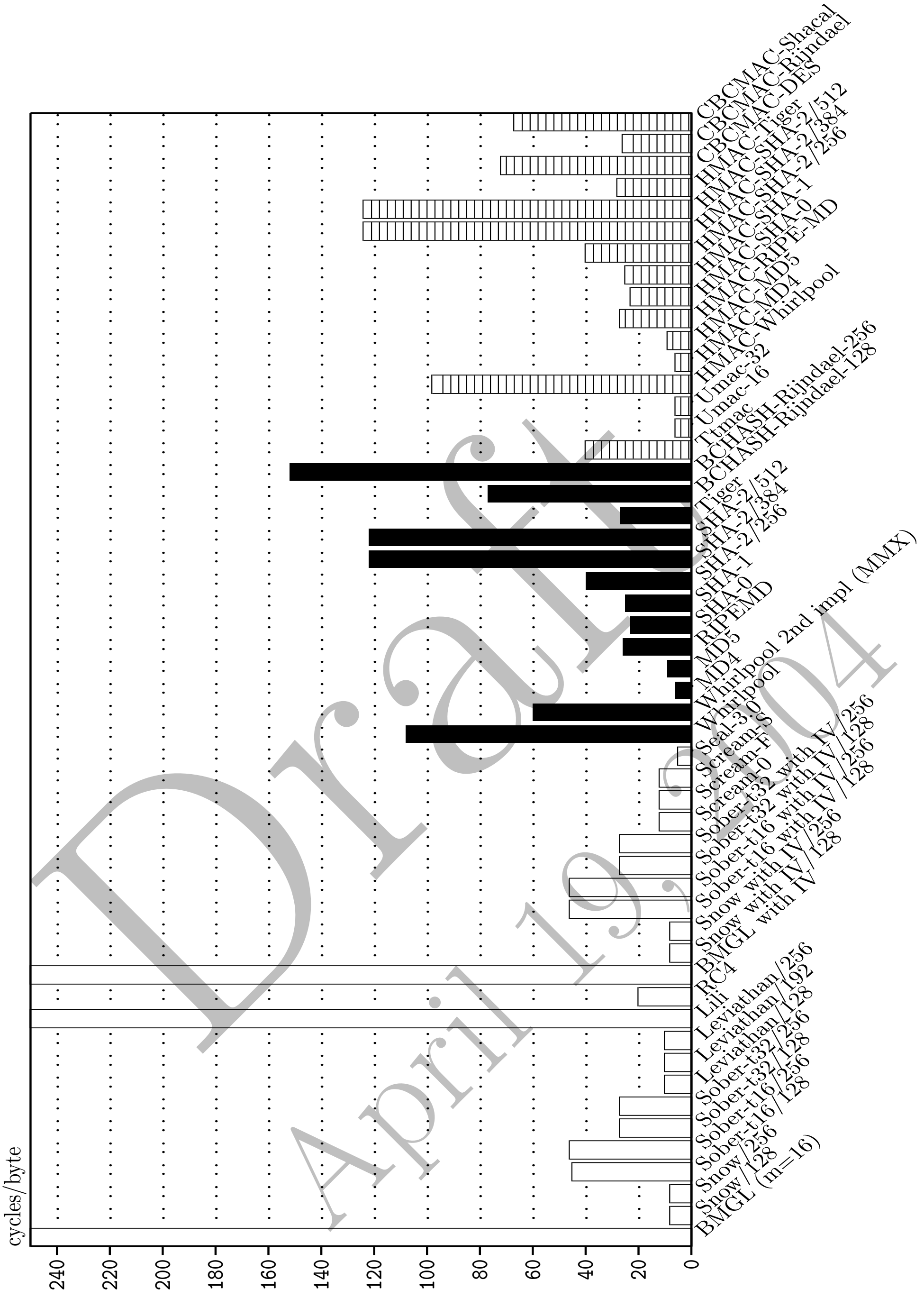


Fig. 16. Stream Ciphers, Hash Functions and MACs on Pentium4

□ stream ciphers ■ hash functions ▤ MACs

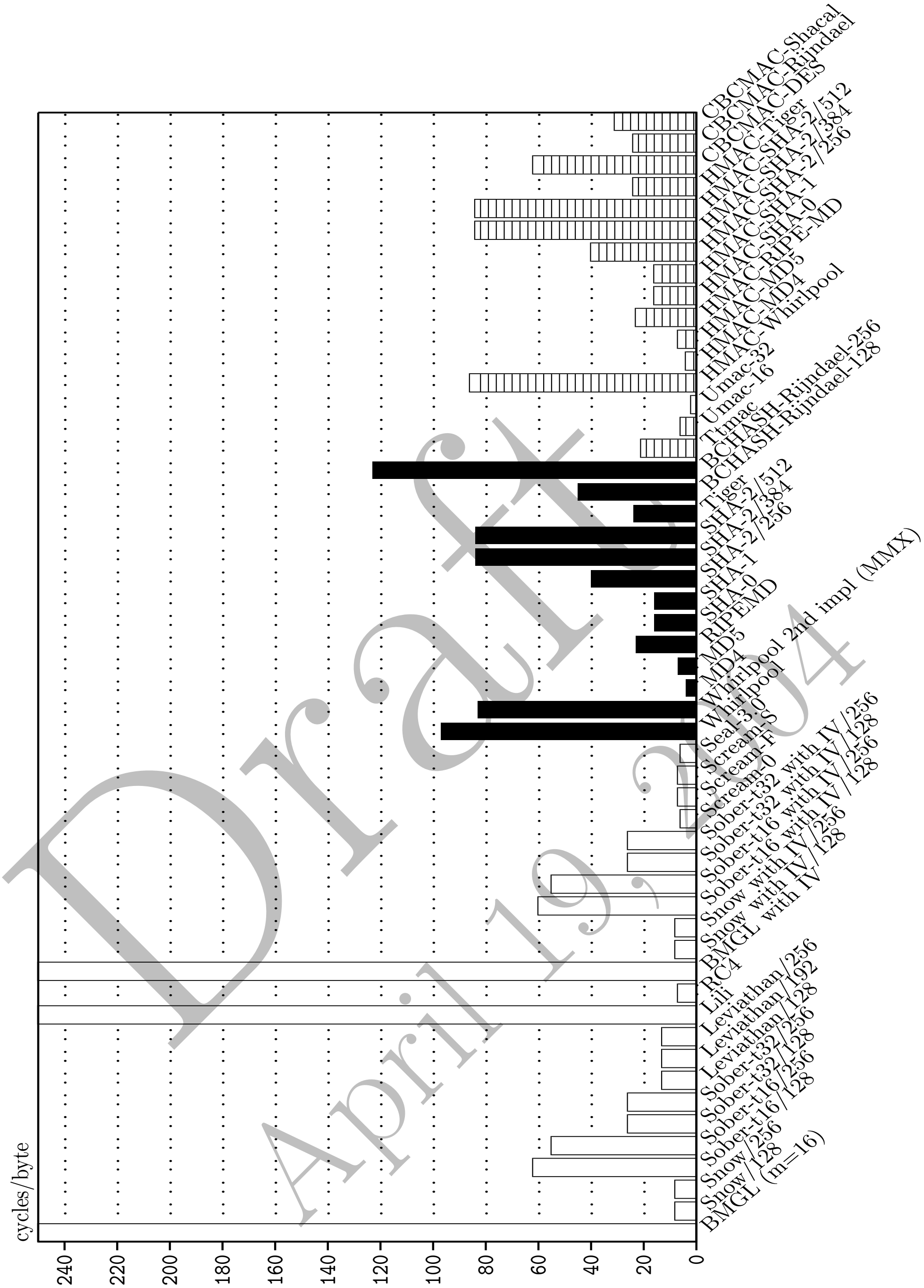


Fig. 17. Stream Ciphers, Hash Functions and MACs on Pentium2

□ stream ciphers ■ hash functions ▤ MACs

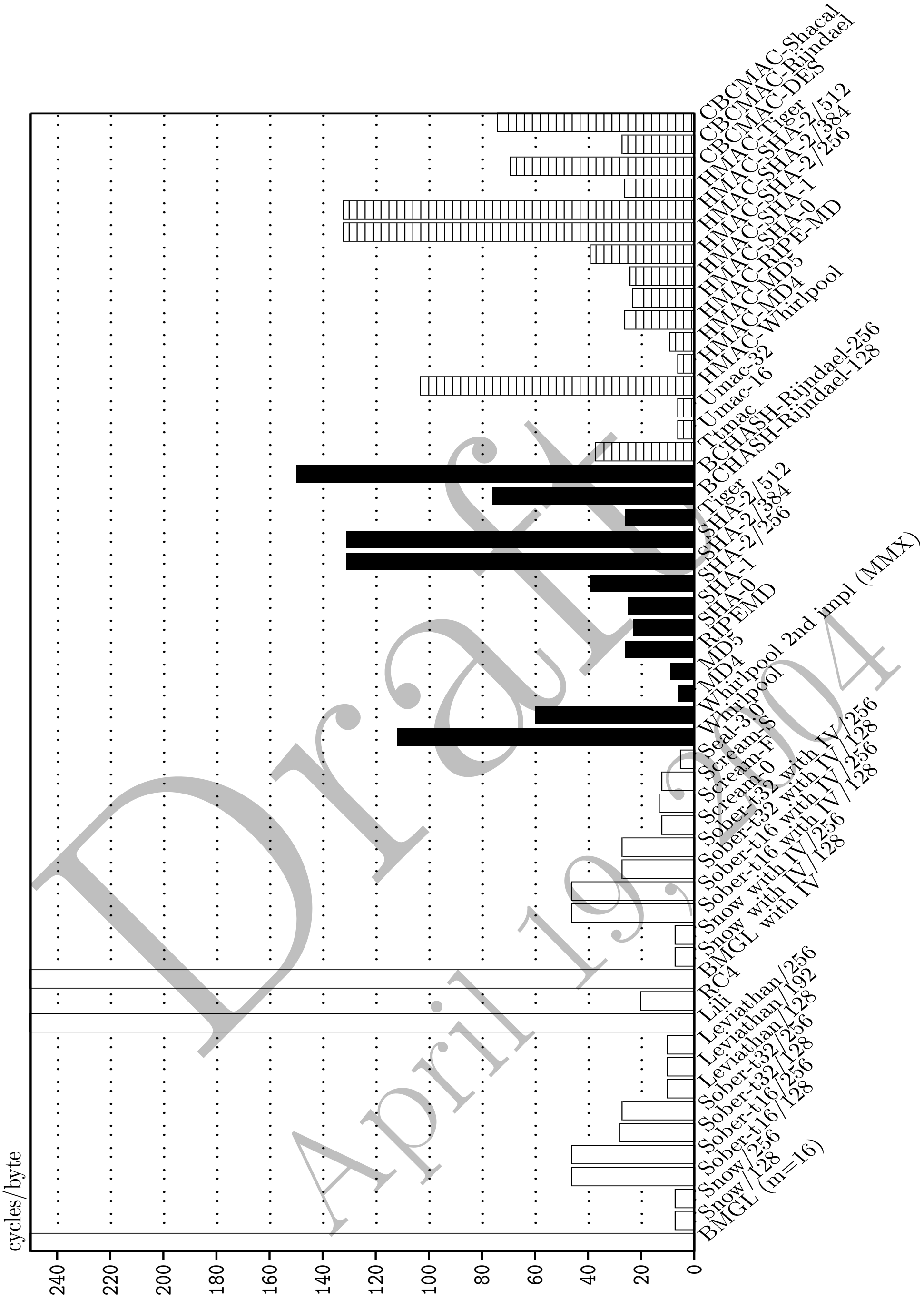


Fig. 18. Stream Ciphers, Hash Functions and MACs on XEON

□ stream ciphers ■ hash functions ▤ MACs

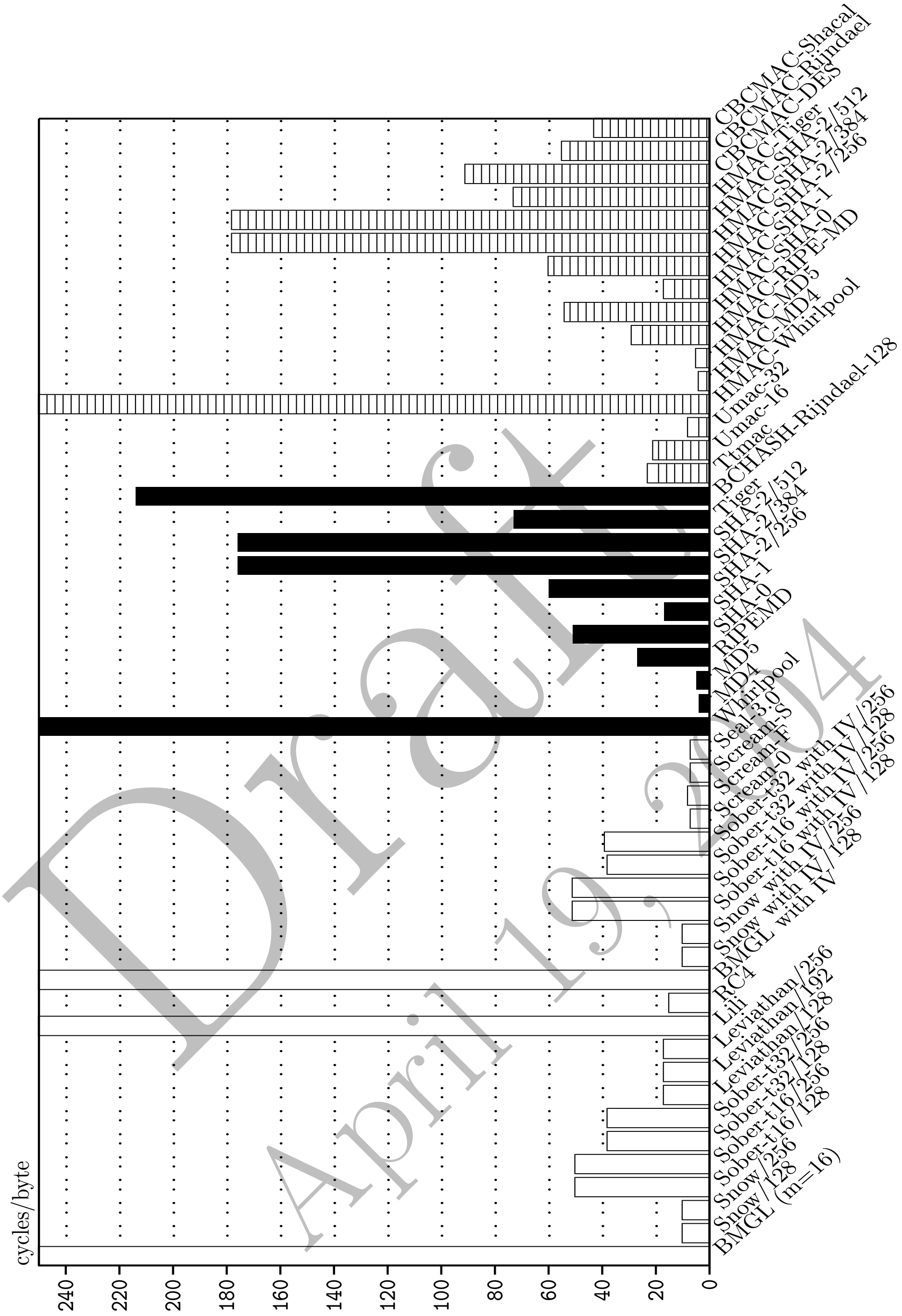


Fig. 19. Stream Ciphers, Hash Functions and MACs on 486

□ stream ciphers ■ hash functions ▤ MACs

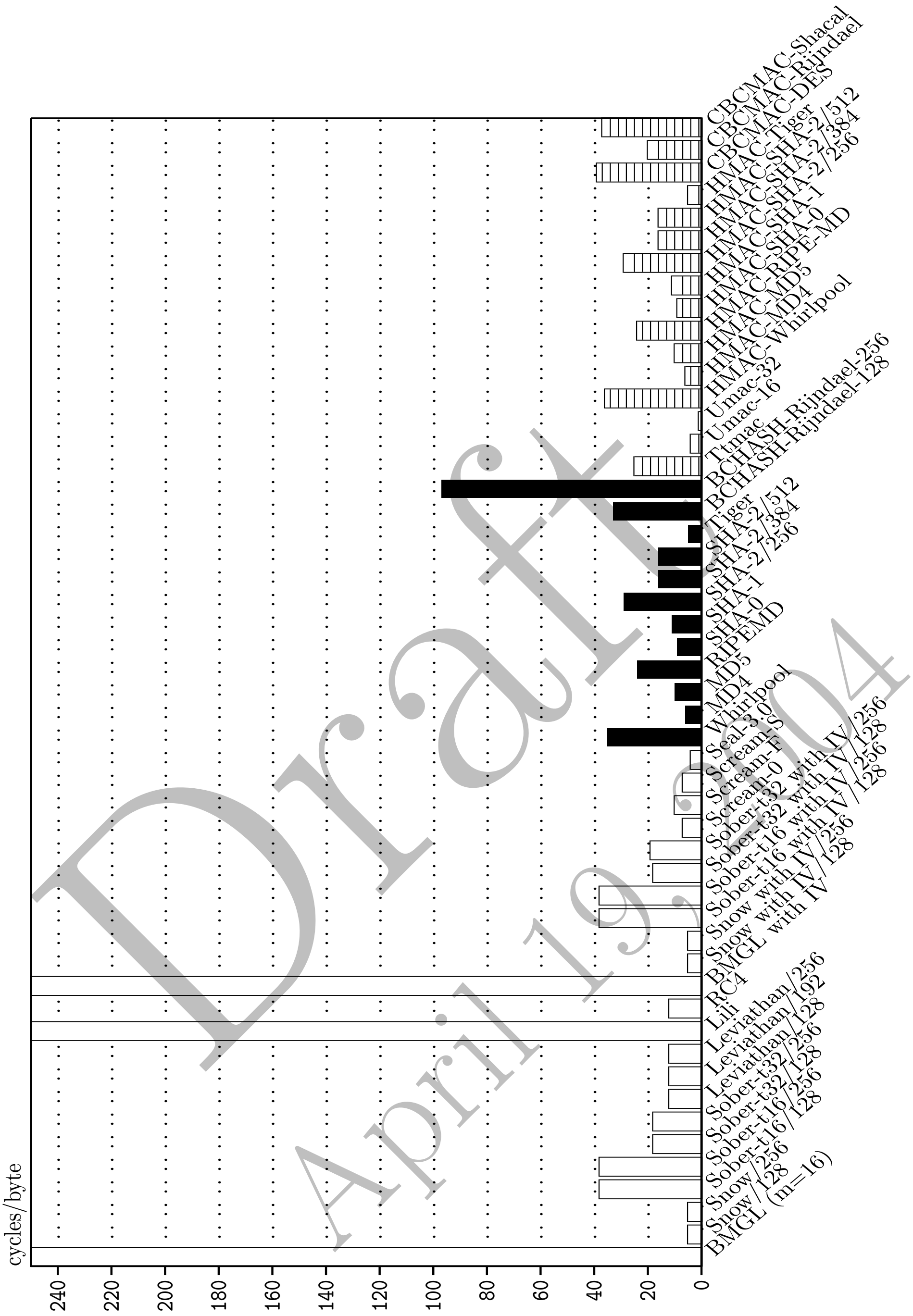


Fig. 20. Stream Ciphers, Hash Functions and MACs on Alpha

□ stream ciphers ■ hash functions ▤ MACs

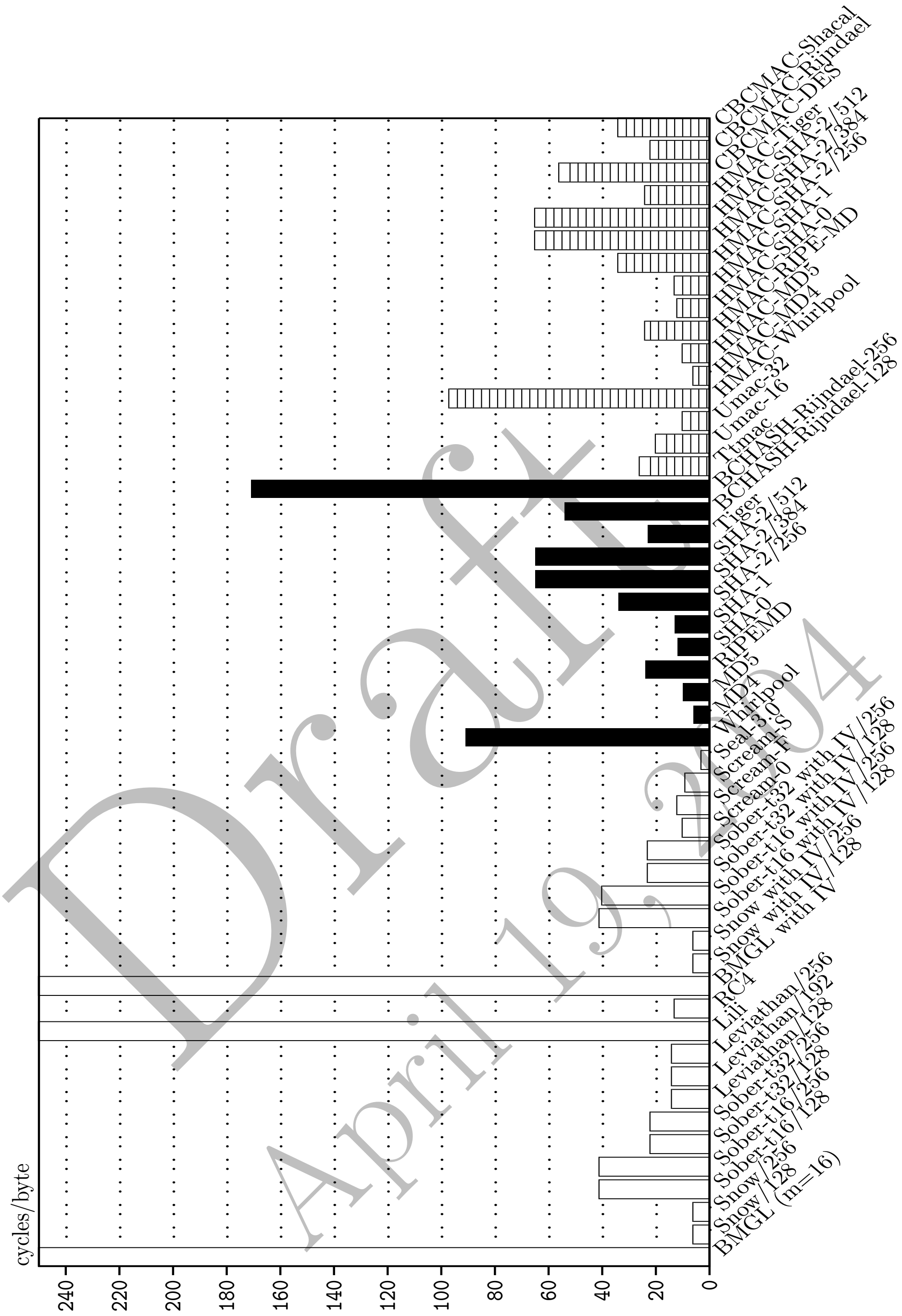


Fig. 21. Stream Ciphers, Hash Functions and MACs on Sparc V9

□ stream ciphers ■ hash functions ▤ MACs

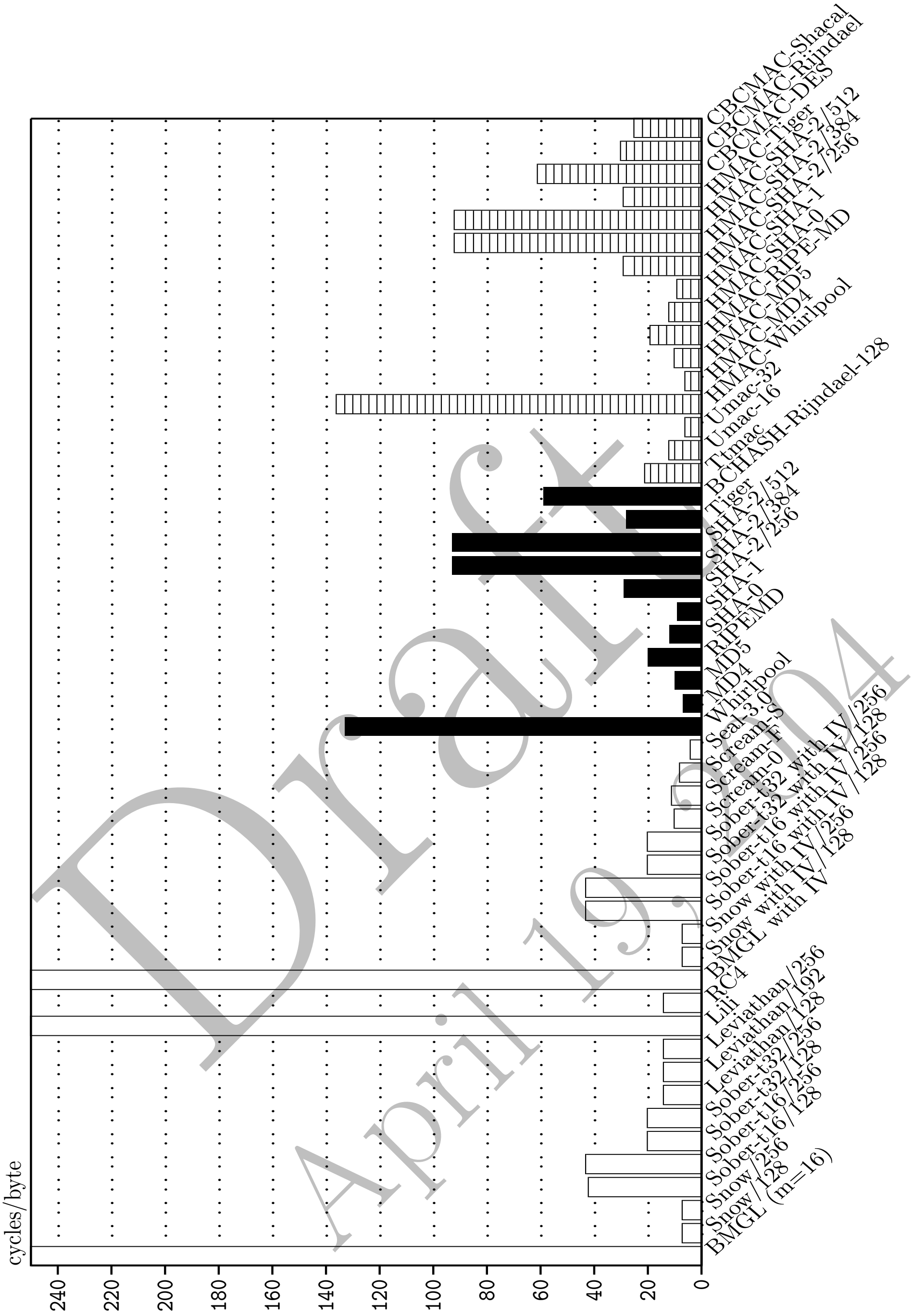


Fig. 22. Stream Ciphers, Hash Functions and MACs on Macintosh

□ stream ciphers ■ hash functions ▤ MACs

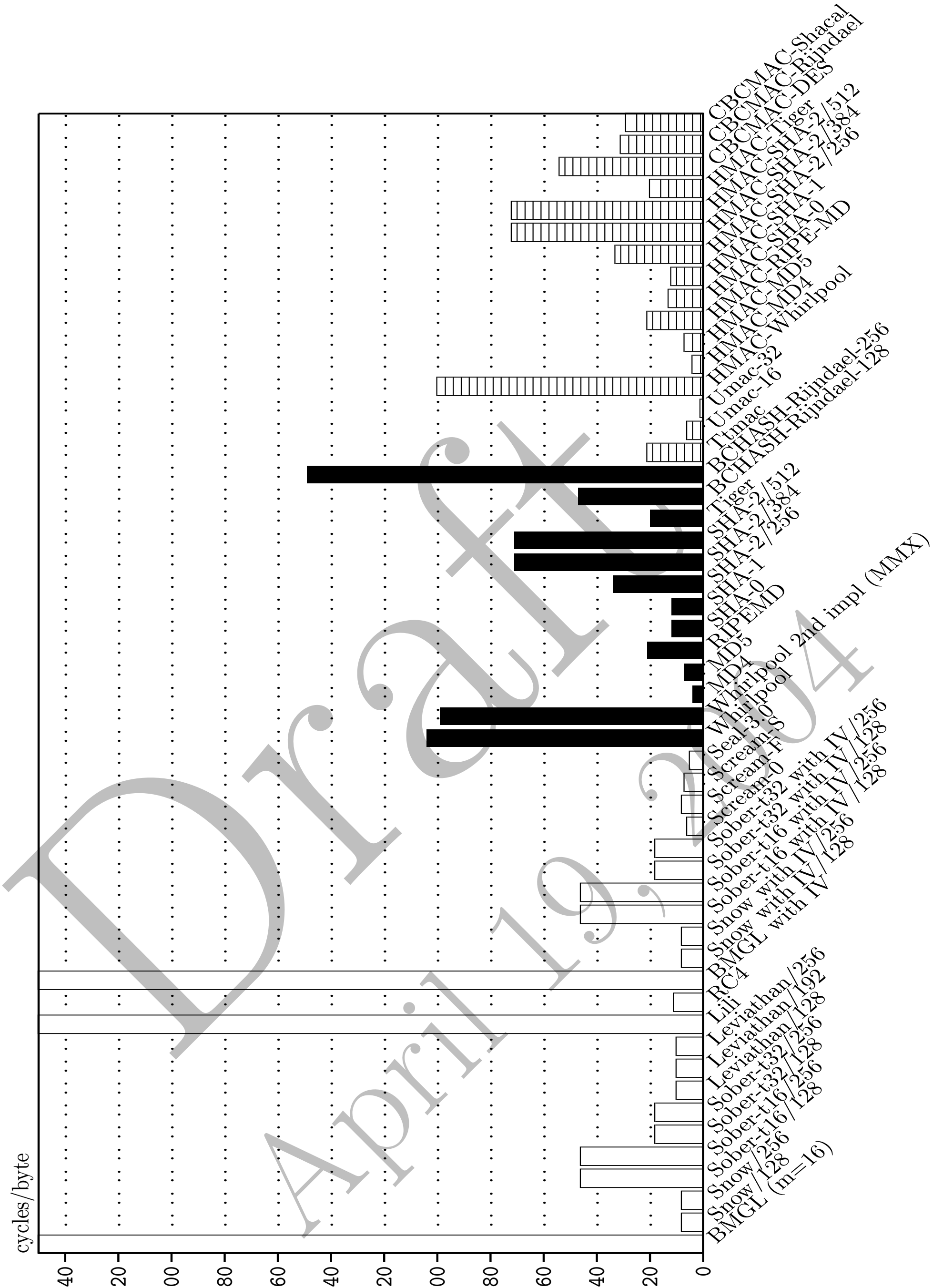


Fig. 23. Stream Ciphers, Hash Functions and MACs on AMD

□ stream ciphers ■ hash functions ▤ MACs

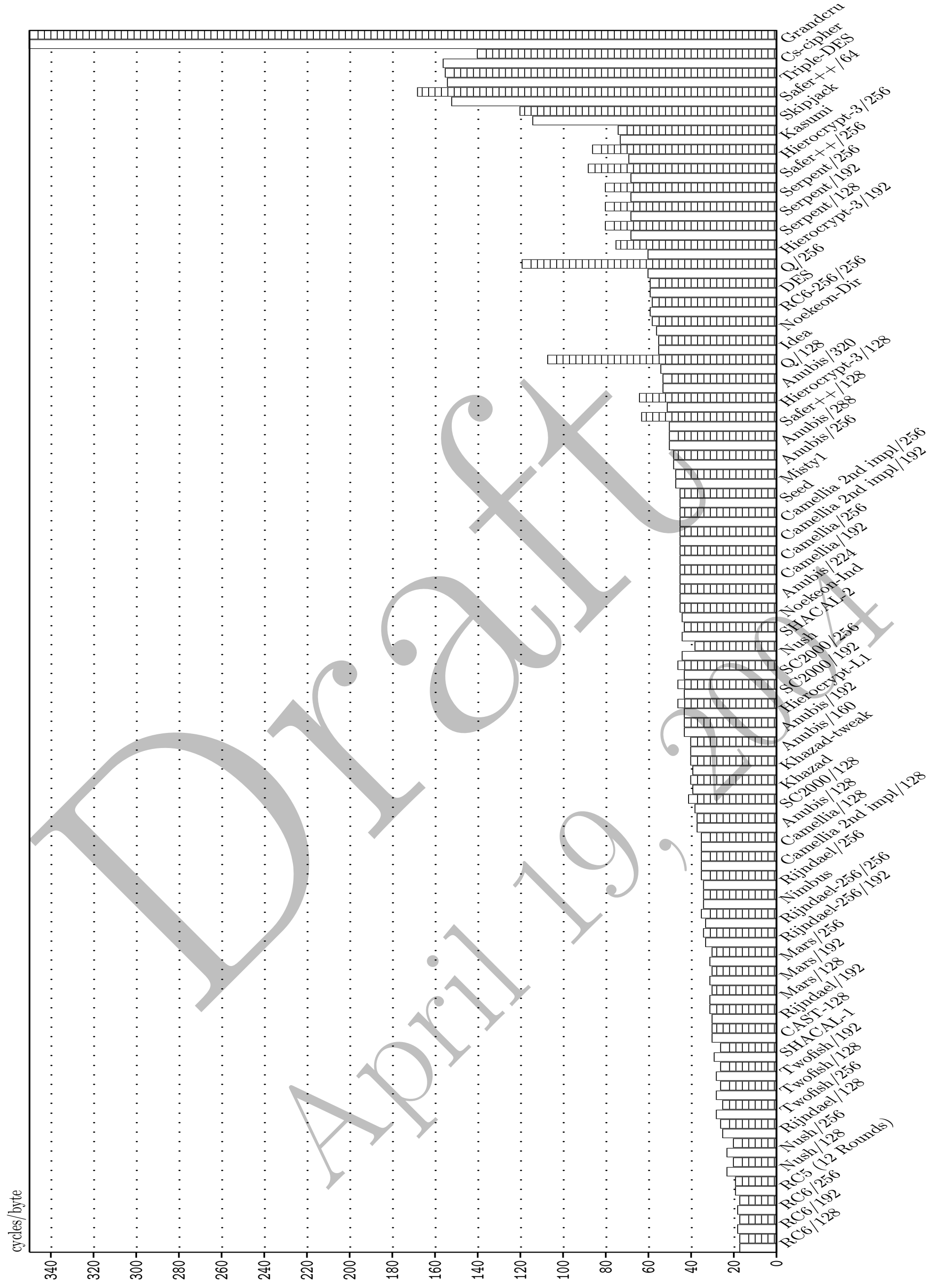


Fig. 24. Block Ciphers on PIII/Linux (Sorted)

□ encryption ▤ decryption

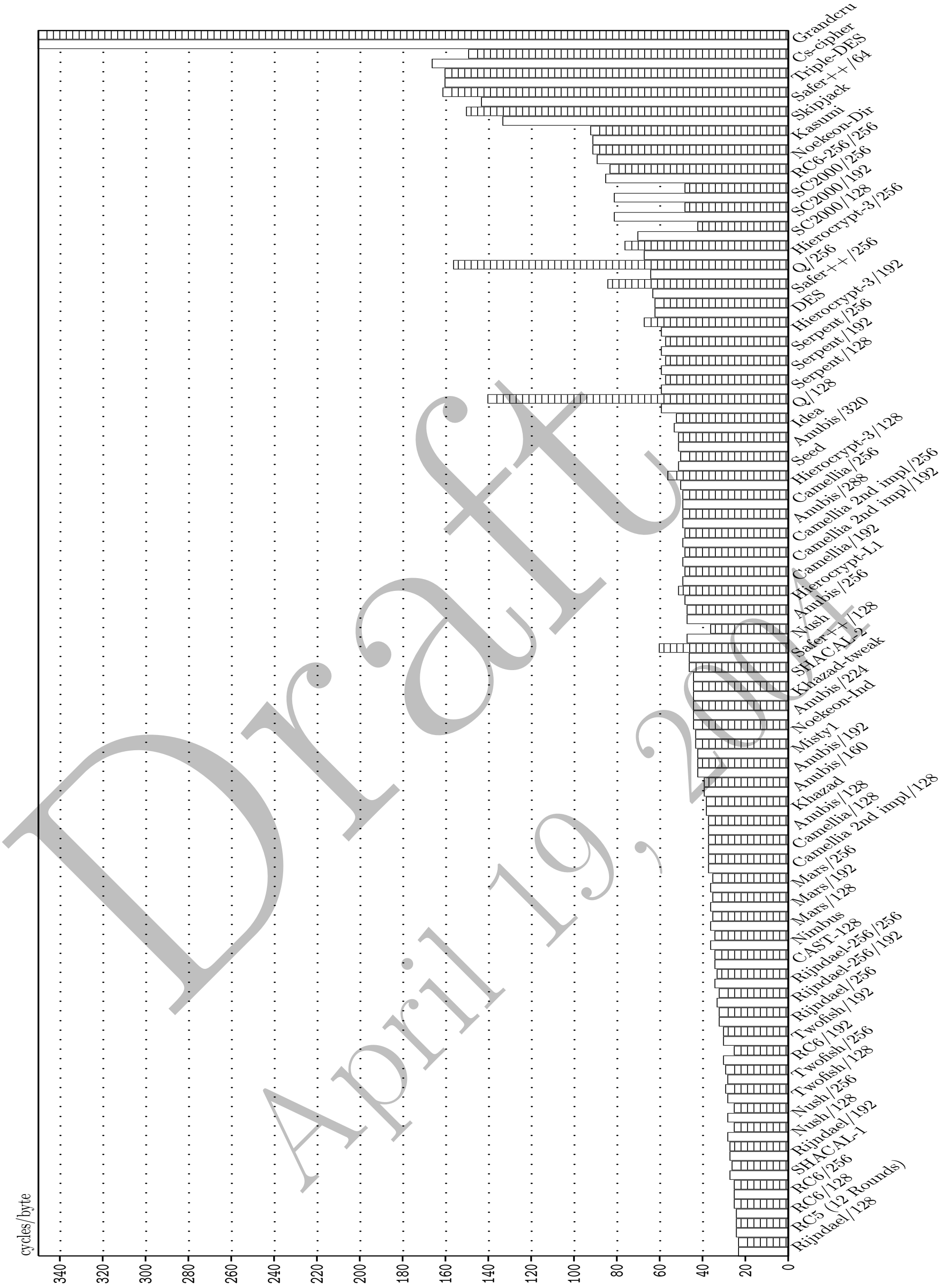


Fig. 25. Block Ciphers on PIII/MS (Sorted)

□ encryption ▤ decryption

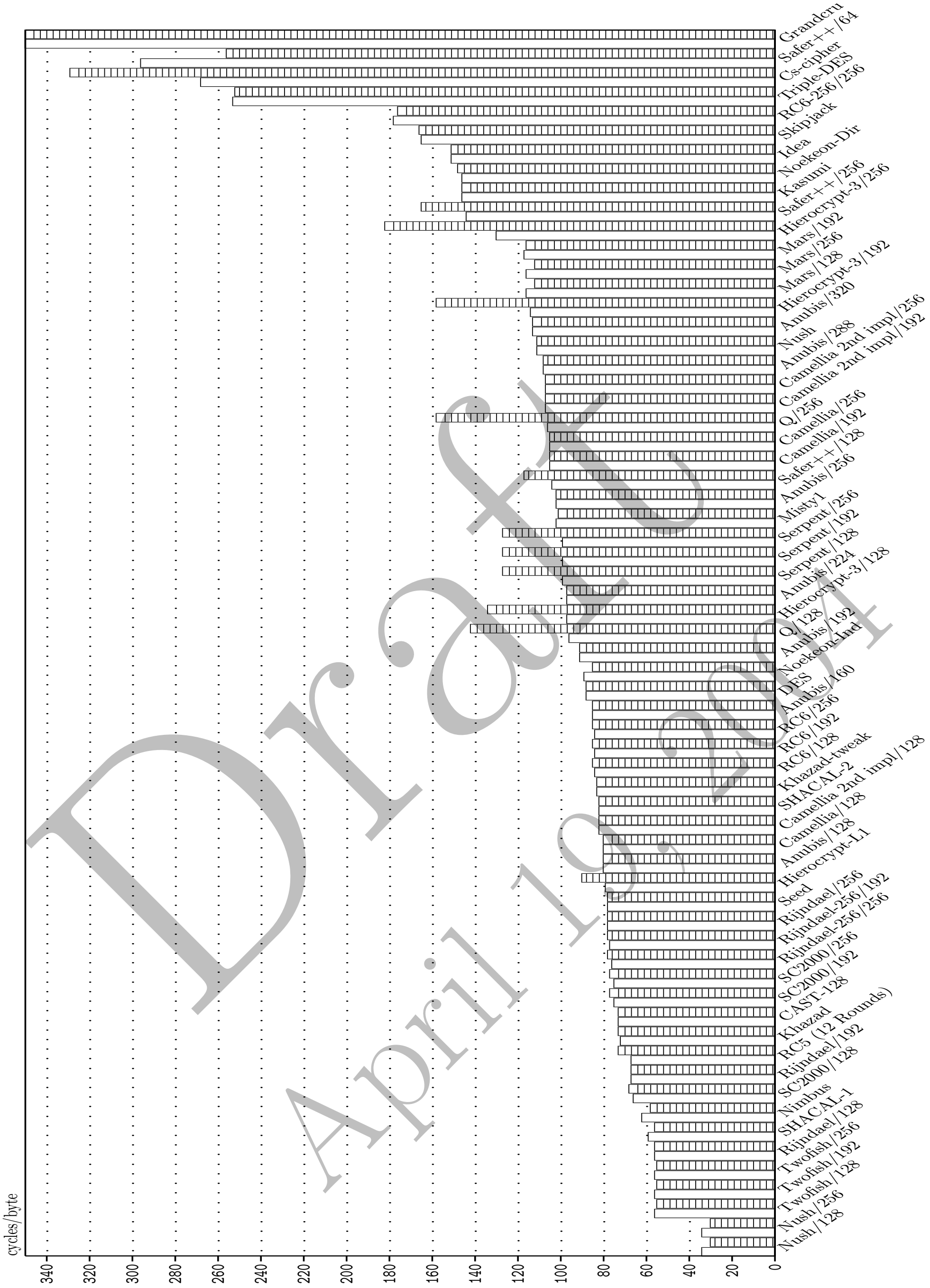


Fig. 26. Block Ciphers on PI/MMX (Sorted)

□ encryption ▤ decryption

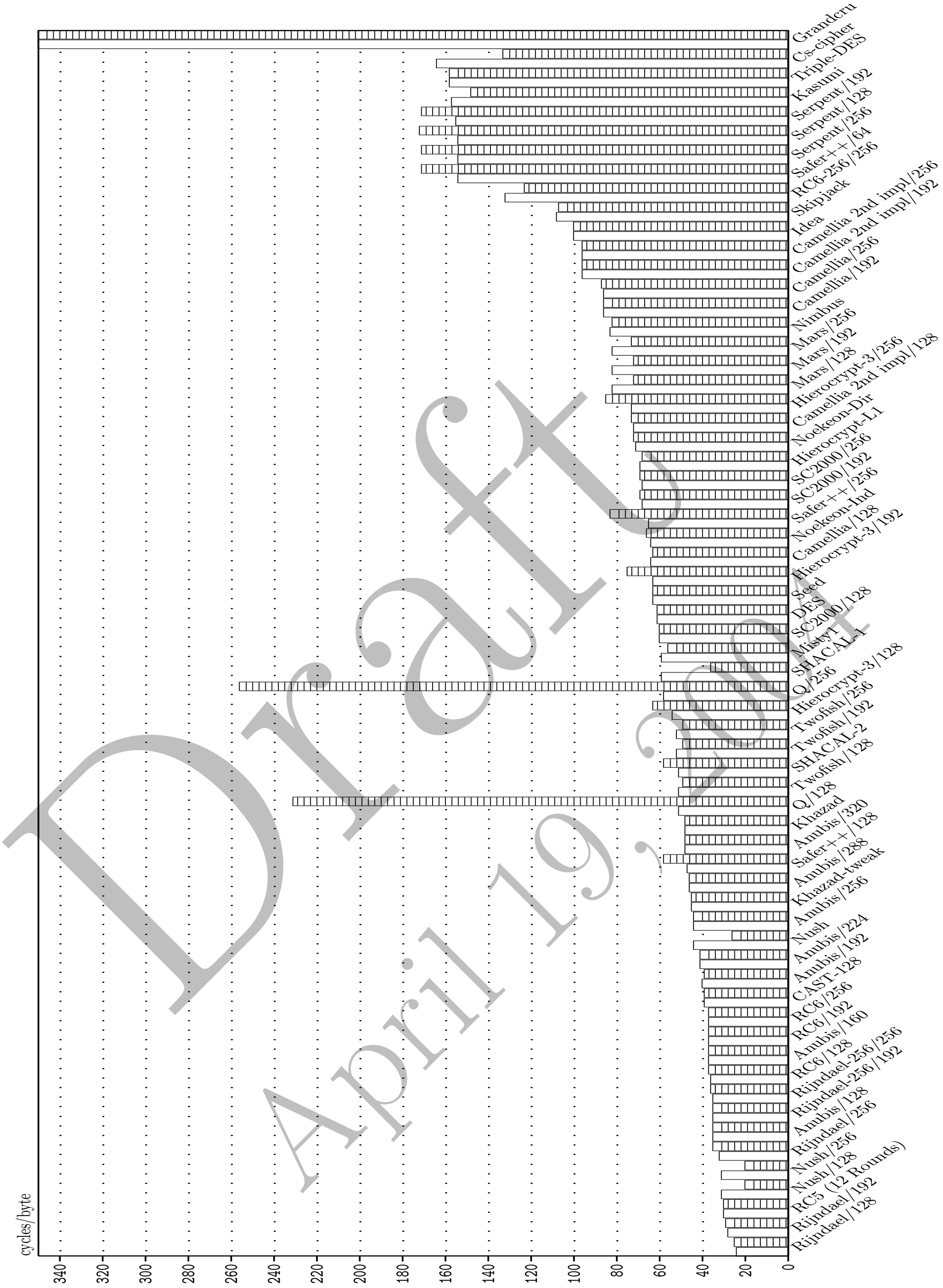


Fig. 27. Block Ciphers on Pentium4 (Sorted)

□ encryption ▤ decryption

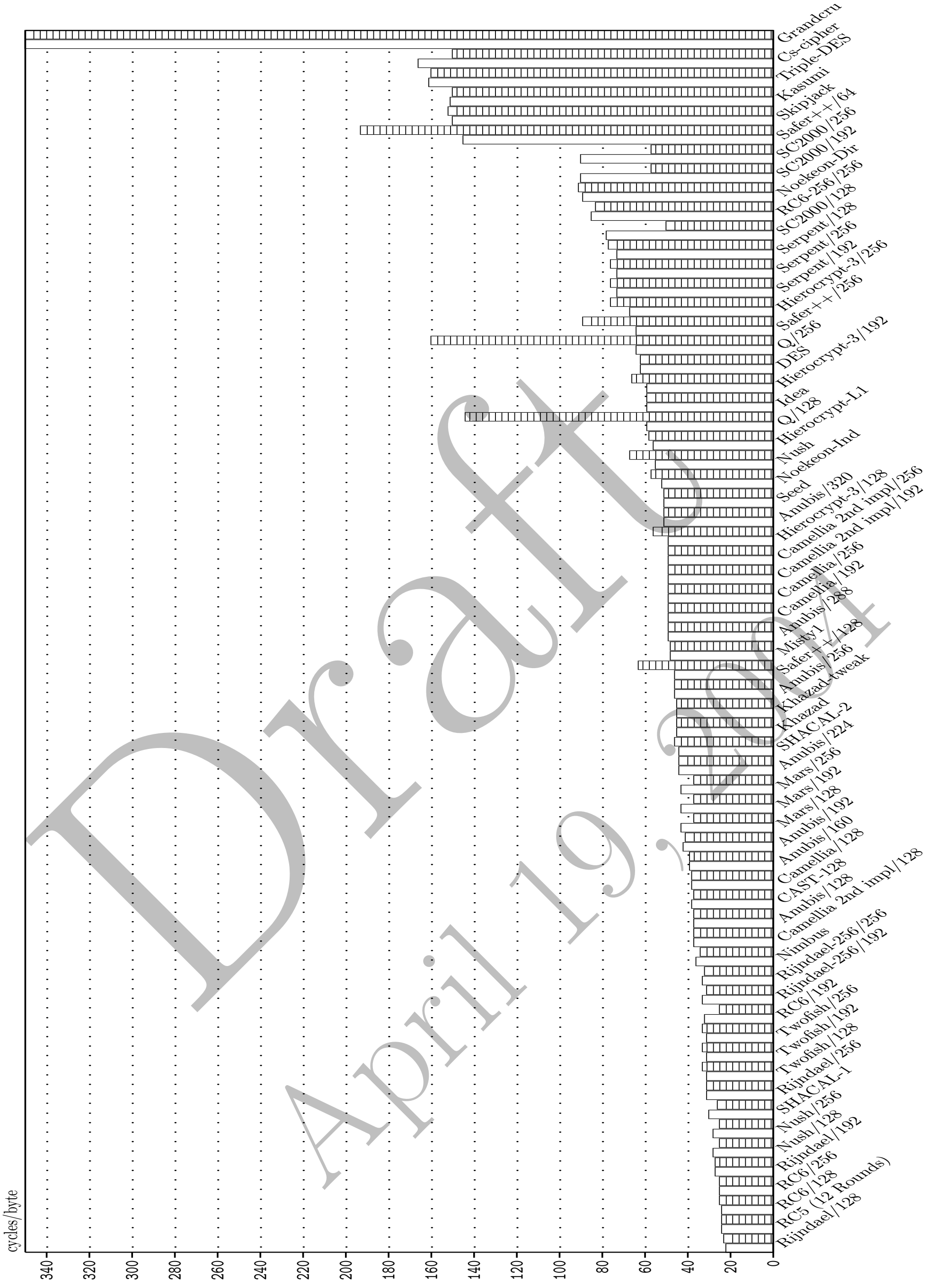


Fig. 28. Block Ciphers on Pentium2 (Sorted)

□ encryption ▤ decryption

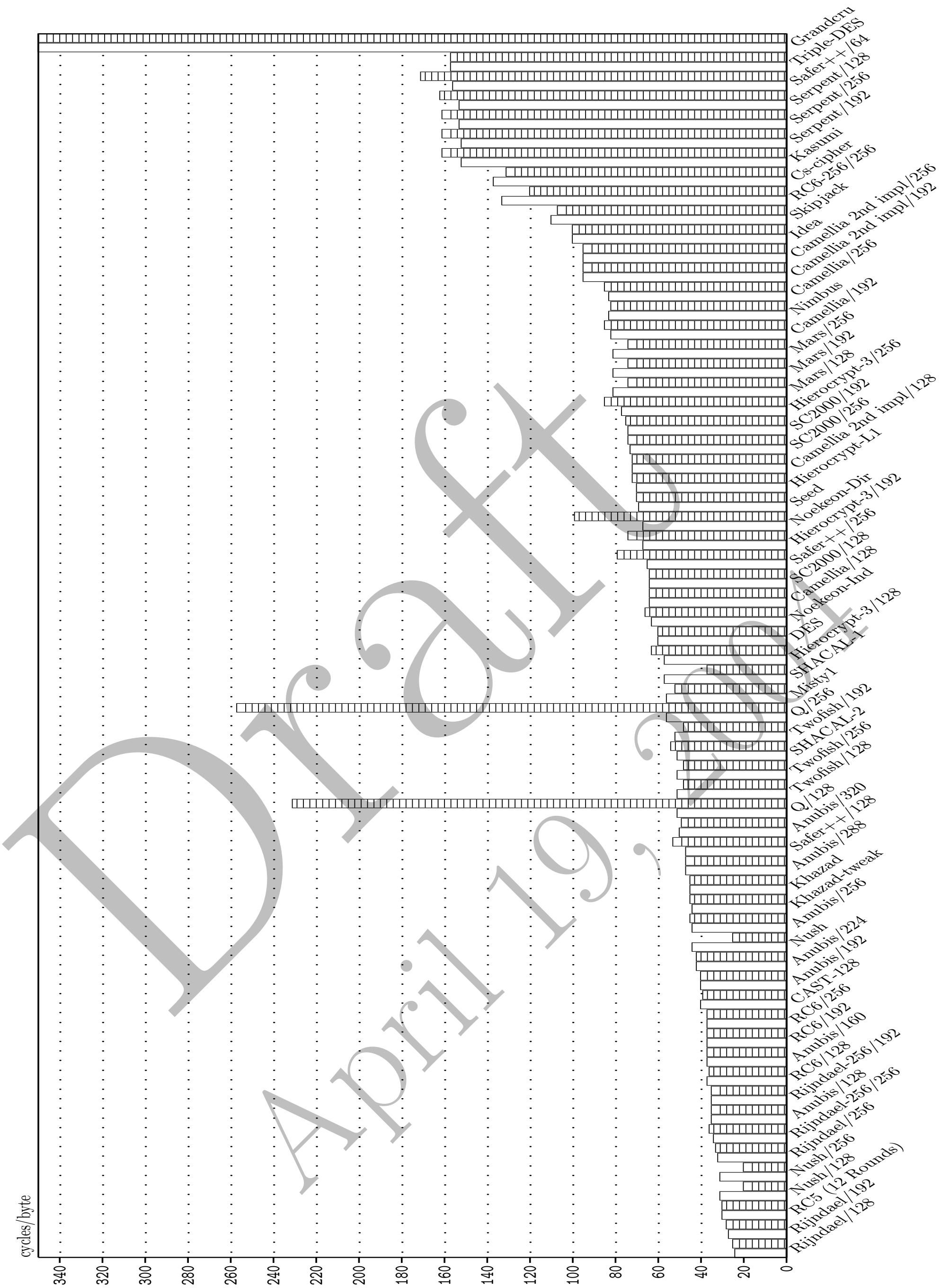


Fig. 29. Block Ciphers on Xeon (Sorted)

□ encryption ▤ decryption

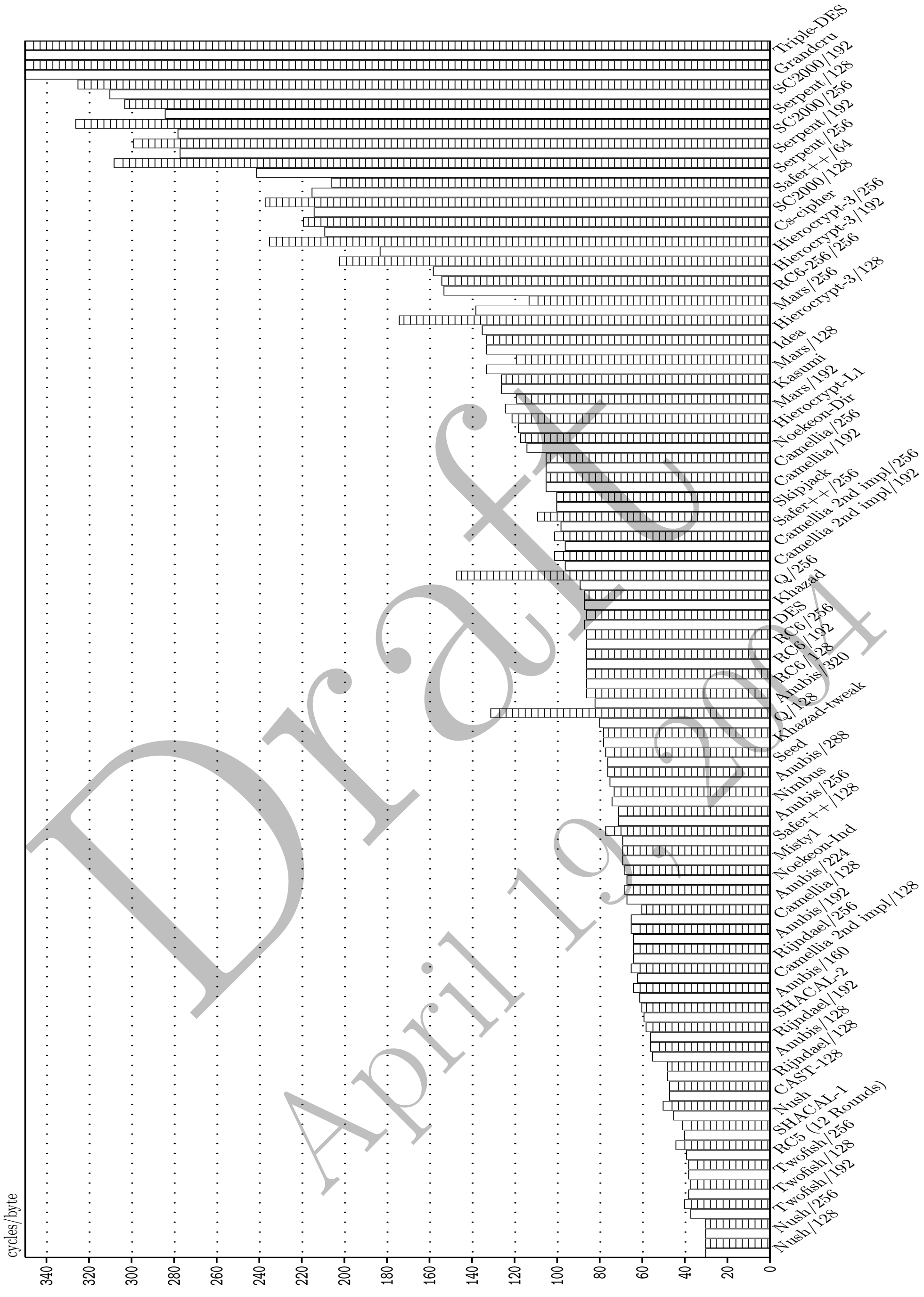


Fig. 30. Block Ciphers on 486 (Sorted)

□ encryption ▤ decryption

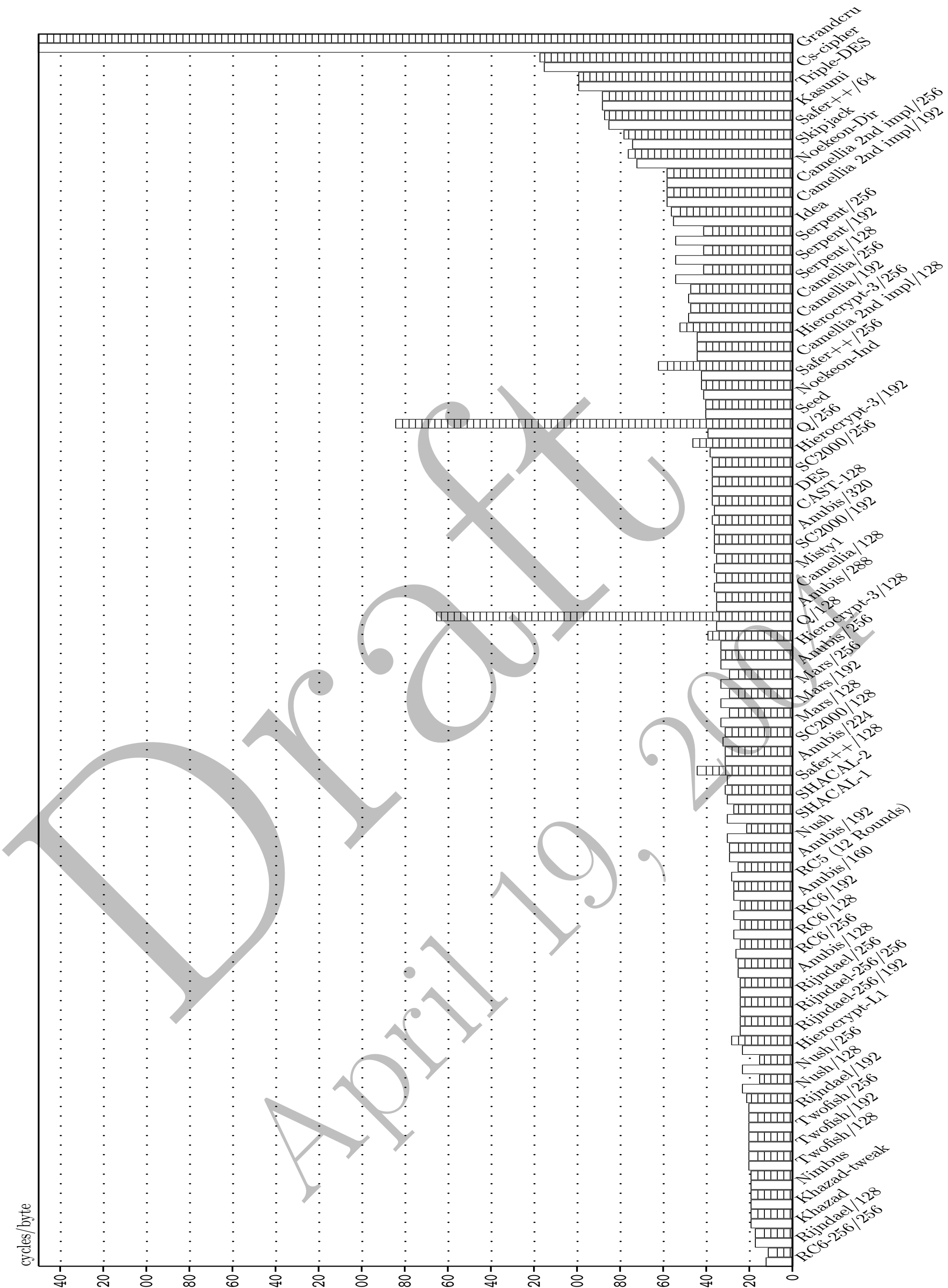


Fig. 31. Block Ciphers on Alpha (Sorted)

□ encryption ▤ decryption

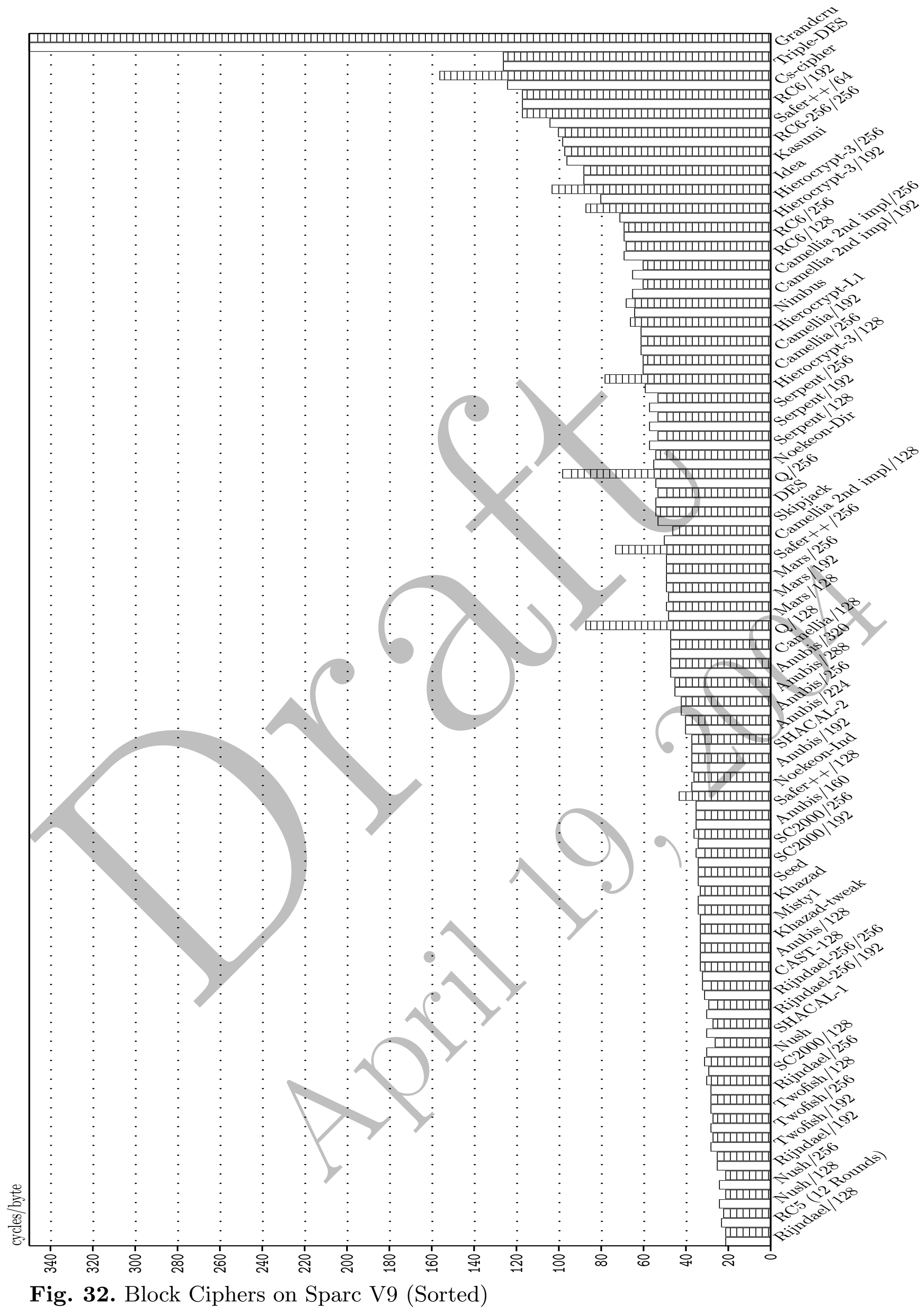


Fig. 32. Block Ciphers on Sparc V9 (Sorted)

□ encryption ▤ decryption

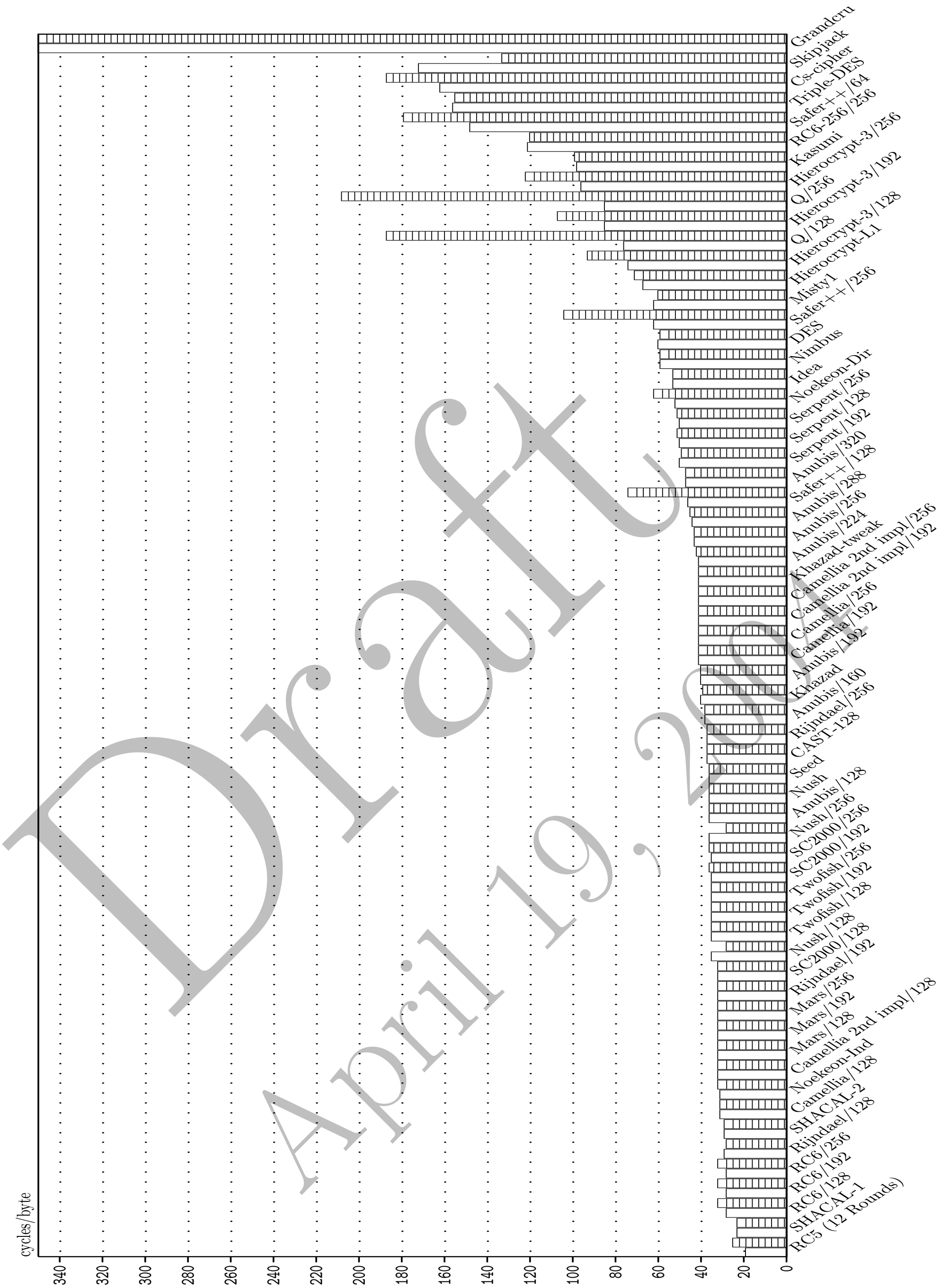


Fig. 33. Block Ciphers on Macintosh (Sorted)

□ encryption ▤ decryption

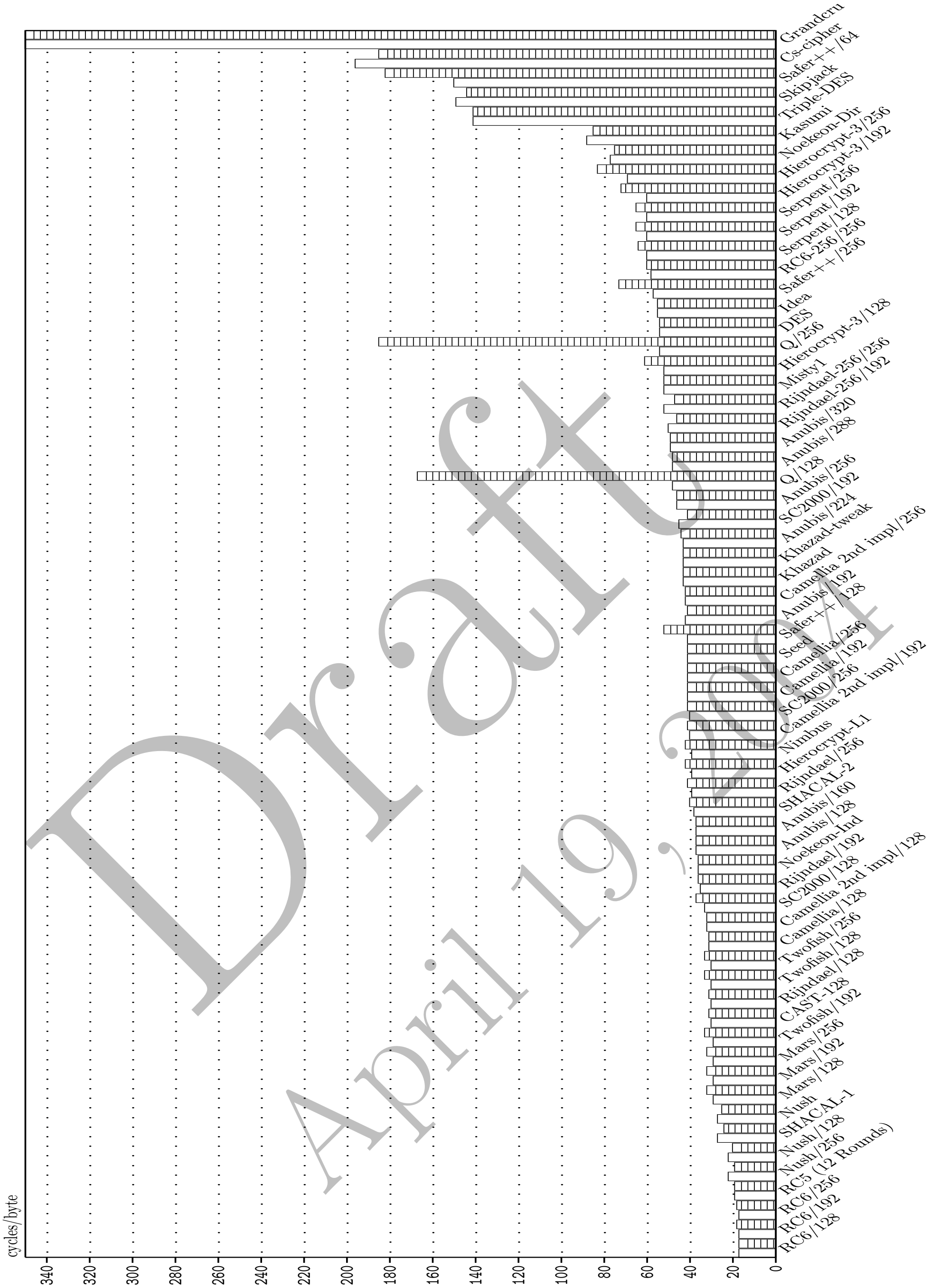


Fig. 34. Block Ciphers on AMD (Sorted)

□ encryption ▤ decryption

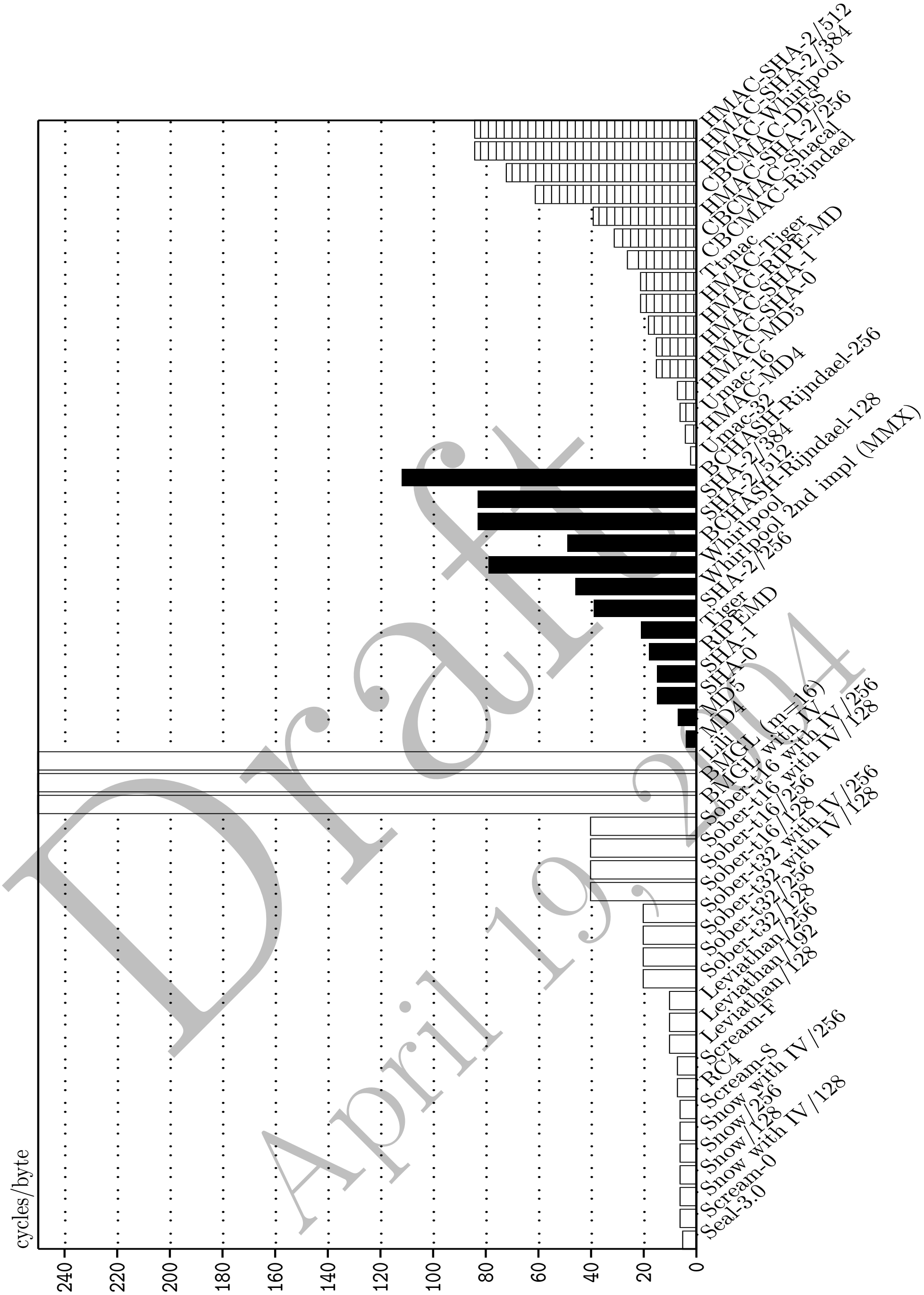


Fig. 35. Stream Ciphers, Hash Functions and MACs on PIII/Linux (Sorted)

□ stream ciphers ■ hash functions ▤ MACs

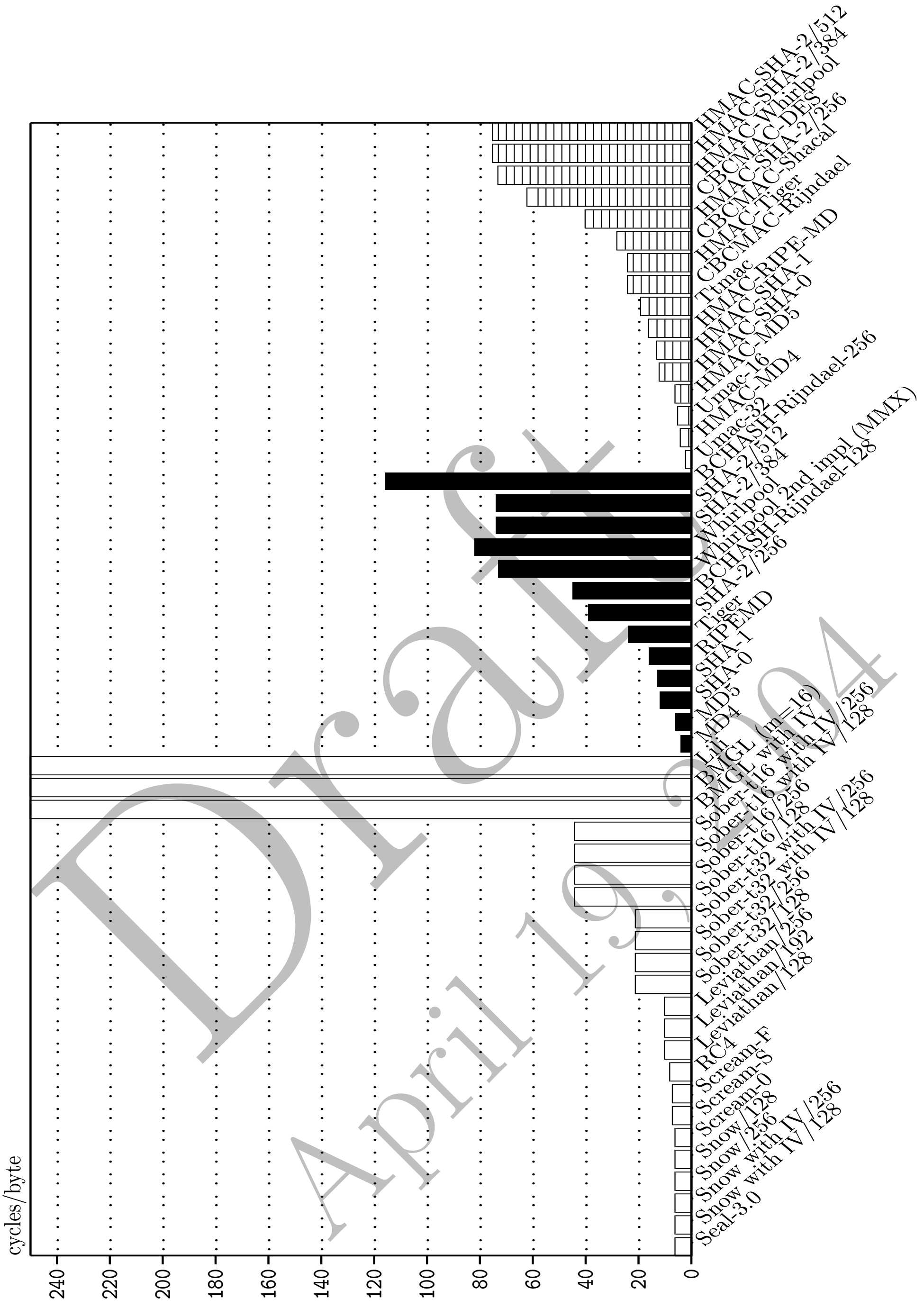


Fig. 36. Stream Ciphers, Hash Functions and MACs on PIII/MS (Sorted)

□ stream ciphers ■ hash functions ▤ MACs

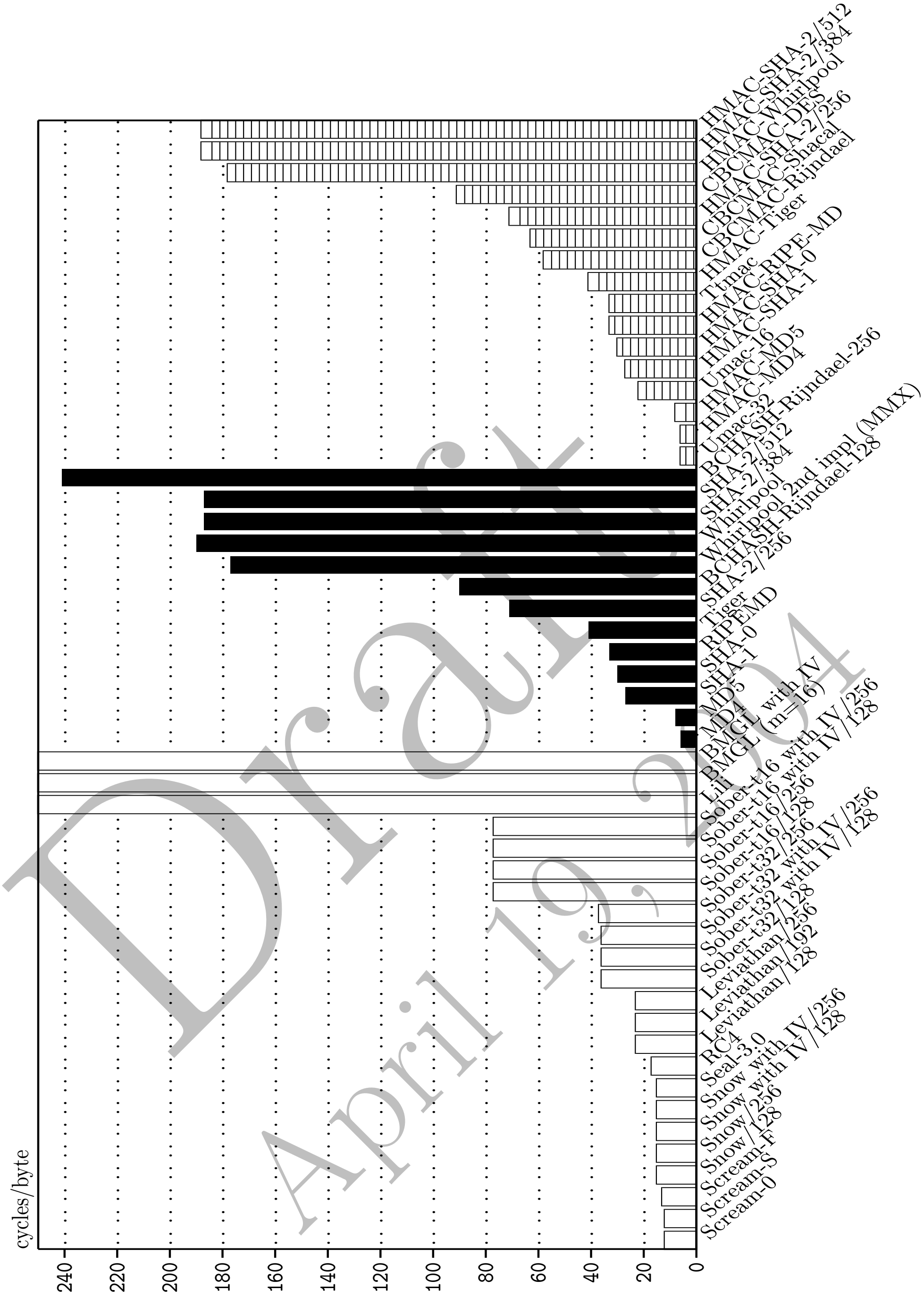


Fig. 37. Stream Ciphers, Hash Functions and MACs on PIII/MMX (Sorted)

□ stream ciphers ■ hash functions ▤ MACs

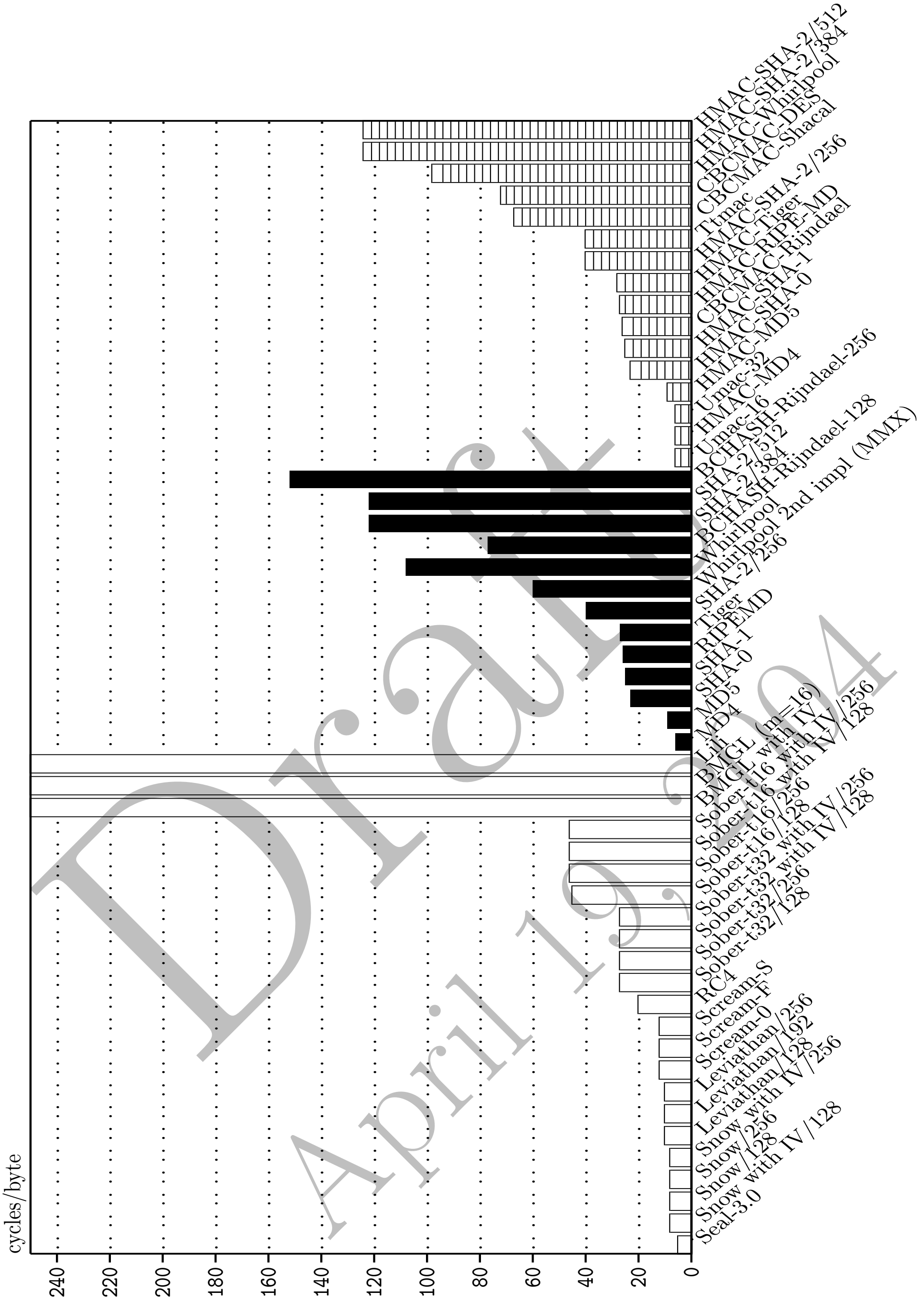


Fig. 38. Stream Ciphers, Hash Functions and MACs on Pentium4 (Sorted)

□ stream ciphers ■ hash functions ▤ MACs

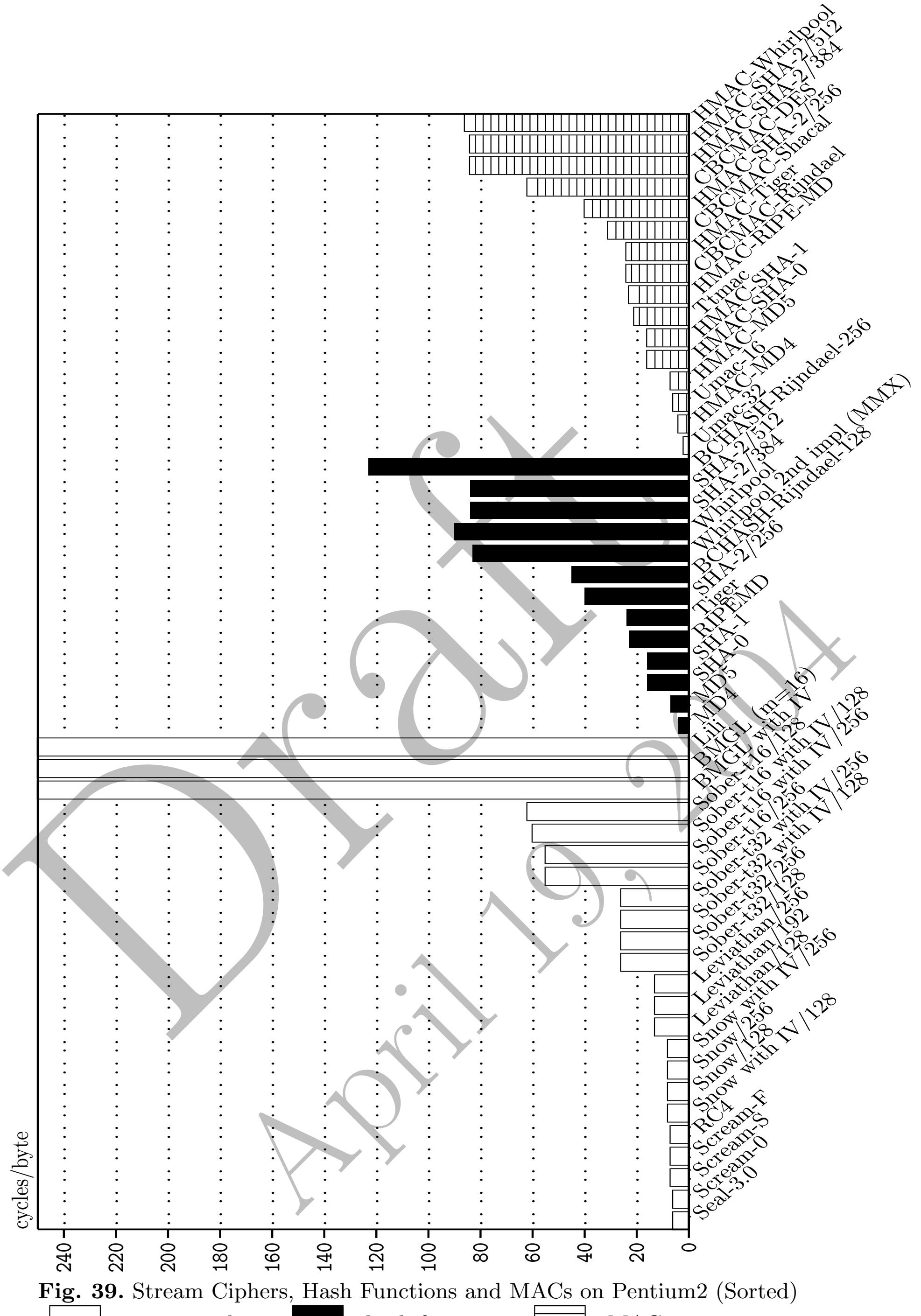


Fig. 39. Stream Ciphers, Hash Functions and MACs on Pentium2 (Sorted)

□ stream ciphers ■ hash functions ▤ MACs

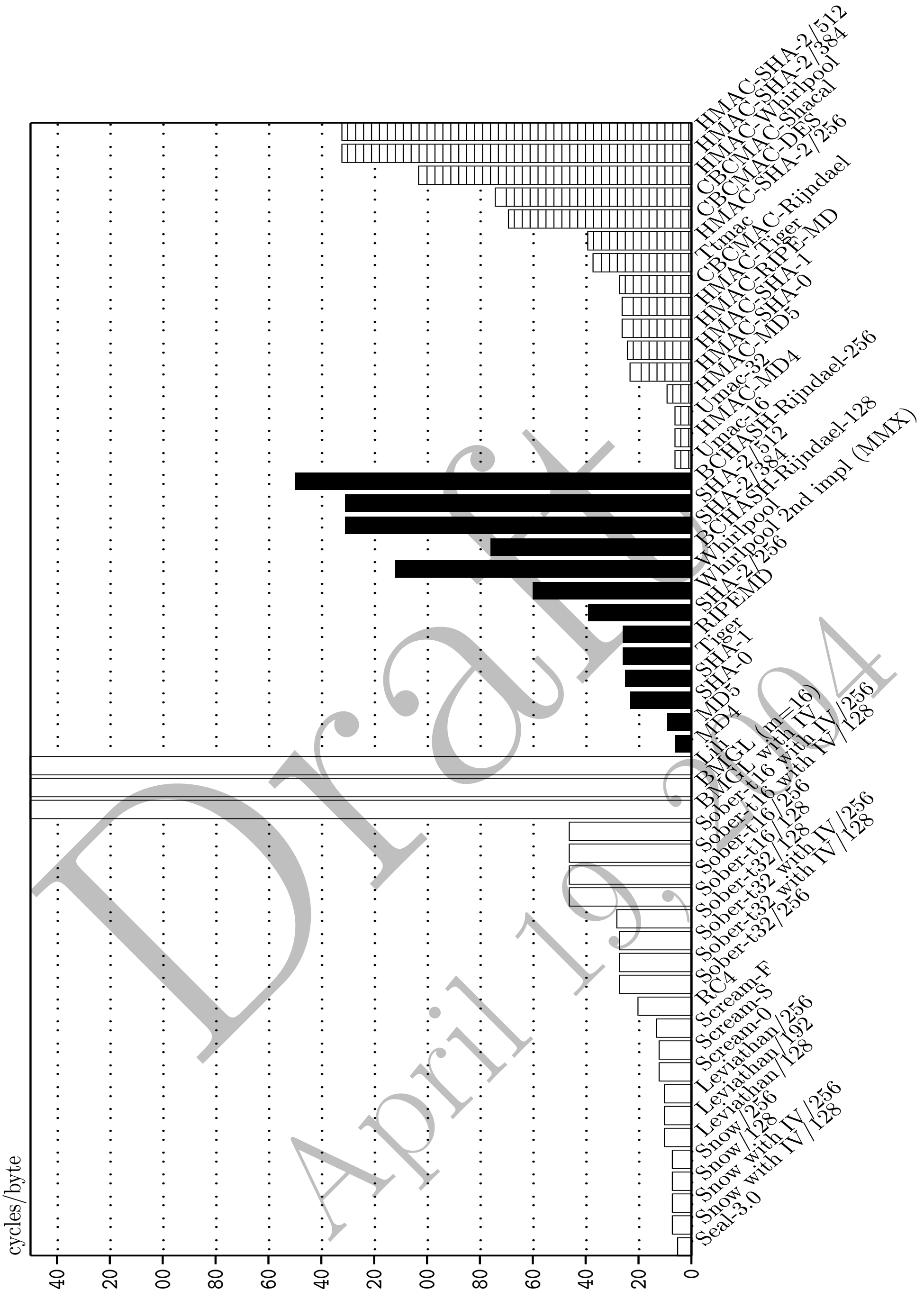


Fig. 40. Stream Ciphers, Hash Functions and MACs on XEON (Sorted)

□ stream ciphers ■ hash functions ▤ MACs

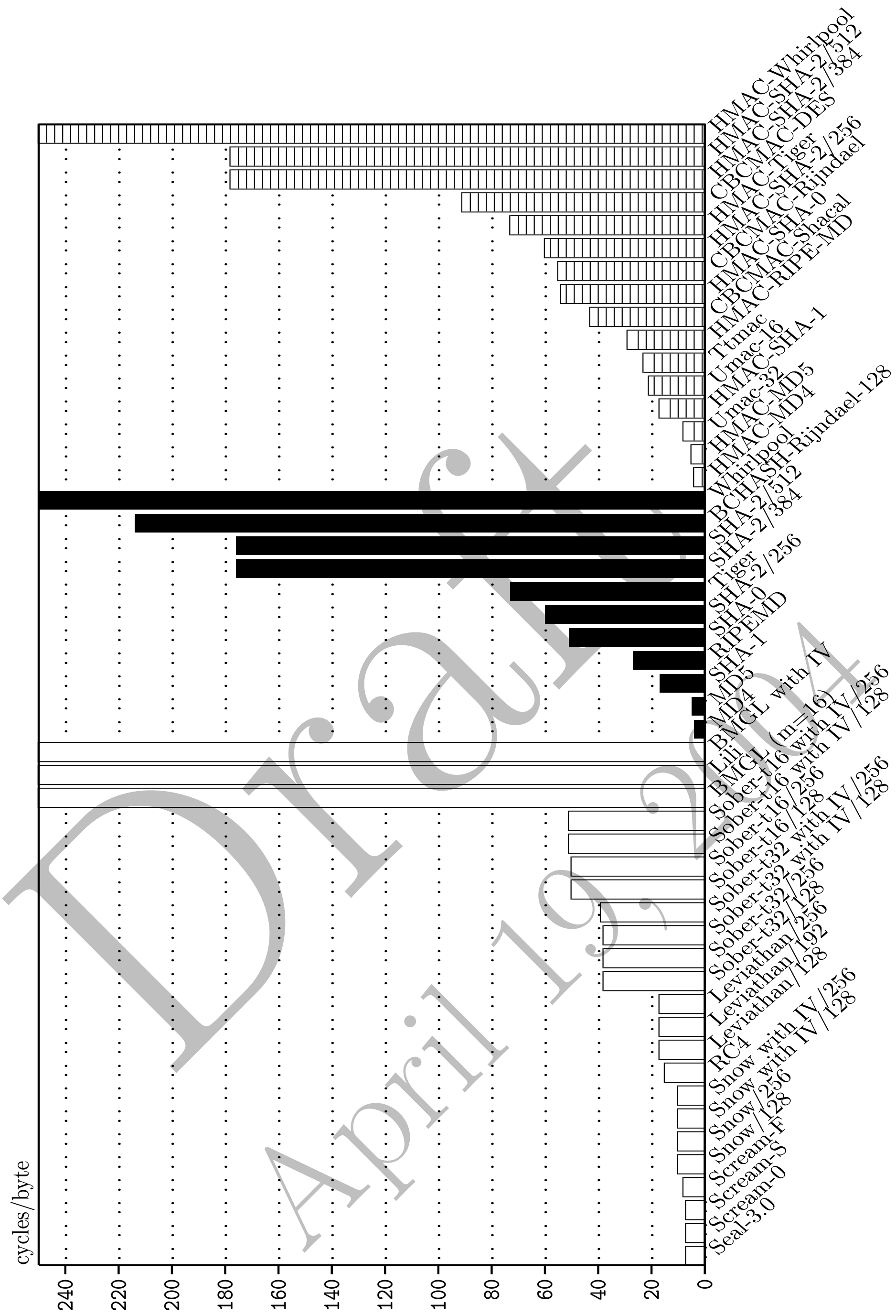


Fig. 41. Stream Ciphers, Hash Functions and MACs on 486 (Sorted)

□ stream ciphers ■ hash functions ▤ MACs

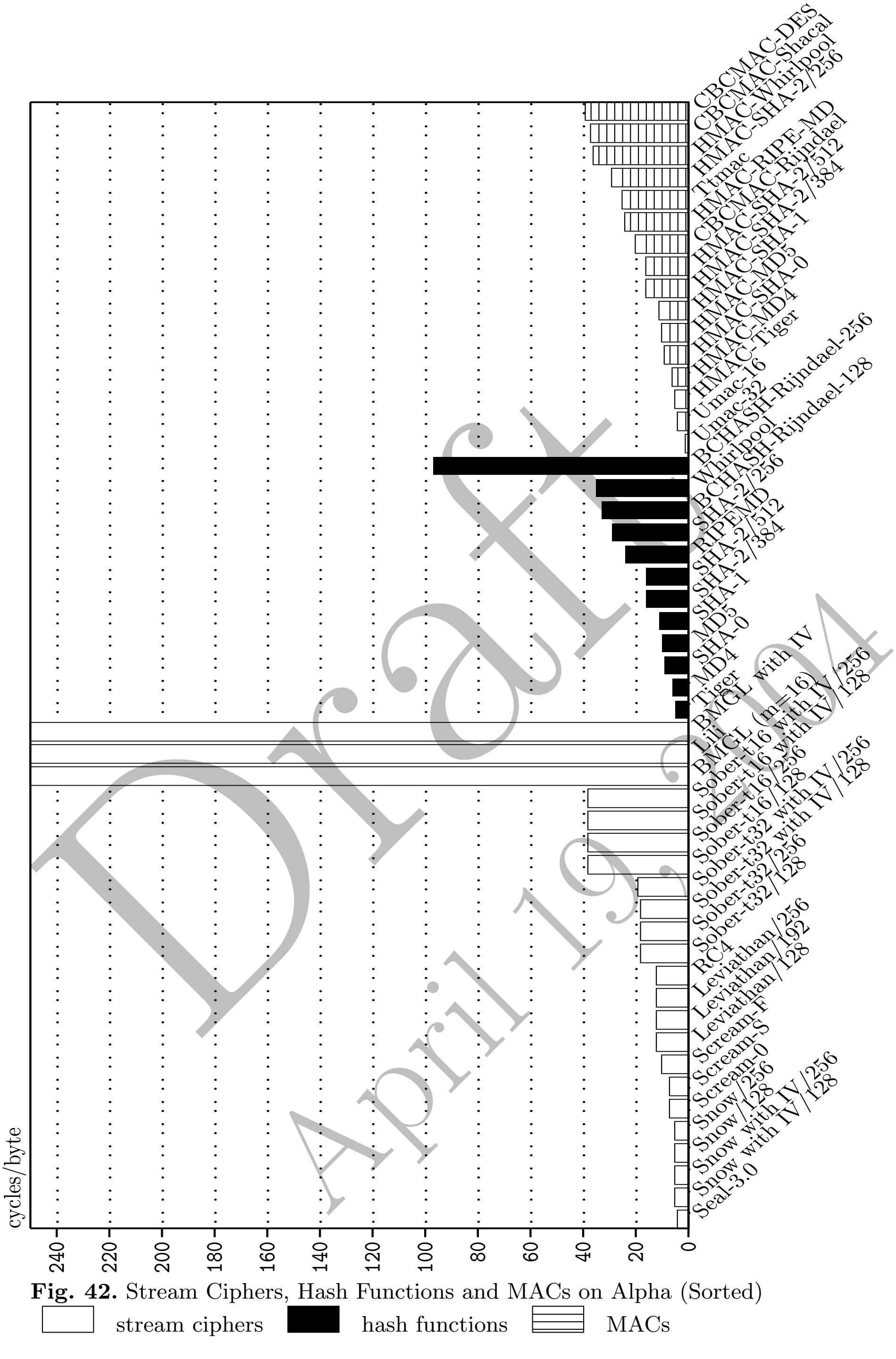


Fig. 42. Stream Ciphers, Hash Functions and MACs on Alpha (Sorted)

□ stream ciphers ■ hash functions ▤ MACs

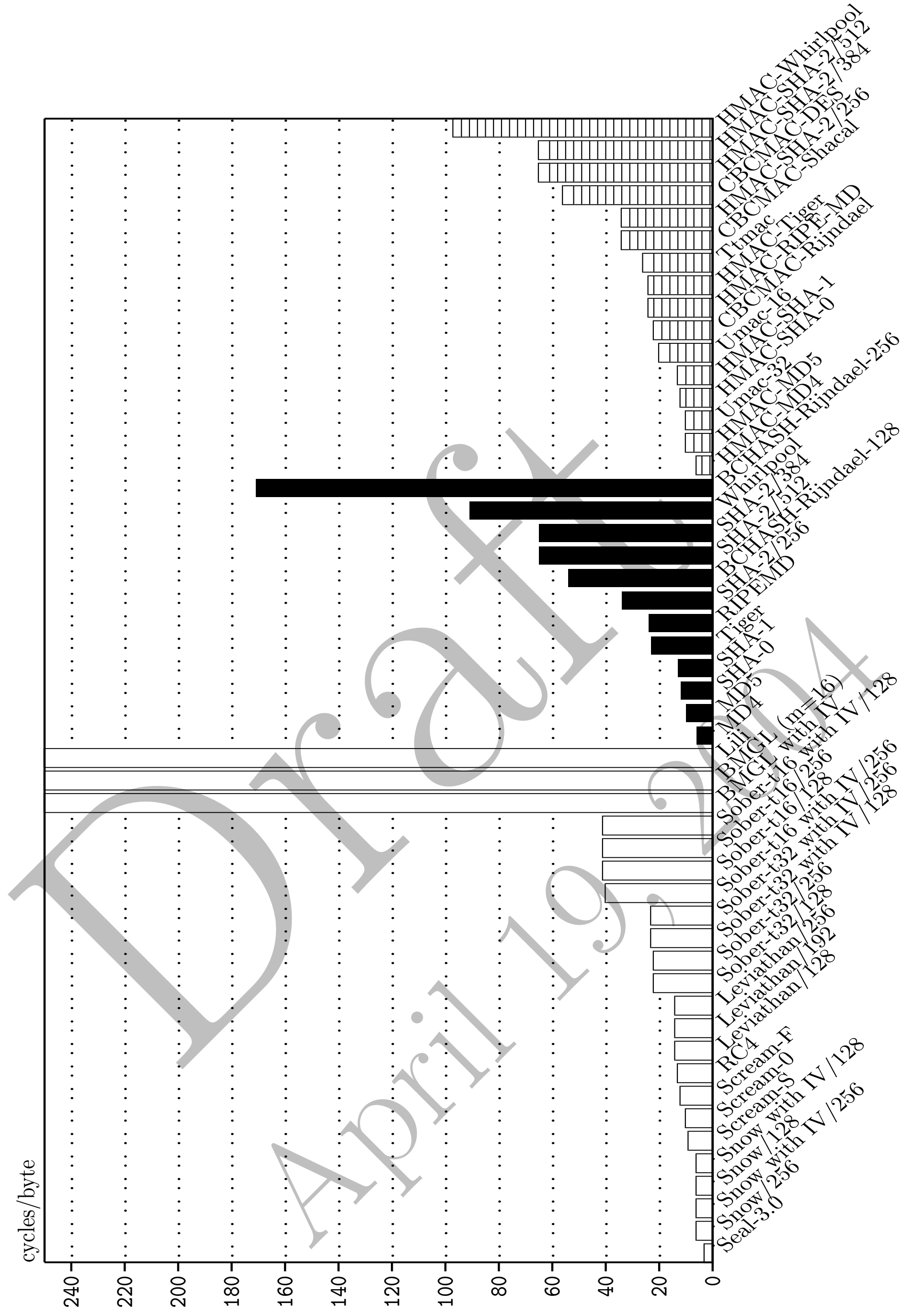


Fig. 43. Stream Ciphers, Hash Functions and MACs on Sparc V9 (Sorted)

□ stream ciphers ■ hash functions ▤ MACs

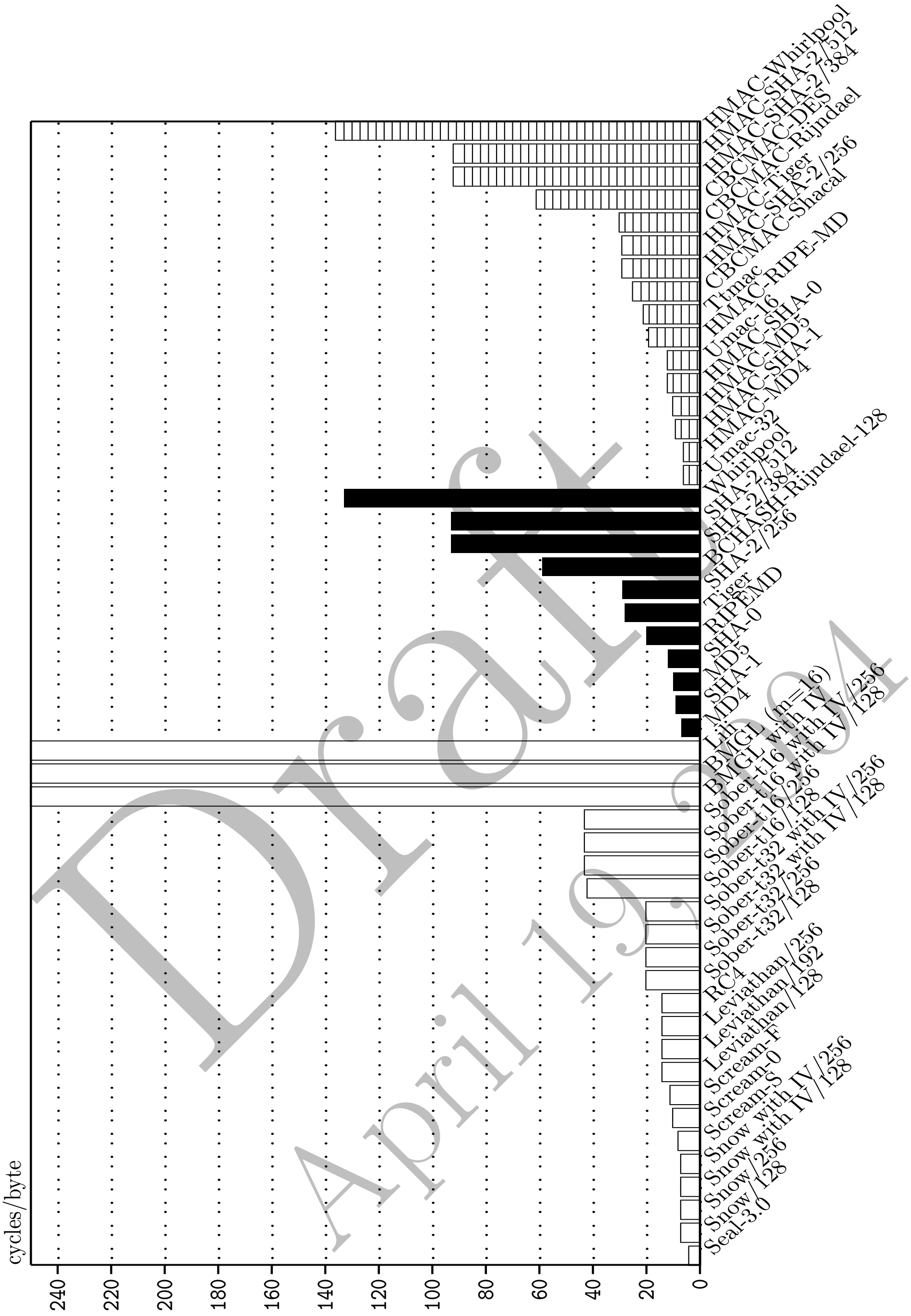


Fig. 44. Stream Ciphers, Hash Functions and MACs on Macintosh (Sorted)

□ stream ciphers ■ hash functions ▤ MACs

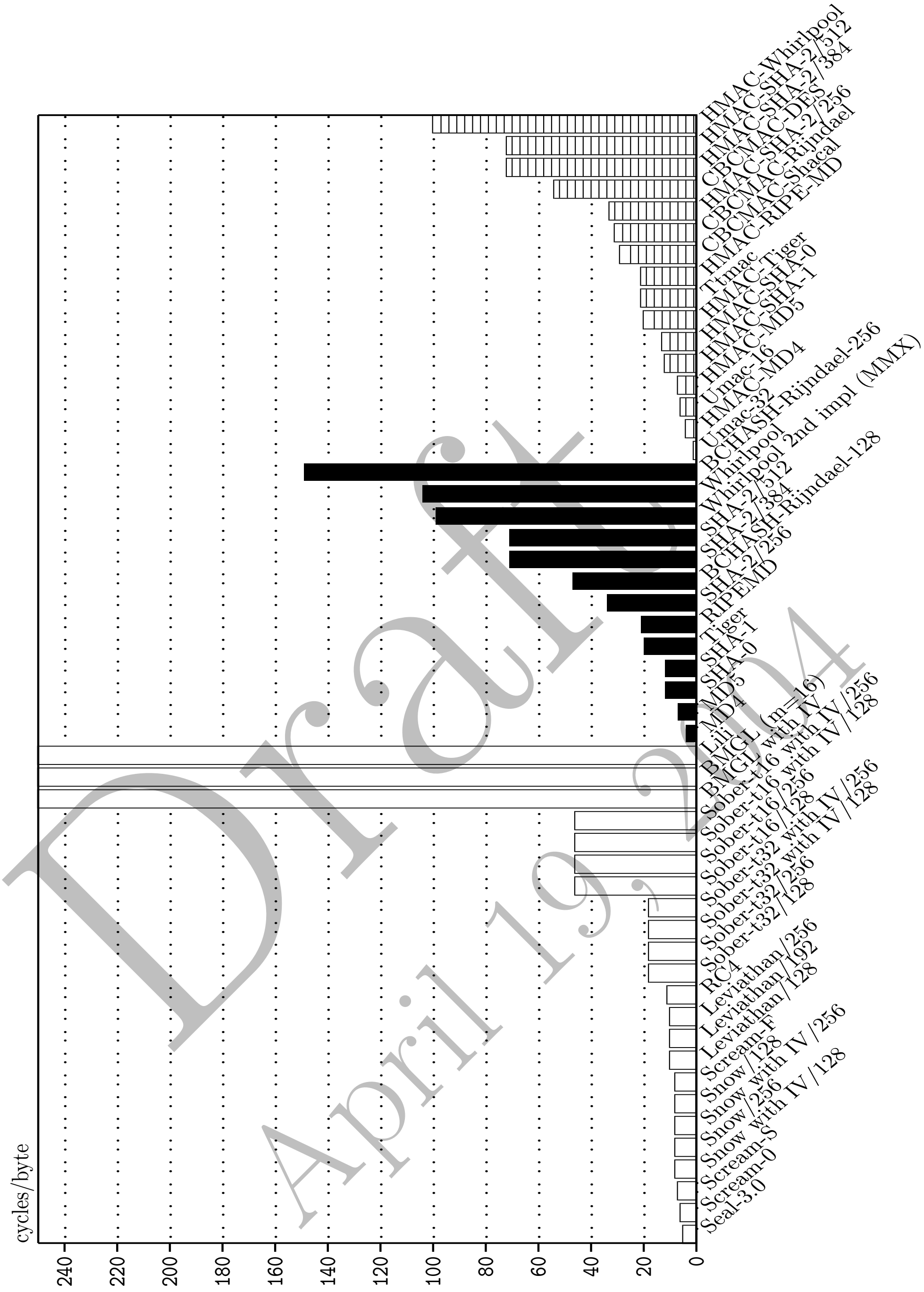


Fig. 45. Stream Ciphers, Hash Functions and MACs on AMD (Sorted)

□ stream ciphers ■ hash functions ▤ MACs

Part D

NESSIE selection for Phase II

1 Introduction

The NESSIE project is a three year project (2000-2002) that is funded by the European Union's *Fifth Framework Programme*. The main objective of the NESSIE project is to put forward a portfolio of strong cryptographic primitives of various types. Further details about the NESSIE project can be found at the NESSIE website `http://www.cryptonessie.org/`.

The start of the NESSIE project was an open call [475] for the submission of cryptographic primitives as well as for evaluation methodologies for these primitives. This call includes a request for the submission of block ciphers (as for the AES call), but also of other cryptographic primitives including hash functions, stream ciphers, and digital signature algorithms. The call also asked for evaluation methodologies for these primitives. The scope of the call was defined in conjunction with the project industry board, and was published in March 2000. This call resulted in forty submissions. The NESSIE project aims to assess these submissions with the goal of producing a portfolio of cryptographic primitives for use in different environments. The NESSIE project proposes to disseminate the project results widely and to build consensus based on these results by using the appropriate bodies: a project industry board, NESSIE workshops, the 5th Framework programme, and various standardisation bodies.

The NESSIE project has been divided into two phases. All primitives are evaluated in Phase I. At the end of Phase I, a subset of primitives is selected for further evaluation in Phase II. At the end of Phase II, a portfolio of primitives for possible standardisation will be chosen. To facilitate the open evaluation process, there are three NESSIE workshops. Submitted primitives were presented at the first NESSIE workshop, which took place on 13-14 November 2000 at K.U. Leuven (Belgium). Early results concerning the primitives were presented at the second NESSIE workshop, which took place on 12-13 September 2001 at Royal Holloway (U.K.). This workshop took place at the end of Phase I. The third workshop will take place at the end of Phase II (autumn 2002). In Phase I, both a security

Book II Part D of this final report was first published under the name ”NESSIE Phase I: Selection of Primitives” as a public NESSIE document.

evaluation and a performance evaluation of the submitted primitives were undertaken by the NESSIE partners. The NESSIE partners have also received many external comments about the submitted primitives. Reports on both the Phase I Security Evaluation [478], which also contains methodological comments, and the Phase I Performance Evaluation [477] are available on the NESSIE website. An overview of the methodology used by the NESSIE project is given in the Phase I Security review [478]
This document gives the selection of primitives made by the NESSIE project for further evaluation in Phase II. The NESSIE project will also consider relevant standards or proposed standards in Phase II, such as AES, CBC-MAC, H-MAC, DSS, SHA-1, SHA-256 and SHA-512.

2 Block Ciphers

We divide the discussion about block ciphers into normal-legacy (64-bit key) block ciphers and normal (128-bit key) or high (256-bit key) block ciphers.

2.1 Legacy Block Ciphers

- Legacy Block Ciphers selected for Phase II Evaluation
  - IDEA
  - Khazad
  - MISTY1
  - SAFER++ (64-bit block)

- Legacy Block Ciphers not selected for Phase II Evaluation
  - CS-Cipher
  - Hierocrypt-L1
  - Nimbus
  - NUSH

No security problems have been reported for IDEA, MISTY1, Khazad and SAFER++. IDEA benefits from having been scrutinised publicly for a decade without the detection of any serious weaknesses. MISTY1 has been in the public domain for five years, and the best attack breaks five of the suggested eight rounds. Khazad borrows elements from the AES, which makes it an attractive candidate for a 64-bit block cipher. Similarly SAFER++ (64-bit block) is a minor modification of the 128-bit block cipher SAFER++ (128-bit block).
CS-Cipher has no reported security problems, though it is the slowest of all the 64-bit submissions. Hierocrypt-L1 is slow, has problems with its key schedule and 3.5 out of its 6 rounds can be attacked. Nimbus has been broken in a chosen plaintext attack with 2^8 texts and 2^{10} time complexity. NUSH has an extremely low security margin, and it seems that a linear attack is faster than an exhaustive key search in the case of 256-bit keys.

2.2 Normal and high level Block Ciphers

- Normal and High Level Block Ciphers selected for Phase II Evaluation
  - SAFER++ (128-bit block)
  - Camellia
  - RC6

SHACAL

  -

- Normal and High Level Block Ciphers not selected for Phase II Evaluation
  - NUSH
  - Grand Cru
  - Noekeon
  - Q
  - Hierocrypt-3
  - SC2000
  - ANUBIS

No security problems have been reported for SAFER++ (128-bit block), Camellia, RC6, and SHACAL.
NUSH has an extremely low security margin, as described above. Grand Cru is based on AES, but is one of the slowest block ciphers submitted to NESSIE. Q can be attacked faster than an exhaustive key search. Noekeon has a related key attack for either of the submitted key schedules. Hierocrypt-3 also has key schedule problems, and there are attacks for up to 3.5 rounds out of 6. SC2000 has a mix of Feistel rounds and SP-network rounds; the benefits of and justification for this design are unclear. Anubis is very similar to the AES. Any advantages that ANUBIS might offer over the AES would not seem sufficient to suggest that ANUBIS would ever be selected as an alternative standard to the AES.

3 MAC and Hash Functions

- MAC and Hash Functions selected for Phase II Evaluation
  - Two-Track-MAC
  - UMAC
  - WHIRLPOOL

There have been no security problems reported for any of the submitted MAC and Hash Function primitives Two-Track-MAC, UMAC and WHIRLPOOL. All three have been selected for further evaluation in Phase II of the NESSIE process.

4 Stream ciphers

- Stream Ciphers selected for Phase II Evaluation
  - SOBER-t16 and SOBER-t32

- SNOW
- BMGL

– Stream Ciphers not selected for Phase II Evaluation
LILI-128
-
- LEVIATHAN

No security problems have been reported for the stream cipher submissions SOBER-t16, SOBER-t32, SNOW and BMGL.
LEVIATHAN possesses severe statistical problems, which have been verified experimentally. For LILI-128, there are attacks that are very much faster than brute force key space search.

5 Asymmetric Primitives

Primitives are selected for Phase II evaluation as detailed below. Phase II evaluation will take note of ongoing standardisation activities such as ISO and P1363.

5.1 Asymmetric Encryption

- Asymmetric Encryption Schemes selected for Phase II Evaluation
  - ACE Encrypt (revised version named ACE-KEM)
  - EPOC-2 (revised version)
  - PSEC-2 (revised version named PSEC-KEM)
  - ECIES
  - RSA-OAEP (if revised)
- Asymmetric Encryption Schemes not selected for Phase II Evaluation
  - EPOC-1 and EPOC-3
  - PSEC-1 and PSEC-3

Primitives based on finite field discrete logarithm

ACE Encrypt is proven to be secure without using the random oracle model but is not as flexible as the other schemes in that the only symmetric cipher it can be based on is MARS. ACE-KEM is a variant of ACE Encrypt with similar security, better performance, and also working in an elliptic curve group. ACE-KEM is supported by the submitters of ACE Encrypt.

Primitives based on elliptic curve discrete logarithm

PSEC-1 is not selected because it has worse security than PSEC-2 and similar performance. PSEC-2 as submitted to NESSIE has weaknesses. PSEC-KEM, a revised version of PSEC-2, is being considered by ISO and will be submitted to NESSIE. PSEC-KEM will be considered in Phase II. PSEC-KEM has an efficient reduction to a better asymmetric assumption, at the cost of a slower decryption, than PSEC-3. Furthermore, PSEC-3 is no longer supported by its submitters. Thus, PSEC-3 is not selected for Phase II.

ECIES and PSEC-3 are the schemes with the most efficient security reduction submitted in this category. Whilst the asymmetric component of ECIES is at least as secure as PSEC-3, it has stronger requirements for its symmetric components. ECIES is selected for Phase II. There is also an ECIES-KEM variant that will be compared to ECIES.

Primitives based on the factorisation problem

EPOC-1 is not selected for Phase II because it has worse security than EPOC-2 and similar performance. EPOC-2 has been revised in P1363 to fix some parameters and encoding methods. Compared to EPOC-3, EPOC-2 has an efficient reduction to a better asymmetric assumption, at the cost of a slower decryption. Furthermore, EPOC-3 is no longer supported by its submitters. Thus EPOC-3 is not selected for Phase II. EPOC-2 is selected for Phase II because it is the only submitted primitive based on factoring and it has efficient and convincing security.

RSA-OAEP is selected for Phase II. However, the OAEP padding has weaknesses and the submission should be modified to use another technique with a better security proof. We are aware of four techniques in the literature that reduce RSA encryption to the RSA problem: OAEP+ [583] and SAEP+ [103] have a bad reduction, REACT [500] and KEM [584] have a tight reduction. Recent results have shown that a Rabin-SAEP scheme [103] has better properties than the RSA-based schemes, as its security has a very efficient reduction to factoring instead of RSA inversion, and it has better performance than RSA-based schemes, though a bad implementation can reveal the secret key.

5.2 Digital Signature Schemes

- Digital Signature Schemes selected for Phase II Evaluation
  - ECDSA
  - ESIGN (revised version)
  - RSA-PSS
  - SFLASH
  - QUARTZ (depending on application)
- Digital Signature Schemes not selected for Phase II Evaluation
  - ACE Sign
  - FLASH

Primitives based on the RSA problem.

ACE Sign and RSA-PSS are both based on the difficulty of factorisation. While both are secure in the random oracle model if the RSA problem is hard (with a much tighter reduction for RSA-PSS), ACE Sign is also secure if the Strong-RSA problem is hard without any random oracle assumption. This additional security property has a high performance cost, which implies that ACE Sign with a 1024-bit modulus has similar performance to RSA-PSS with a 3000-bit modulus. Thus RSA-PSS is selected for Phase II, and ACE Sign is not selected for Phase II.

Primitives based on elliptic curve discrete logarithm problem

ECDSA is selected for Phase II. There have been no reported security problems, and ECDSA creates shorter signatures than RSA-PSS with a shorter signing time with an equivalent verification time.

Primitives based on the approximate e-th root modulo p^2q

The security of ESIGN is based on a similar but stronger assumption than RSA. It has similar performance to ECDSA, and its security also seems to be similar. ESIGN has been revised in P1363 to change some parameters and encoding methods so that signature generation for long messages requires only one pass through a hash function. ESIGN is selected for Phase II.

Primitives based on multivariate quadratic polynomials

FLASH and SFLASH are both C^{*--} schemes targeted to a smart card environment. SFLASH is selected for Phase II and FLASH is not. SFLASH has similar security and similar performance as FLASH, but with a much smaller public key. QUARTZ has very slow signature generation but the resulting signature is only 128 bits long. If such short signatures are considered useful in standardised applications, then QUARTZ should be selected for Phase II. The fact that these schemes do not have provable security is not a significant problem for applications which need signatures with only a short validity period.

5.3 Digital Identification Schemes

- Digital Identification Schemes selected for Phase II Evaluation
  - GPS

GPS, the only primitive submitted to NESSIE in this category, has good performance with high security. The submitted documents contain some minor flaws in the specification, but these have been corrected. GPS is selected for further evaluation in Phase II.

Part E

NESSIE portfolio

1 Introduction

This document presents and motivates the NESSIE portfolio of recommended cryptographic primitives.Further technical information supporting these decisions can be found in the NESSIE security [481] and performance [480] evaluation documents.

The information in this document is provided as is, and no guarantee or warranty is given or implied that the information is fit for any particular purpose. The user thereof uses the information at its sole risk and liability.

2 Block Cipher Encryption Schemes

2.1 64-bit Block Ciphers

NESSIE portfolio. The 64-bit block cipher included in the NESSIE portfolio is MISTY1.

– The NESSIE project did not find an attack on MISTY1. Furthermore, MISTY1 is similar to the block cipher KASUMI, so much of the analysis for KASUMI would also be applicable to MISTY1. KASUMI has been scrutinised prior to its adoption as a 3GPP standard. However, many NESSIE partners are concerned that the simple algebraic structure of MISTY1 may lead to future breakthroughs in the analysis of MISTY1.

Comments on the other 64-bit block ciphers studied in Phase II.

– No attacks were found on Khazad and in the opinion of the NESSIE project it is an interesting primitive for future research. However, concerns were expressed with regard to the structural symmetry of Khazad.

Book II Part E of this final report was first published under the name ”Portfolio of recommended cryptographic primitives” as a public NESSIE document.

- The NESSIE project did not find an attack on IDEA. However, it was not selected because of Intellectual Property Rights issues and some concerns about its key schedule.
- The NESSIE project did not find an attack on SAFER++ (64-bit block). However, SAFER++ (64-bit block) was not selected because there were some concerns about certain structural properties of SAFER++ (64-bit block), as discussed in the NESSIE Security Evaluation Report. It was also found to be slower than the other ciphers except in smart cards.
- The NESSIE project considers 3-DES to be a secure but slow block cipher.

2.2 128-bit Block Ciphers

NESSIE portfolio. The 128-bit block ciphers included in the NESSIE portfolio are the AES and Camellia.

- The AES has been scrutinised by the U.S. National Institute of Standards and Technology as a secure block cipher and adopted as a U.S. Federal Information Processing Standard. Camellia has many similarities to the AES, so much of the analysis for the AES is also applicable to Camellia. It is also the case that the NESSIE project did not find an attack on either the AES or Camellia. However, the NESSIE partners, as well as the wider cryptographic community, have a wide range of views about the AES and Camellia. Many NESSIE partners have significant concerns that the simple algebraic structure of the AES, and to a somewhat lesser extent Camellia, may lead to future breakthroughs in the analysis of these block ciphers.

Comments on the other 128-bit block ciphers studied in Phase II.

- The NESSIE partners felt unable to consider the selection of RC6 owing to ongoing serious Intellectual Property Rights issues.
- The NESSIE project did not find an attack on SAFER++ (128-bit block). However, the NESSIE project did not select SAFER++ (128-bit block) because of some concerns, both about certain structural properties of SAFER++ (128-bit block) and about the low security margin of SAFER++ (128-bit block), as discussed in the NESSIE Security Evaluation Report.

2.3 256-bit Block Ciphers

NESSIE portfolio. The 256-bit block cipher included in the NESSIE portfolio is SHACAL-2.

Comments on the other block ciphers with block length larger than 128 bits studied in Phase II.

- The NESSIE partners felt unable to consider the selection of RC6 owing to ongoing serious Intellectual Property Rights issues.
- The NESSIE project did not select SHACAL-1 because of concerns about its key schedule.

3 Stream Ciphers and Pseudorandom Number Generators

NESSIE portfolio. The NESSIE portfolio in this category is empty.

Comments on the stream ciphers studied in Phase II.

- NESSIE does not recommend SNOW for encryption, because there are distinguishing attacks and guess and determine attacks faster than exhaustive key search.
- For SOBER-t16 and SOBER-t32 there are distinguishing attacks faster than exhaustive key search. Owing to the irregular decimation of SOBER-t16 and SOBER-t32 there are certain reservations with respect to the vulnerability of implementations of these algorithms with respect to side-channel attacks. NESSIE, therefore, does not recommend either primitive.
- The first drawback of BMGL is its small internal state, which makes it vulnerable to a time-memory tradeoff attack. In addition, BMGL is too slow for encryption, so NESSIE does not recommend BMGL.

4 Collision-Resistant Hash Functions

NESSIE portfolio. The collision-resistant hash functions included in the NESSIE portfolio are Whirlpool, SHA-256, SHA-384 and SHA-512.

- The NESSIE project selects Whirlpool as a collision-resistant hash function, with an output length of 512 bits. The design of Whirlpool is based on an underlying 512-bit block cipher that is used in Myaguchi-Preneel mode. This block cipher has a structure similar to Rijndael. The best known attack on Whirlpool finds non-random properties when the compression function is reduced to six rounds or less (out of ten); this gives a good security margin. The performance of Whirlpool is acceptable, though on most platforms it is slightly slower than SHA-2/512.
- The NESSIE project selects SHA-256, SHA-384 and SHA-512 as collision-resistant hash functions, with an output length of 256, 384 or 512 bits. These algorithms have recently been added to the NIST standard for hash functions. In contrast to the AES process this was not an open standardisation process and the design strategy was not made public. These algorithms are rather a new designs that have some similarities to SHA-1 but there are important differences in the structure. They were not submitted to NESSIE and owing to a lack of resources only limited evaluation was done for them. Current results indicate no security problems and these primitives seems to have a large security margin against known attacks. The performance of these algorithms is acceptable, SHA-512 and SHA-384 being slightly faster than Whirlpool on most platforms. SHA-256 is about twice as fast on most platforms.

No security weaknesses were found for these primitives. However, Whirlpool, SHA-256, SHA-384 and SHA-512 are newly designed primitives which have undergone only limited evaluation by the cryptographic community so far.

NESSIE comments on "legacy" hash functions. The standard primitives SHA-1 and RIPEMD-160 do not meet the NESSIE security requirement for symmetric primitives, because their output is only 160 bits, but they can be recommended for applications where this security level is sufficient.

5 Message Authentication Codes

NESSIE portfolio. The message authentication codes included in the NESSIE portfolio are UMAC, TTMAC, EMAC and HMAC.

- For the authentication of long message streams UMAC is by far the fastest of the MAC primitives considered by NESSIE (at the cost of greater complexity and worse key-agility compared to the other primitives). UMAC is based on universal hash function families and has provable security: a break of the primitive would imply a break of the block cipher that is used by the scheme as a pseudo-random function (the current specification chooses AES as block cipher).
- TTMAC (also known as Two-Track-MAC) has the highest security level of the MAC primitives considered by NESSIE. The design of TTMAC is based on the hash function RIPEMD-160 (with small modifications). The security can be proven on the assumption that the underlying compression function is pseudo-random. TTMAC has specific performance advantages: it is especially efficient in the case of short messages, and has optimal key-agility.
- EMAC (also known as DMAC) has the advantage that it allows the reuse of an existing block cipher implementation (in CBC-mode with an extra encryption as output transformation). The security can be proven on the assumption that the underlying block cipher is pseudo-random. The performance and key-agility are reasonable (EMAC is preferable for short messages because the block length is smaller compared to the schemes based on a hash function). NESSIE recommends the use of this construction with a 128-bit block cipher included in the NESSIE portfolio.
- HMAC has the advantage that it allows the reuse of an existing hash function implementation. The security can be proven on the following assumptions: the underlying hash function is collision-resistant for a secret initial value; the compression function keyed by the initial value is a secure MAC primitive (for messages of one block); the compression function is a weak pseudorandom function. These assumptions are weaker than the assumptions required for TTMAC and EMAC. The performance and key-agility are reasonable. NESSIE recommends the use of this construction with a collision-resistant hash function included in the NESSIE portfolio.

No security weaknesses were found for any of these primitives. NESSIE makes a broad recommendation in this area because every primitive has its own specific advantages.

6 Asymmetric encryption schemes

NESSIE portfolio. The asymmetric encryption schemes included in the NESSIE portfolio are PSEC-KEM, RSA-KEM and ACE-KEM.

– The primary recommendation is PSEC-KEM. Its security is based on the Computational Diffie-Hellman assumption with an efficient proof of security. From a performance point of view, it compares favourably with other schemes that offer a similar security level. The elliptic curve should be carefully chosen and the base field should be at least of size 160 bits, which should be sufficient for medium term security (5 to 10 years). A prime field is preferable, unless implementation constraints favour a field of characteristic 2.
– A secondary recommendation is RSA-KEM with exponent at least 65537 and public keys of at least 1536 bits, which should be sufficient for medium term security (5 to 10 years). Exponent 3 can be used if fast encryption is important. Its security is based on the RSA assumption with an efficient proof, but it has a relatively slow decryptionand longer keys than PSEC-KEM. It may be more difficult to protect implementations of RSA-KEM against side-channel attacks than implementations of PSEC-KEM.
– ACE-KEM is recommended where performance is not critical. It has several provable security arguments, and therefore its security is better than that of the other encryption schemes. Depending on the application, either a 160-bit (or more) elliptic curve or a 1536-bit (or more) prime field can be used.

Comments on the other asymmetric encryption schemes studied in Phase II.

– ECIES and ECIES-KEM have slightly better performance than PSEC-KEM, but the Gap Diffie-Hellman security assumption makes the security proof less convincing than PSEC-KEM.
– EPOC-2 compares unfavourably with Rabin-based schemes such as HIME(R) and Rabin-SAEP.

NESSIE recommendation if very long term security is important. For very high level security we note that double encryption using ACE-KEM and RSA-KEM with different DEMs gives a good range of security, based on various different assumptions. Triple encryption that also uses a public-key scheme not based on number-theoretical assumptions might increase the security against future breakthrough.

7 Digital signature schemes

NESSIE portfolio. The digital signature schemes included in the NESSIE portfolio are RSA-PSS, ECDSA and SFLASH.

– The primary recommendation is a digital signature scheme based on the RSA-PSS submission to NESSIE with exponent at least 65537 and public keys of at

least 1536 bits, which should be sufficient for medium term security (5 to 10 years). Exponent 3 can be used if fast verification is important.
- A secondary recommendation is ECDSA. This scheme is well suited to applications where the signing time or the appendix length are important. The elliptic curve should be carefully chosen and the base field should be at least of size 160 bits, which should be sufficient for medium term security (5 to 10 years). A prime field is preferable, unless implementation constraints favour a field of characteristic 2.
- SFLASH is not recommended for general use but this signature scheme is very efficient on low cost smart cards, where the size of the public key is not a constraint.

If this causes no interoperability problems, a tweak of the submitted schemes is strongly recommended. One should include some certification data in the inputs of the hash functions. This would bind the scheme parameters, public key and expiration time, to a particular signature. (This was a proposal of the KCDSA scheme.)

Comments on the other digital signature schemes studied in Phase II.

- The ESIGN family of digital signature schemes has convincing security and some performance advantages, but RSA-PSS and ECDSA have more convincing security and cover most common applications. Both ESIGN-D and ESIGN-R may be suited to specific uses.
- QUARTZ does not meet our security requirements for the submitted parameters. A modification of the parameter d might be sufficient, but is not fully evaluated. Still, if a digital signature scheme with appendix shorter than 250 bits is needed, QUARTZ with a larger d can be used.

NESSIE recommendation if very long term security is important. Use simultaneously different digital signature schemes, based on different mathematical assumptions and distinct hash functions.

8 Asymmetric Identification Schemes

NESSIE portfolio. The asymmetric identification scheme included in the NESSIE portfolio is GPS.

- GPS is the only primitive submitted to NESSIE in the category of asymmetric identification schemes and has good performance with high security. It compares favourably with the other zero-knowledge identification schemes that have been described in the literature.
  For medium term security (5 to 10 years) the modulus should have at least 1536 bits and the other parameters as recommended by the submitters of the scheme. The flexibility of GPS allows this scheme to be used in a variety of situations, and different security parameters might be appropriate.

A trivial protocol flaw was found in the original submitted version, and was corrected. Implementations of GPS should not forget any item of the protocol, as a mistake may have serious security implications.

Book III

The NESSIE portfolio

Organization of Book III

For each primitive in the NESSIE portfolio, we give a complete description and test vectors, that should help to validate an implementation.

While the previous Books in this report were written to support the selection made by NESSIE and were aimed at cryptologists, Book IIIis aimed at implementors of cryptographic primitives and is self-contained.

Part A

Definitions and notations

1. Mathematical objects

1.1 Notations

$\mathtt{0}_\mathtt{b}$	The bit 0.
$\mathtt{0100}_\mathtt{b}$	A bit string of length 4.
$\mathtt{09}_\mathtt{x}$	An octet.
$\mathtt{09AB}_\mathtt{x}$	The octet string of length 2 with first octet $\mathtt{09}_\mathtt{x}$ and last octet $\mathtt{AB}_\mathtt{x}$.
$X\|Y$	The concatenation of two bit strings or two octet strings X and Y.
$\overline{X}$	The bitwise negation of X
$X \oplus Y$	The bitwise exclusive-or (XOR) of two bit strings or two octet strings X and Y of same length.
$X \vee Y$	The bitwise OR of X and Y (having same length).
$X \wedge Y$	The bitwise AND of X and Y (having same length).
$a \vdash a$	The truncation of X to the a leftmost bits.
$X \lll_a$	The cyclic left-rotation by a bits of X
$X \ggg_a$	The cyclic right-rotation by a bits of X
$X \gg_a$	The right-shift by a bits of X.
$\lceil x \rceil$	The smallest integer greater than or equal to x.
$\lfloor x \rfloor$	The larger integer smaller than or equal to x.
$\log_2 x$	The logarithmic function of a number x to the base 2.
$\mathbb{F}_q$	The finite field containing q elements.
$E(\mathbb{F}_q)$	An elliptic curve defined over the finite field $\mathbb{F}_q$.
$\#E(\mathbb{F}_q)$	The order of the elliptic curve $E(\mathbb{F}_q)$.
$\mathcal{O}$	The point at infinity on an elliptic curve.

1.2 Finite fields

A *finite field* (or *Galois field*) is a field containing a finite number of elements. The *order* of a finite field is the number of elements it contains. A finite field of order $q > 1$ exists if and only if q is a power of a prime number. Furthermore, a finite field of given order q is "unique", meaning that two such fields share the same algebraic structure. This field is denoted $\mathbb{F}_q$ (or $GF(q)$). On the other hand, there exist several different representations for a given field, leading to different ways to perform arithmetic. In cryptography, we are mainly interested in the finite fields $\mathbb{F}_q$ where q is either a prime number (called *prime fields*), or a power of 2 (called *characteristic 2 finite fields*). We will see in the following how they are commonly represented.

1.2.1 Prime finite fields

A prime finite field $\mathbb{F}_p$ is represented as the set of integers $\{0, 1, \ldots, p-1\}$, where the operations are performed modulo p. The addition is defined for all $x, y \in \mathbb{F}_p$ by $x + y = r$ where r is the remainder when the integer $x + y$ is divided by p, and the multiplication by $xy = t$ where t is the remainder when the integer xy is divided by p.

1.2.2 Characteristic 2 finite fields

For an integer $m > 0$, the elements of the field $\mathbb{F}_{2^m}$ are the 2^m possible bit strings of length m. For $a = (a_{m-1}a_{m-2} \ldots a_1a_0)$ and $b = (b_{m-1}b_{m-2} \ldots b_1b_0)$ in $\mathbb{F}_{2^m}$, the addition is the bitwise XOR, i.e.:

$$a + b = c = (c_{m-1}c_{m-2} \ldots c_1c_0), \text{ with } c_i = a_i \oplus b_i.$$

We see that addition is easy to implement since the bit string c is obtained by XORing the bit strings a and b. Elements of $\mathbb{F}_{2^m}$ can also be seen as m-dimensional vectors over $\mathbb{F}_2$, and the addition in $\mathbb{F}_{2^m}$ is the addition in $(\mathbb{F}_2)^m$.

The implementation of the multiplication depends on the representation of $\mathbb{F}_{2^m}$ we choose, i.e we have to specify the way the bit strings are interpreted.

A *polynomial basis representation* of $\mathbb{F}_{2^m}$ is determined by choosing an irreducible polynomial $f(X)$ of degree m. The field is then represented as the set of polynomials of maximal degree $m-1$, with coefficients in $\mathbb{F}_2$. More explicitly, the bit string $(b_{m-1}b_{m-2} \ldots b_1b_0)$ is represented by the binary polynomial $b_{m-1}X^{m-1} + b_{m-2}X^{m-2} + \ldots + b_1X + b_0$. The operations on these polynomials are performed in $\mathbb{F}_2(X)$, modulo the polynomial $f(X)$, which is called the *reduction polynomial*. More precisely, for $a = a_{m-1}X^{m-1} + \ldots + a_1X + a_0$ and $b = b_{m-1}X^{m-1} + \ldots + b_1X + b_0$ in $\mathbb{F}_{2^m}$,

The multiplication is defined by:

$$ab = d = d_{m-1}X^{m-1} + \ldots + d_1X + d_0$$

where d is the remainder when the polynomial ab is divided by $f(X)$, and all the operations on the coefficients are performed modulo 2.

In this representation, the additive identity element is the polynomial 0, and the multiplicative identity element is the polynomial 1.

1.3 Elliptic curve points

Elliptic curves are a subset of the cubic plane curves, i.e. they are defined to be the set of points P which coordinates (x, y) verify some equation $F(x, y) = 0$ of degree 3. In cryptography, we are mainly interested in elliptic curves defined over a finite field $\mathbb{F}_q$, where q is a prime or a power of 2. An elliptic curve over the finite field $\mathbb{F}_q$ is a set of points $P = (x_P, y_P)$, where x_P and y_P are elements of $\mathbb{F}_q$ verifying an equation with coefficients in $\mathbb{F}_q$, plus a special point $\mathcal{O}$ called the *point at infinity*. The most common equations used to define elliptic curves are the *Weierstrass equations*. They depend on whether the field $\mathbb{F}_q$ is a prime field or a characteristic 2 finite field:

– if q is an odd prime with $q > 3$, the Weierstrass equation is

$$y^2 = x^3 + ax + b,$$

where a and b are elements of $\mathbb{F}_q$ satisfying $4a^3 + 27b^2 \neq 0$ in $\mathbb{F}_q$;

– if q is a power of 2, the Weierstrass equation is

$$y^2 + xy = x^3 + ax^2 + b,$$

where a and b are elements of $\mathbb{F}_q$ with $b \neq 0$.

The so-defined elliptic curve, denoted $E(\mathbb{F}_q)$, is thus characterised by a and b, which are called the *coefficients* of the curve. The number of points on $E(\mathbb{F}_q)$ is called the *order* of $E(\mathbb{F}_q)$ and is denoted $\#E(\mathbb{F}_q)$.

It is possible to define an additive operation on the points of an elliptic curve that possesses the properties of a commutative group rule. This operation can be defined geometrically. We define the *inverse* of a point $P = (x_P, y_P)$ to be the point $Q = -P$ where:

$$x_Q = x_P, \quad y_Q = \begin{cases} -y_P & \text{if } q = p \text{ prime} \\ x_P + y_P & \text{if } q = 2^m \end{cases}$$

The *sum* of two points P and Q of $E(\mathbb{F}_q)$ is then defined as the point R of the curve such that $-R$ lies on the line passing through P and Q. When $P = Q$, the line we consider is the tangent line at the point P. The point at infinity $\mathcal{O}$ plays the role of the number 0 in the ordinary addition, so we have $P + \mathcal{O} = P$ and $P + (-P) = \mathcal{O}$ for all point P of the curve.

The coordinates of the resulting point can be expressed with the coordinates of the points to add. Let $R = (x_R, y_R)$ be the sum of two distinct points $P = (x_P, y_P)$ and $Q = (x_Q, y_Q)$, with $x_P \neq x_Q$ (otherwise $P+Q = \mathcal{O}$). The expression depends on whether the field is a prime field, or a finite characteristic 2 field.

– If q is an odd prime number, we have

$$x_R = \lambda^2 - x_P - x_Q \text{ in } \mathbb{F}_p, \; y_R = \lambda(x_P - x_R) - y_P \text{ in } \mathbb{F}_p,$$

$$\text{where } \lambda = \frac{y_Q - y_P}{x_Q - x_P} \text{ in } \mathbb{F}_p.$$

– If $q = 2^m$, we have

$$x_R = \lambda^2 + \lambda + x_P + x_Q + a \text{ in } \mathbb{F}_{2^m}, \; y_R = \lambda(x_P + x_R) + x_R + y_P \text{ in } \mathbb{F}_{2^m},$$

$$\text{where } \lambda = \frac{y_Q + y_P}{x_Q + x_P} \text{ in } \mathbb{F}_{2^m}.$$

Now, let $R = (x_R, y_R)$ be the sum of $P = (x_P, y_P)$ with itself. This operation is called *doubling*, and the coordinates of the resulting point are as follows.

– If q is an odd prime number, we have

$$x_R = \lambda^2 - 2x_P \text{ in } \mathbb{F}_p,\ y_R = \lambda(x_P - x_R) - y_P \text{ in } \mathbb{F}_p,$$

$$\text{where } \lambda = \frac{3{x_P}^2 + a}{2y_P} \text{ in } \mathbb{F}_p.$$

– If $q = 2^m$, we have

$$x_R = \lambda^2 + \lambda + a \text{ in } \mathbb{F}_{2^m},\ y_R = {x_P}^2 + (\lambda + 1)x_R \text{ in } \mathbb{F}_{2^m},$$

$$\text{where } \lambda = x_P + \frac{y_P}{x_P} \text{ in } \mathbb{F}_{2^m}.$$

The *scalar multiplication* is the multiplication of an elliptic curve point P by an integer k defined by:

$$kP = \underbrace{P + P + \ldots + P}_{k \text{ times}}$$

2. Data type conversions

2.1 Integers

Converting integers into octet strings should be done according to the following method.

Converting integers to octet strings: I2OSP.

```
Input:   an integer $x$.
         a length $Slen$ with $256^{Slen} > x$.
Output:  an octet string $S$ of length $Slen$.
  1. write the unique decomposition of $x$ in base 256:
       $x = x_{Slen-1}256^{Slen-1} + x_{Slen-2}256^{Slen-2} + \ldots + x_1 256 + x_0;$
  2. for $0 \leq i \leq Slen - 1$, let $S_i$ be an octet and set:
       $S_i := x_i;$
  3. return the octet string $S = S_{Slen-1}S_{Slen-2} \ldots S_1 S_0$.
```

Converting octet strings to integers: OS2IP.

```
Input:   an octet string $S = S_{Slen-1}S_{Slen-2} \ldots S_1 S_0$ of length $Slen$.
Output:  an integer $x$.
  1. set
       $x := \sum_{i=0}^{Slen-1} 2^{8i} S_i;$
  2. return $x$.
```

2.2 Finite fields

As we have seen, an element of a finite field $\mathbb{F}_q$ is represented as an integer x in the set $\{0, \ldots, p-1\}$ if $q = p$ an odd prime, and as a bit string of length m if $q = 2^m$. So, the conversion between field elements and octet strings depends on the field.

Converting a finite field element to an integer FE2IP.

Input: an element a of the finite field $\mathbb{F}_q$.
Output: an integer x.

1. If $q = p$ an odd prime, there is a unique integer $x \in \{0, ..., p-1\}$ such that $a \equiv x \bmod p$. Return x.
2. If $q = 2^m$, a is a bit string $a = (a_{m-1}a_{m-2} \ldots a_1a_0)$. Let x be the integer defined by: $x := \sum_{i=0}^{m-1} a_i 2^i$. Return x.

Converting an integer to a finite field element: IP2FEP.

Input: an integer x.
Output: an element a of the finite field $\mathbb{F}_q$.

1. If $q = p$ an odd prime: If $x \notin \{0, \ldots, p-1\}$, return "`invalid`". Else, return x.
2. If $q = 2^m$: If $x \geq 2^m$, return "`invalid`". Else, write $x = \sum_{i=0}^{m-1} a_i 2^i$ and return the bit string $a := a_{m-1}a_{m-2} \ldots a_1a_0$.

Converting a finite field element to an octet string: FE2OSP.

Input: an element a of the finite field $\mathbb{F}_q$.
Output: an octet string S of length $Slen = \lceil \log_2 q/8 \rceil$.

1. Compute $x := \mathsf{FE2IP}(a)$.
2. Compute $S := \mathsf{I2OSP}(x, \lceil \log_2 q/8 \rceil)$.
3. Return S.

Converting an octet string to a finite field element: OS2FEP.

Input: an octet string S of length $Slen = \lceil \log_2 q/8 \rceil$.
Output: an element a of the finite field $\mathbb{F}_q$.

1. Compute $x := \mathsf{OS2IP}(a)$.
2. Compute $S := \mathsf{IP2FEP}(x)$.
3. Return S.

2.3 Elliptic curve points

An elliptic curve point should be converted into an octet string following one of the methods below. The first one is the basic method, which idea is to convert each coordinate of the point to an octet string using the FE2OSP routine. The second method uses the so-called *point compression*. The idea is to compress one coordinate of the point into 1 bit. The full value can be recovered from the other coordinate and the coefficients of the curve.

Converting elliptic curve points to octet strings: ECP2OSP.

Basic algorithm.

```
Input:    a point $P = (x_P, y_P)$.
Output:   an octet string $S$ of length $Slen$ octets, with:
          – $Slen = 1$ if $P = \mathcal{O}$;
          – $Slen = 2\lceil(\log_2 q)/8\rceil + 1$ if $P \neq \mathcal{O}$.
  1. if $P = \mathcal{O}$, return $S := \texttt{00}_{\texttt{x}}$;
  2. else, perform the following steps:
     a) convert $x_P$ to an octet string $X := \mathsf{FE2OSP}(x_P)$ ,
     b) convert $y_P$ to an octet string $Y := \mathsf{FE2OSP}(y_P)$,
     c) return $S := \texttt{04}_{\texttt{x}} \| Y \| X$.
```

Using point compression algorithm.

```
Input:    a point $P = (x_P, y_P)$.
Output:   an octet string $S$ of length $Slen$ octets, with:
          – $Slen = 1$ if $P = \mathcal{O}$;
          – $Slen = \lceil(\log_2 q)/8\rceil$ if $P \neq \mathcal{O}$.
  1. if $P = \mathcal{O}$, return $S := \texttt{00}_{\texttt{x}}$;
  2. else, perform the following steps:
     a) convert $x_P$ to an octet string $X := \mathsf{FE2OSP}(x_P)$,
     b)  i. if $q = p$ an odd prime: $\tilde{y_P} = y_P \bmod 2$,
            if $q = 2^m$: $\tilde{y_P} = z_0$, where $z = z_{m-1}X^{m-1} + z_{m-2}X^{m-2} + \ldots +$
            $z_1X + z_0$ is defined by $z = y_P {X_P}^{-1}$;
        ii. if $\tilde{y_P} = 0$, $Y := \texttt{02}_{\texttt{x}}$,
            if $\tilde{y_P} \neq 0$, $Y := \texttt{03}_{\texttt{x}}$;
       iii. return $S = Y \| X$.
```

Converting octet strings to elliptic curve points: OS2ECPP. This routine converts an octet string into an elliptic curve point, this point being compressed or not, and checks that the point is on the right curve.

```
Input:    the coefficients $a$ and $b$ of an elliptic curve $E(\mathbb{F}_q)$, and an octet
          string $S$ of length $Slen$ octets.
Output:   a point $P = (x_P, y_P)$.
  1. If $S = \texttt{00}_{\texttt{x}}$, return $P = \mathcal{O}$;
  2. If $Slen = \lceil(\log_2 q)/8\rceil + 1$, perform the following steps:
     a) Parse $S$ as $Y \| X$, where $Y$ is a single octet, and $X$ has length
        $\lceil(\log_2 q)/8\rceil$.
     b) Convert the octet string $X$ to a field element using the routine
        OS2FEP. If the routine returns "invalid", return "invalid".
     c) If $Y = \texttt{02}_{\texttt{x}}$, set $\tilde{y}_p = 0$.
        If $Y = \texttt{03}_{\texttt{x}}$, set $\tilde{y}_p = 1$.
        Else, return "invalid".
     d) Decompress the point $(x_P, y_P)$ from $x_P$ and $\tilde{y}_P$:
         i. If $q = p$ an odd prime:
            A. Compute the element $\alpha = {x_P}^3 + ax_P + b$ in $\mathbb{F}_p$.
            B. Compute a square root of $\alpha$ in $\mathbb{F}_p$.
               If there are no square root of $\alpha$ in $\mathbb{F}_P$, return "invalid".
               Else, let $\beta$ be this square root.
            C. If $\beta \equiv \tilde{y}_P \pmod 2$, $y_P := \beta$.
               Else, $y_P := p - \beta$.
         ii. If $q = 2^m$ and $x_P = 0$, set $y_P := b^{2^{m-1}}$.
        iii. If $q = 2^m$ and $x_P \neq 0$:
            A. Compute the element $\gamma = x_P + a + b x_P^{-2}$ in $\mathbb{F}_{2^m}$.
            B. Compute an element of $\mathbb{F}_{2^m}$

               $$z = z_{m-1} X^{m-1} + z_{m-2} X^{m-2} + \ldots + z_1 X + z_0$$

               that verifies the equation $z^2 + z = \gamma$ in $\mathbb{F}_{2^m}$.
               If no such element exists, return "invalid".
            C. If $z_0 = \tilde{y}_P$, set $y_P := x_P z$ in $\mathbb{F}_{2^m}$.
               Else, set $y_P := x_P(z + 1)$ in $\mathbb{F}_{2^m}$.
     e) Return $P = (x_P, y_P)$.
  3. If $mLen = 2\lceil(\log_2 q)/8\rceil + 1$, perform the following steps:
     a) Parse $S$ as $W \| X \| Y$, where $w$ is a single octet, $X$ and $Y$ are
        $\lceil(\log_2 q)/8\rceil$-octet strings.
     b) If $W \neq 04_{\mathtt{x}}$, return "invalid".
     c) Convert the octet string $X$ to a field element $x_P$ using the routine
        OS2FEP. If the routine returns "invalid", return "invalid".
     d) Convert the octet string $Y$ to a field element $y_P$ using the routine
        OS2FEP. If the routine returns "invalid", return "invalid".
     e) Check that coordinates of the point $P = (x_P, y_P)$ verify the defining
        equation of the curve. If not, return "invalid".
     f) Return $p = (x_P, y_P)$.
```

3. Key derivation functions

Some of the asymmetric encryption schemes (see Part E) require a key derivation function (KDF) or a mask generating function (MGF). Key derivation functions and mask generating functions take as input octet strings of arbitrary length and an output length, and output an octet string of the prescribed length. For any fixed length, these functions should have the same security properties as a hash function.

The NESSIE project is not recommending any specific key derivation functions or mask generating functions for use however all of the submitted schemes used one of two mechanisms which we will detail here. These functions will be used to generate test vectors for the primitives. No weaknesses have been found in either of the primitives specified here. For further information the reader is referred to Book II Part B [481].

Both functions use a hash function *Hash* which takes as input octet strings of arbitrary length and outputs octet strings of length *HashLen*. The hash function SHA-1 can be used for this application but it would be preferable to use a hash function with a longer output (see Part C). Note that in the description of the functions we will assume that a hash function can process an input of any length. In reality, however, a hash function may output "error" if the length of the input is greater than some certain (very large) bound. In this case the KDF or MGF should output "error" and abort.

The first KDF/MGF specified will be referred to as $Mech1$. It runs as follows.

Description:

```
Input:    An input octet string $X$ of any length
          An output length $Len$
Output:   An octet string $Y$ of length $Len$

1. If $Len > 2^{32} HashLen$ then output "error" and abort.
2. Set $k := \lceil Len/HashLen \rceil$.
3. Set $Y$ to be the empty string.
4. For $i$ from 0 to $k-1$ do
   a) $Y := Y || Hash(X || \mathsf{I2OSP}(i, 4))$.
5. Set $Y$ to be equal to the first $Len$ octets of $Y$.
6. Output $Y$.
```

The second KDF/MGF specified will be referred to as *Mech*2. It runs as follows.

Description:

```
Input:   An input octet string $X$ of any length
         An output length $Len$
Output:  An octet string $Y$ of length $Len$

1. If $Len > 2^{32} HashLen$ then output "error" and abort.
2. Set $k := \lceil Len/HashLen \rceil$.
3. Set $Y$ to be the empty string.
4. For $i$ from 1 to $k$ do
     a) $Y := Y || Hash(X || \mathsf{I2OSP}(i, 4))$.
5. Set $Y$ to be equal to the first $Len$ octets of $Y$.
6. Output $Y$.
```

Part B

Block Ciphers

1. MISTY1

1.1 Introduction

1.1.1 Overview

MISTY1 was first published in 1996 [424, 425] and uses 128-bit keys. It was submitted to NESSIE by Eisaku Takeda on behalf of Mitsubishi. MISTY1 can be implemented in situations where resources are heavily constrained, and the constituent lookup tables are optimised for hardware performance. The entire algorithm is built from recursive components such that at each level the structure is again a secure Feistel-like structure. There exists a variant of MISTY1, namely MISTY2, which was published around the same time [424, 425] and also uses 128-bit keys. Another variant of MISTY1, namely KASUMI, has been chosen for the 3GPP standard [1].

1.1.2 Outline of the primitive

MISTY1 is an iterated cipher that operates over 8 rounds or, more generally, a multiple of 4 rounds. It accepts 64-bit plaintext and 128-bit key. It uses two S-boxes, a 7×7 S-box, S7, and a 9×9 S-box, S9. These are incorporated into the lowest level of a recursively constructed Feistel-like structure. They are designed to obtain good resistance to linear and differential attacks but are also designed so that they can be implemented using relatively few logical components which implement boolean functions of relatively low degree, with S7 comprising a set of cubic functions, and S9 comprising a set of quadratic functions. Each successively higher level of the cipher is built from instances of the previous level which are linked together in a Feistel-like manner. The key-schedule of MISTY1 is relatively simple compared to other block ciphers and uses multiple instances of the lowest-level module (which include S7 and S9 S-boxes) to generate a further 128-bit sub-key from the original 128-bit input key. These $128 + 128 = 256$ key schedule bits are then used as round keys for key inputs to the Feistel-like modules. MISTY1 also uses so-called keyed-FL layers after every second round so as to introduce irregularity into the cipher round function and thereby enhance security at little extra implementation cost. Decryption for MISTY1 is very similar to encryption. The only changes are that each FL layer must be replaced by its inverse, and secondly, the order in which the round keys are introduced is reversed.

1.1.3 Security and performance

MISTY1 [426] has been widely studied for five years and for the duration of the NESSIE project and no serious security flaws have been found. The cipher and its key-schedule perform satisfactorily on all software and hardware platforms, and requires relatively simple logic to implement its 7×7 and 9×9 S-boxes. Many attacks on MISTY1 may also be relevant to the similar ciphers, MISTY2, and KASUMI, and vice versa. It appears to be easier to attack MISTY1 without FL functions. Conversely, this implies that the insertion of the FL functions in MISTY1 enhances its security. Attacks on MISTY1 without the FL operations have been accomplished up to five rounds. The low algebraic degree of the constituent functions of the MISTY1 S-boxes has invited higher order differential attacks by Lai [385], Knudsen [352], and Tanaka *et al.* [604] on MISTY1 without FL functions. The Slide attack has been proposed against MISTY1 by Biryukov and Wagner [90] where the same subkey is applied to every nth round. MISTY1 is designed to have provable security against differential and linear cryptanalysis, and this proof is achieved by bounding the average differential/linear probabilities for the recursive layers of MISTY1; if the average differential/linear probability of each layer is p then the complete cipher has probability upper-bounded by p^4. It is claimed by the designers that the unequal division of the S-boxes into 7 bits and 9 bits has an advantage against differential and linear cryptanalysis, as the probability bound can be made lower for S-boxes that use odd as opposed to even numbers of bits. Recently, Integral Cryptanalysis [371, 448] has been applied to MISTY1 including FL functions. The integral attacks are over 4 and 5 rounds and exploit the Sakurai-Zheng property that was initially applied to MISTY2. Also a new attack, the Slicing Attack by Kuhn [380, 381] has been applied to the 4-round version of MISTY1, making use of the special structure and position of the key-dependent linear FL functions.

1.2 Description

MISTY1 is a Feistel network based on a 32-bit nonlinear function. It takes 64-bit plaintext and a 128-bit key, and is recommended for 8 rounds (more generally a multiple of 4 rounds). We will now describe each module.

1.2.1 Encryption

The encryption operation is as shown in Fig 46. The 64-bit plaintext input, P, is split into two halves, L_0 and R_0, each of 32-bits. Before the first round and after every two rounds, both left and right halves, L_{i-1} and R_{i-1}, respectively, are passed through keyed FL modules, which take a 32-bit input, a 32-bit subkey, KL_i, and produce a 32-bit output. During each round the left half is input to a keyed FO module which accepts a 32-bit input, a 48-bit sub-key, KI_i, a 64-bit sub-key, KO_i, and produces a 32-bit output. This output is then XOR'ed with the right half to produce a modified right half (XOR is short for bitwise eXclusive OR). At the end of each round, the left and right halves are swapped to give L_i and R_i, respectively. After 8 rounds (or, more generally, some multiple of 4 rounds) there is a final pass through the keyed FL modules followed by a final swap of left and right halves, before a final recombination of left and right halves, L_{n+1} and R_{n+1} to produce an output ciphertext of length 64 bits, C, where n is the number of rounds, (normally 8). The final FL operation after the last round is to ensure that decryption is like encryption apart from a reverse of the subkey order and the interchange of FL and FL^{-1}.

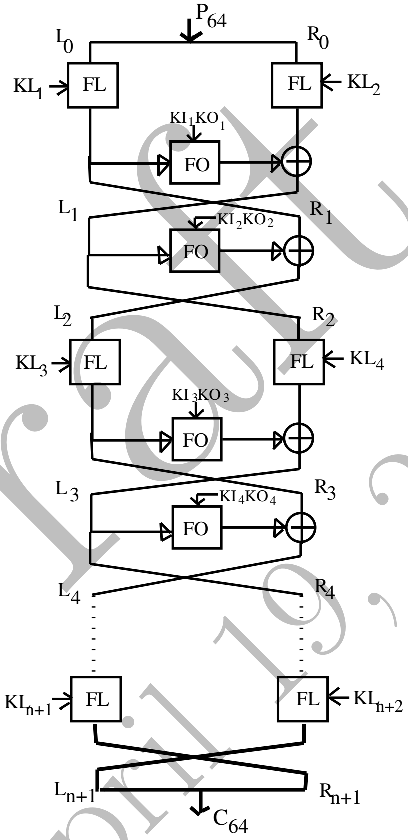


Fig. 46. Encryption for MISTY1

We can summarise the encryption inputs as,

$$\begin{aligned} P &= L_0 \| R_0 \\ KL &= \{KL_i\} \\ KO &= \{KO_i\} \\ KI &= \{KI_i\} \end{aligned}$$

The encryption function is then defined as,

```
for $i = 1, 3, \ldots, n-1$ :
    $R_i = FL(L_{i-1}, KL_i)$
    $L_i = FL(R_{i-1}, KL_{i+1}) \oplus FO(R_i, KO_i, KI_i)$
    $L_{i+1} = R_i \oplus FO(L_i, KO_{i+2}, KI_{i+1})$
    $R_{i+1} = L_i$
for $i = n+1$ :
    $R_i = FL(L_{i-1}, KL_i)$
    $L_i = FL(R_{i-1}, KL_{i+1})$
```

where the output is $C = L_{n+1} \| R_{n+1}$.

The above variables are of the following bit-widths:

P	: 64 bits
L_i	: 32 bits
R_i	: 32 bits
KL_i	: 32 bits
KI_i	: 48 bits
KO_i	: 64 bits
C	: 64 bits

Note that the order (and position) in which the sub-keys are input is as follows (shown for $n = 8$ rounds):

KL_1	KI_1, KO_1	KL_2
	KI_2, KO_2	
KL_3	KI_3, KO_3	KL_4
	KI_4, KO_4	
KL_5	KI_5, KO_5	KL_6
	KI_6, KO_6	
KL_7	KI_7, KO_7	KL_8
	KI_8, KO_8	
KL_9		KL_{10}

1.2.2 Decryption

The decryption operation is as shown in Fig 47, and is identical in operation to encryption apart from the following two modifications.

- All FL modules are replaced by their inverse modules, FL^{-1}.
- The order in which the sub-keys are applied is reversed. To be more explicit, for decryption the sub-keys are input as follows (shown for $n = 8$ rounds):

KL_{10}	KI_8, KO_8	KL_9
	KI_7, KO_7	
KL_8	KI_6, KO_6	KL_7
	KI_5, KO_5	
KL_6	KI_4, KO_4	KL_5
	KI_3, KO_3	
KL_4	KI_2, KO_2	KL_3
	KI_1, KO_1	
KL_2		KL_1

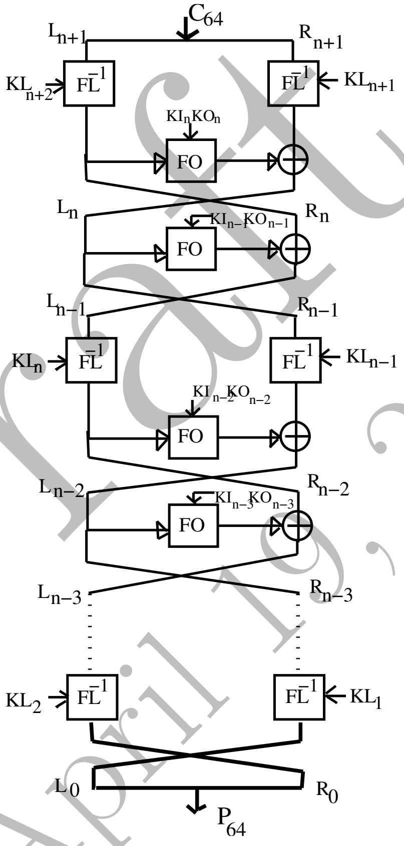


Fig. 47. Decryption for MISTY1

1.2.3 FL Module

The FL module is used in encryption only and is shown in Fig. 48. It takes a 32-bit input, X_{32}, and splits this into two 16-bit halves, X_L and X_R. The right half, X_R, is modified by XOR'ing with a modified version of the left half after the left half, X_L, has been bitwise AND'ed with a 16-bit key, KL_{iL} (AND is shown as $\wedge$ in Fig. 48). Following this, the left half is modified by XOR'ing with a modified version of the right half after the right half has been bitwise OR'ed with a 16-bit key, KL_{iR} (OR is shown as $\vee$ in Fig. 48). Finally the 16-bit left and 16-bit right halves, Y_L and Y_R, are concatenated to form a 32-bit output, Y_{32}.

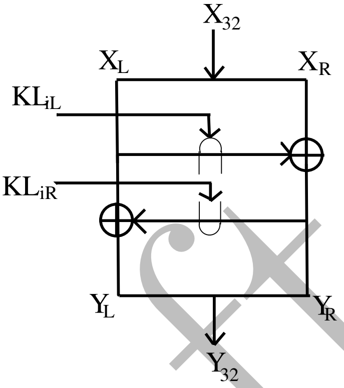


Fig. 48. FL function for MISTY1

We can summarise the FL inputs as,

$$
\begin{aligned}
X_{32} &= X_L \| X_R \\
KL_i &= KL_{iL} \| KL_{iR}
\end{aligned}
$$

The FL function is then defined as,

$$
\begin{aligned}
Y_R &= (X_L \wedge KL_{iL}) \oplus X_R \\
Y_L &= (Y_R \vee KL_{iR}) \oplus X_L
\end{aligned}
$$

where the output is $Y_{32} = Y_L \| Y_R$.

The above variables are of the following bit-widths:

X_{32}	: 32 bits
X_L	: 16 bits
X_R	: 16 bits
KL_i	: 32 bits
KL_{iL}	: 16 bits
KL_{iR}	: 16 bits
Y_L	: 16 bits
Y_R	: 16 bits
Y_{32}	: 32 bits

1.2.4 FL^{-1} Module

The FL^{-1} module is the inverse to the FL module, is used in decryption only, and is shown in Fig. 49. It takes a 32-bit input, Y_{32}, and splits this into two 16-bit halves, Y_L and Y_R. The left half, Y_L, is modified by XOR'ing with a modified version of the right half after the right half, Y_R, has been bitwise OR'ed with a 16-bit key, KL_{iR} (OR is shown as $\vee$ in Fig. 49). Following this, the right half is modified by XOR'ing with a modified version of the left half where the left half has been bitwise AND'ed with a 16-bit key, KL_{iL} (AND is shown as $\wedge$ in Fig. 49). Finally the 16-bit left and 16-bit right halves, X_L and X_R, are concatenated to form a 32-bit output, X_{32}.

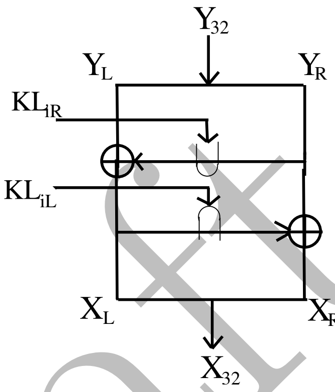


Fig. 49. FL^{-1} function for MISTY1

We can summarise the FL^{-1} inputs as,

$$\begin{aligned} Y_{32} &= Y_L \| Y_R \\ KL_i &= KL_{iL} \| KL_{iR} \end{aligned}$$

The FL^{-1} function is then defined as,

$$\begin{aligned} X_L &= (Y_R \vee KL_{iR}) \oplus Y_L \\ X_R &= (X_L \wedge KL_{iL}) \oplus Y_R \end{aligned}$$

where the output is $X_{32} = X_L \| X_R$.

The above variables are of the following bit-widths:

Y_{32}	: 32 bits
Y_L	: 16 bits
Y_R	: 16 bits
KL_i	: 32 bits
KL_{iL}	: 16 bits
KL_{iR}	: 16 bits
X_L	: 16 bits
X_R	: 16 bits
X_{32}	: 32 bits

1.2.5 FO Module

The FO module is used for encryption and decryption and is shown in Fig. 50. It takes a 32-bit input, X_{32}, and splits this into two 16-bit halves, L_0 and R_0. The left half, L_0, is XOR'ed with a 16-bit sub-key, KO_{i1}, and then input to the FI module, keyed by KI_{i1}. The 16-bit output of this FI module is then XOR'ed with R_0 before the left and right sides are swapped. The above process is repeated two more times, before a final XOR of the left-hand side, L_3, with KO_{i4}. Finally the output, Y_{32}, is obtained by concatenating left and right-hand sides, L_3 and R_3.

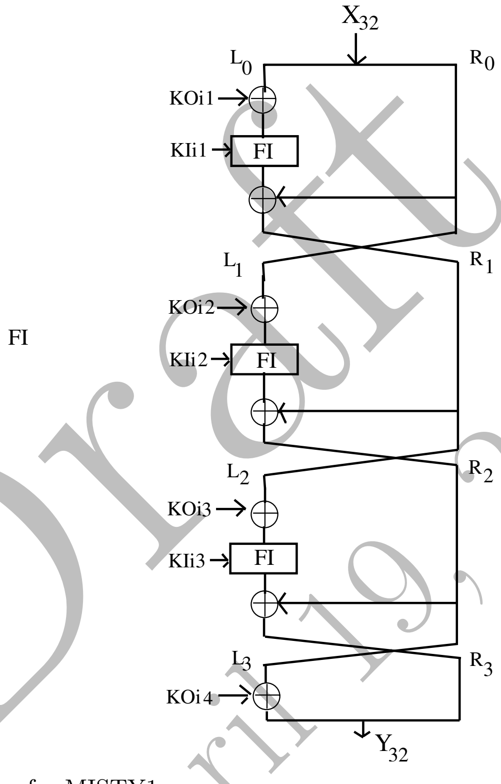


Fig. 50. FO function for MISTY1

We can summarise the FO inputs as,

$$X_{32} = L_0 \| R_0$$
$$KO_i = KO_{i1} \| KO_{i2} \| KO_{i3} \| KO_{i4}$$
$$KI_i = KI_{i1} \| KI_{i2} \| KI_{i3}$$

The FO function is then defined as,

```
for j = 1 to 3 do :
    $R_j = FI(L_{j-1} \oplus KO_{ij}, KI_{ij}) \oplus R_{j-1}$
    $L_j = R_{j-1}$
$Y_{32} = (L_3 \oplus KO_{i4}) \| R_3$
```

where Y_{32} is the output.

The above variables are of the following bit-widths:

X_{32}	: 32 bits
L_j	: 16 bits
R_j	: 16 bits
KO_i	: 64 bits
KO_{ij}	: 16 bits
KI_i	: 48 bits
KI_{ij}	: 16 bits
Y_{32}	: 32 bits

1.2.6 FI Module

The FI module is used for encryption, decryption and the key schedule, and is shown in Fig. 51. It takes a 16-bit input, X_{16}, and splits this into two unequal-length parts, L_0 and R_0, of length 9 bits and 7 bits, respectively. The left half, L_0, is then input to the 9×9 S-box, S9, before being XOR'ed with a modified R_0, where R_0 has been *zero-extended* from 7 bits to 9 bits by the concatenation of two bits on the left side. The left and right sides are then swapped. The 7-bit left half, L_1, is then input to the 7×7 S-box, S7, before being XOR'ed with a modified R_1, where R_1 has been *truncated* from 9 bits to 7 bits by dropping two bits on the left side. Then the 7-bit left-hand side is XOR'ed with a sub-key, KI_{ijL}, and the 9-bit right-hand side is XOR'ed with a sub-key, KI_{ijR}. The left and right sides are then swapped. The 9-bit left half, L_2, is then input to the 9×9 S-box, S9, before being XOR'ed with a modified R_2, where R_2 has been *zero-extended* from 7 bits to 9 bits by the concatenation of two bits on the left side. Finally, the left and right sides are then swapped before a final concatenation, $L_3 \| R_3$, to produce the output Y_{16}.

We can summarise the FI inputs as,

$$\begin{aligned} X_{16} &= L_0 \| R_0 \\ KI_{ij} &= KI_{ijL} \| KI_{ijR} \end{aligned}$$

The FI function is then defined as,

$$\begin{aligned}
R_1 &= S9[L_0] \oplus (\mathtt{00_b} \| R_0) \\
L_1 &= R_0 \\
R_2 &= S7[L_1] \oplus (\text{truncate}(R_1)) \oplus KI_{ijL} \\
L_2 &= R_1 \oplus KI_{ijR} \\
R_3 &= S9[L_2] \oplus (\mathtt{00_b} \| R_2) \\
L_3 &= R_2 \\
Y_{16} &= L_3 \| R_3
\end{aligned}$$

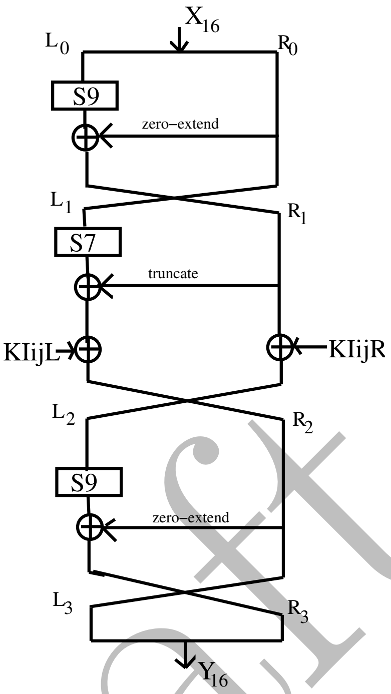


Fig. 51. FI function for MISTY1

where Y_{16} is the output.

The above variables are of the following bit-widths:

X_{16}	: 16 bits
L_0	: 9 bits
R_0	: 7 bits
L_1	: 7 bits
R_1	: 9 bits
L_2	: 9 bits
R_2	: 7 bits
L_3	: 7 bits
R_3	: 9 bits
KI_{ij}	: 16 bits
KI_{ijL}	: 7 bits
KI_{ijR}	: 9 bits
Y_{16}	: 16 bits

1.2.7 Key Schedule

The Key Schedule is used before encryption or decryption to prepare an extra key material, K', from the input key, K, where K and K' are both of length 128 bits. The Key Schedule is shown in Fig. 52. It comprises 8 consecutive applications of the FI module. Firstly, K is split up into eight parts, K_i, $1 \leq i \leq 8$, each of length 16 bits. Then K_i is considered as the input to FI with K_{i+1} acting as the key to the FI module. The 16-bit output from each FI module is K'_i, $1 \leq i \leq 8$. Note that K_9 is interpreted as K_1.

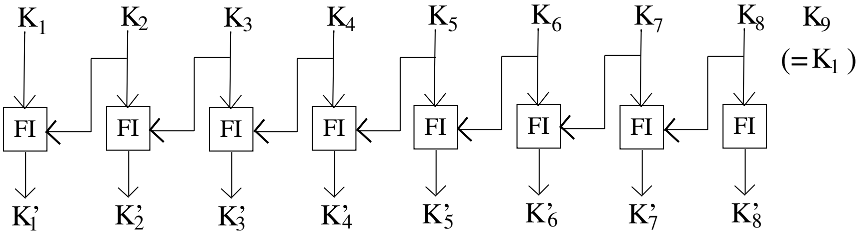


Fig. 52. Key Schedule for MISTY1

We can summarise the Key Schedule inputs as,

$$K = K_1 \| K_2 \| K_3 \| K_4 \| K_5 \| K_6 \| K_7 \| K_8$$

The Key Schedule function is then defined as,

```
for $i = 1$ to 8 do
    $K'_i = FI(K_i, K_{i+1})$
$K' = K'_1 \| K'_2 \| K'_3 \| K'_4 \| K'_5 \| K'_6 \| K'_7 \| K'_8$
```

where $K_9 = K_1$.

The above variables are of the following bit-widths:

K	: 128 bits
K_i	: 16 bits
K'	: 128 bits
K'_i	: 16 bits

The sub-keys, $KL_{iL/R}$, KO_{ij} and KI_{ij}, with $i = 1 \ldots 10$, $i = 1 \ldots 8$ and $j = 1 \ldots 4$, $i = 1 \ldots 8$ and $j = 1 \ldots 3$, respectively, are then derived from the K and K' keys by the assignment shown in Table 52. These sub-keys are 16 bits long.

Table 52. Subkey Mapping Table

Round	KO_{i1}	KO_{i2}	KO_{i3}	KO_{i4}	KI_{i1}	KI_{i2}	KI_{i3}	KL_{iL}	KL_{iR}
Actual	K_i	K_{i+2}	K_{i+7}	K_{i+4}	K'_{i+5}	K'_{i+1}	K'_{i+3}	$K_{\frac{i+1}{2}}$ odd i $K'_{\frac{i}{2}+2}$ even i	$K'_{\frac{i+1}{2}+6}$ odd i $K_{\frac{i}{2}+4}$ even i

The indices in Table 52 should be between 1 and 8 where one should subtract a multiple of 8 if necessary.

1.2.8 S-boxes

We here give a logical description of the two S-boxes of MISTY1, S7 and S9. The variables $x_1, x_2, \ldots$ are the binary inputs that form the 7 or 9-bit inputs to the two S-boxes, respectively, and the variables $y_1, y_2 \ldots$ are the corresponding binary outputs from the two S-boxes. The subsequent section gives pseudo-code for MISTY1 and also includes an alternative description of the two S-boxes as permutations over the integers $\{0, 1, \ldots, 127\}$ (S7) and $\{0, 1, \ldots, 511\}$ (S9). This alternative description is particularly useful for software implementations where the S-boxes are typically implemented as lookup tables.

Description of S7

$$\begin{aligned}
y_7 = \quad & x_1x_2x_5 \oplus x_1x_4x_7 \oplus x_1x_5x_6 \oplus x_2x_4x_6 \oplus x_3x_4x_5 \oplus x_1x_3 \oplus x_1x_6 \oplus x_2x_4 \oplus x_2x_5 \\
& \oplus x_2x_7 \oplus x_4x_7 \oplus x_6x_7 \oplus x_4 \\
y_6 = \quad & x_1x_2x_5 \oplus x_2x_6x_7 \oplus x_3x_5x_7 \oplus x_4x_5x_6 \oplus x_5x_6x_7 \oplus x_1x_7 \oplus x_2x_4 \oplus x_2x_7 \oplus x_3x_6 \\
& \oplus x_4x_7 \oplus x_5 \oplus x_6 \oplus x_7 \\
y_5 = \quad & x_1x_2x_3 \oplus x_1x_2x_6 \oplus x_2x_4x_7 \oplus x_2x_5x_6 \oplus x_3x_4x_6 \oplus x_1x_6 \oplus x_2x_5 \oplus x_3x_7 \oplus x_4x_5 \oplus x_2 \oplus 1 \\
y_4 = \quad & x_1x_3x_7 \oplus x_1x_4x_6 \oplus x_2x_3x_6 \oplus x_5x_6x_7 \oplus x_1x_2 \oplus x_1x_5 \oplus x_3x_5 \oplus x_4x_7 \oplus x_6 \oplus x_7 \oplus 1 \\
y_3 = \quad & x_1x_3x_5 \oplus x_1x_4x_7 \oplus x_2x_3x_4 \oplus x_2x_3x_7 \oplus x_3x_6x_7 \oplus x_4x_5x_7 \oplus x_1x_3 \oplus x_1x_4 \oplus x_1x_6 \\
& \oplus x_2x_7 \oplus x_3x_6 \oplus x_5x_6 \oplus x_3 \\
y_2 = \quad & x_1x_2x_7 \oplus x_1x_3x_6 \oplus x_1x_4x_5 \oplus x_2x_3x_5 \oplus x_1x_4 \oplus x_1x_7 \oplus x_2x_6 \oplus x_3x_4 \oplus x_3x_7 \oplus x_5x_7 \\
& \oplus x_1 \oplus 1 \\
y_1 = \quad & x_1x_2x_4 \oplus x_1x_2x_7 \oplus x_1x_6x_7 \oplus x_2x_5x_7 \oplus x_3x_4x_7 \oplus x_1x_5 \oplus x_2x_3 \oplus x_2x_6 \oplus x_4x_6 \oplus x_7 \oplus 1
\end{aligned}$$

Description of S9

$$\begin{aligned}
y_9 = \quad & x_1x_5 \oplus x_1x_6 \oplus x_2x_6 \oplus x_2x_7 \oplus x_3x_7 \oplus x_3x_8 \oplus x_4x_8 \oplus x_4x_9 \oplus x_5x_9 \oplus 1 \\
y_8 = \quad & x_1x_4 \oplus x_1x_6 \oplus x_1x_9 \oplus x_3x_7 \oplus x_3x_9 \oplus x_4x_5 \oplus x_5x_6 \oplus x_6x_7 \oplus x_6x_8 \oplus x_7x_9 \oplus x_2 \oplus x_6 \oplus 1 \\
y_7 = \quad & x_2x_6 \oplus x_2x_8 \oplus x_3x_4 \oplus x_3x_9 \oplus x_4x_5 \oplus x_5x_6 \oplus x_5x_7 \oplus x_5x_9 \oplus x_6x_8 \oplus x_8x_9 \oplus x_1 \oplus x_5 \\
y_6 = \quad & x_1x_5 \oplus x_1x_7 \oplus x_2x_3 \oplus x_2x_8 \oplus x_3x_4 \oplus x_4x_5 \oplus x_4x_6 \oplus x_4x_8 \oplus x_5x_7 \oplus x_7x_8 \oplus x_4 \oplus x_9 \\
y_5 = \quad & x_1x_2 \oplus x_1x_7 \oplus x_2x_3 \oplus x_3x_4 \oplus x_3x_5 \oplus x_3x_7 \oplus x_4x_6 \oplus x_4x_9 \oplus x_6x_7 \oplus x_6x_9 \oplus x_3 \oplus x_8 \\
y_4 = \quad & x_1x_2 \oplus x_1x_9 \oplus x_2x_3 \oplus x_2x_4 \oplus x_2x_6 \oplus x_3x_5 \oplus x_3x_8 \oplus x_5x_6 \oplus x_5x_8 \oplus x_6x_9 \oplus x_2 \oplus x_7 \\
y_3 = \quad & x_1x_2 \oplus x_1x_3 \oplus x_1x_5 \oplus x_1x_9 \oplus x_2x_4 \oplus x_2x_7 \oplus x_4x_5 \oplus x_4x_7 \oplus x_5x_8 \oplus x_8x_9 \oplus x_1 \oplus x_6 \oplus 1 \\
y_2 = \quad & x_1x_8 \oplus x_2x_3 \oplus x_2x_5 \oplus x_2x_9 \oplus x_3x_6 \oplus x_3x_8 \oplus x_5x_9 \oplus x_6x_7 \oplus x_7x_8 \oplus x_8x_9 \oplus x_4 \oplus x_8 \oplus 1 \\
y_1 = \quad & x_1x_3 \oplus x_1x_6 \oplus x_1x_9 \oplus x_2x_9 \oplus x_3x_4 \oplus x_3x_6 \oplus x_4x_7 \oplus x_4x_9 \oplus x_7x_8 \oplus x_8x_9 \oplus x_5 \oplus x_9 \oplus 1
\end{aligned}$$

1.2.9 Pseudo-Code

```
MAIN    /* an example encryption then decryption, both over 8 rounds */
{
    K = 0x00112233445566778899aabbccddeeff;
    plaintext = 0x0123456789abcdef;
    K' = MISTY1KEYSCHEDULE(K); /* = 0xcf518e7f5e29673acdbc07d6bf355e11 */
    ciphertext <- MISTY1(plaintext,K,K',8,0); /* = 0x8b1da5f56ab3d07c */
    plaintext <- MISTY1(ciphertext,K,K',8,1); /* = 0x0123456789abcdef */
} /* MAIN */

64Bit <- MISTY1(text,K,K',n,f)
    /* f=0 to encrypt, f=1 to decrypt */
{
    left = plaintext{1,32};
    right = plaintext{33,64};
    firstround = 1;
    lastround = n + 1;
    if (f is equal to 1)
    {
        SWAP(firstround,lastround);
        step = -1;
    }
    else
        step = 1;
    FLlayer = 1;
    i = firstround;
    while (i is not equal to (lastround + step))
    {
        roundindex = i;
        if (FLlayer == 1)
        {
            KL = SUBKEYL(K,K',roundindex,f,0);
            left = FL(left,KL,f);
            KL = SUBKEYL(K,K',roundindex,f,1);
            right = FL(right,KL,f);
            FLlayer = 0;
        }
        else
            FLlayer = 1;
        if (f is equal to 1)
            roundindex = roundindex - 1;
        if (i is not equal to lastround)
        {
            KI = SUBKEYI(K',roundindex);
            KO = SUBKEYO(K,roundindex);
            leftmod = FO(left,KO,KI);
            right = XOR(right,leftmod);
        }
        SWAP(left,right);
        i = i + step;
    }
    RETURN(left || right);
} /* MISTY1 */

32Bit <- SUBKEYL(K,K',i,f,s)
{
    k = (8 * i) - 7;
    if ((f XOR s) == 0)
        RETURN(K{k,k + 15} || K'{k + 96,k + 111});
    else
        RETURN(K'{k + 32,k + 47} || K{k + 64,k + 79});
} /* SUBKEYL */

48Bit <- SUBKEYI(K',i)
{
    k = (16 * i) - 15;
    RETURN(K'{k + 80,k + 95} || K'{k + 16,k + 31} || K'{k + 48,k + 63});
} /* SUBKEYI */

64Bit <- SUBKEYO(K,i)
{
    k = (16 * i) - 15;
    KO = K{k,k + 15} || K{k + 32,k + 47};
    RETURN(KO || K{k + 112,k + 127} || K{k + 64,k + 95});
} /* SUBKEYO */

32Bit <- FL(X32,KL,f)   /* f = 0 for FL, f = 1 for FL inverse */
{
    Xl = X32{1,16};
    Xr = X32{17,32};
    keyl = KL{1,16};
    keyr = KL{17,32};
    if (f == 0)
    {
        Xl’ = AND(Xl,keyl);
        Xr = XOR(Xr,Xl’);
        Xr’ = OR(Xr,keyr);
        Xl = XOR(Xl,Xr’);
    }
    else
    {
        Xr’ = OR(Xr,keyr);
        Xl = XOR(Xl,Xr’);
        Xl’ = AND(Xl,keyl);
        Xr = XOR(Xr,Xl’);
    }
    RETURN(Xl || Xr);
} /* FL */

32Bit <- FO(X32,KO,KI)
{
    l16 = X32{1,16};
    r16 = X32{17,32};
    for r = 1 to 3 do
    {
        k = (16 * r) - 15;
        KO16 = KO{k,k + 15};
        KI16 = KI{k,k + 15};
        l16 = XOR(l16,KO16);
        l16 = FI(l16,KI16);
        l16 = XOR(l16,r16);
        SWAP(l16,r16);
    }
    KO16 = KO{49,64};
    l16 = XOR(l16,KO16);
    RETURN(l16 || r16);
} /* FO */

16Bit <- FI(X16,KI16)
{
    l9 = X16{1,9};
    r7 = X16{10,16};
    key7 = KI16{1,7};
    key9 = KI16{8,16};
    l9 = S9[l9];
    l9 = XOR(l9,00 || r7);
    r7 = S7[r7];
    r7 = XOR(r7,TRUNC(l9));
    r7 = XOR(r7,key7);
    l9 = XOR(l9,key9);
    l9 = S9[l9];
    l9 = XOR(l9,00 || r7);
    RETURN(r7 || l9);
} /* FI */

128Bit <- MISTY1KEYSCHEDULE(K)
{
    K’ = ();
    for i = 1 to 8 do
    {
        k = (16 * i) - 15;
        K16 = FI(K{k,k + 15},K{k + 16,k + 31});
        K’ = K’ || K16;
    }
    RETURN(K’);
} /* MISTY1KEYSCHEDULE */

/* S7 is here specified in decimal */
S7 = [27, 50, 51, 90, 59, 16, 23, 84, 91, 26,114,115,107, 44,102, 73,
      31, 36, 19,108, 55, 46, 63, 74, 93, 15, 64, 86, 37, 81, 28,  4,
      11, 70, 32, 13,123, 53, 68, 66, 43, 30, 65, 20, 75,121, 21,111,
      14, 85,  9, 54,116, 12,103, 83, 40, 10,126, 56,  2,  7, 96, 41,
      25, 18,101, 47, 48, 57,  8,104, 95,120, 42, 76,100, 69,117, 61,
      89, 72,  3, 87,124, 79, 98, 60, 29, 33, 94, 39,106,112, 77, 58,
       1,109,110, 99, 24,119, 35,  5, 38,118,  0, 49, 45,122,127, 97,
      80, 34, 17,  6, 71, 22, 82, 78,113, 62,105, 67, 52, 92, 88,125]

/* S9 is here specified in decimal */
S9 = [451,203,339,415,483,233,251, 53,385,185,279,491,307,  9, 45,211,
      199,330, 55,126,235,356,403,472,163,286, 85, 44, 29,418,355,280,
      331,338,466, 15, 43, 48,314,229,273,312,398, 99,227,200,500, 27,
        1,157,248,416,365,499, 28,326,125,209,130,490,387,301,244,414,
      467,221,482,296,480,236, 89,145, 17,303, 38,220,176,396,271,503,
      231,364,182,249,216,337,257,332,259,184,340,299,430, 23,113, 12,
       71, 88,127,420,308,297,132,349,413,434,419, 72,124, 81,458, 35,
      317,423,357, 59, 66,218,402,206,193,107,159,497,300,388,250,406,
      481,361,381, 49,384,266,148,474,390,318,284, 96,373,463,103,281,
      101,104,153,336,  8,  7,380,183, 36, 25,222,295,219,228,425, 82,
      265,144,412,449, 40,435,309,362,374,223,485,392,197,366,478,433,
      195,479, 54,238,494,240,147, 73,154,438,105,129,293, 11, 94,180,
      329,455,372, 62,315,439,142,454,174, 16,149,495, 78,242,509,133,
      253,246,160,367,131,138,342,155,316,263,359,152,464,489,  3,510,
      189,290,137,210,399, 18, 51,106,322,237,368,283,226,335,344,305,
      327, 93,275,461,121,353,421,377,158,436,204, 34,306, 26,232,  4,
      391,493,407, 57,447,471, 39,395,198,156,208,334,108, 52,498,110,
      202, 37,186,401,254, 19,262, 47,429,370,475,192,267,470,245,492,
      269,118,276,427,117,268,484,345, 84,287, 75,196,446,247, 41,164,
       14,496,119, 77,378,134,139,179,369,191,270,260,151,347,352,360,
      215,187,102,462,252,146,453,111, 22, 74,161,313,175,241,400, 10,
      426,323,379, 86,397,358,212,507,333,404,410,135,504,291,167,440,
      321, 60,505,320, 42,341,282,417,408,213,294,431, 97,302,343,476,
      114,394,170,150,277,239, 69,123,141,325, 83, 95,376,178, 46, 32,
      469, 63,457,487,428, 68, 56, 20,177,363,171,181, 90,386,456,468,
       24,375,100,207,109,256,409,304,346,  5,288,443,445,224, 79,214,
      319,452,298, 21,  6,255,411,166, 67,136, 80,351,488,289,115,382,
      188,194,201,371,393,501,116,460,486,424,405, 31, 65, 13,442, 50,
       61,465,128,168, 87,441,354,328,217,261, 98,122, 33,511,274,264,
      448,169,285,432,422,205,243, 92,258, 91,473,324,502,173,165, 58,
      459,310,383, 70,225, 30,477,230,311,506,389,140,143, 64,437,190,
      120,  0,172,272,350,292,  2,444,162,234,112,508,278,348, 76,450]
```

```
Data, Logical and Arithmetic Definitions
=========================================

            0x1fa4....
A number of this type, with 0x at the beginning, should be interpreted as
hexadecimal. The above example represents the bit string 0001111110100100.

            a || b || c ||..
outputs the concatenation of the bitstrings, a,b,c,.... with a leftmost.
Example:  a = 1011, b = 0110 -> a || b = 10110110

            XOR(a,b)
outputs the bitwise XOR of a and b.
Example:  a = 1011, b = 0110 -> XOR(a,b) = 1101

            OR(a,b)
outputs the bitwise OR of a and b.
Example:  a = 1011, b = 0110 -> OR(a,b) = 1111

            AND(a,b)
outputs the bitwise AND of a and b.
Example:  a = 1011, b = 0110 -> AND(a,b) = 0010

            00 || a
outputs the concatenation of two zero bits onto the left side of a.
Example:  a = 1011 -> 00 || a = 001011

            TRUNC(a)
outputs the truncation of a by the removal of the two leftmost bits of a.
Example:  a = 1011 -> TRUNC(a) = 11

            SWAP(a,b)
swaps the contents of a and b.
Example:  a = 1011, b = 0110 -> SWAP(a,b) -> a = 0110, b = 1011

            A{a,b}
outputs the bit-segment of the bit string, A, starting from bit a of A
and up to and including bit b of A. Thus, A{a,b} has bit-length b - a + 1.
The first (leftmost) bit in A is A{1}.
Example:  A = 10110110, -> A{4,6} = 101

            A[a]
outputs the integer element stored at position a in the integer array, A.
The first integer in A is A[0].
```

```
Example:  A = [125,32,7,89,36,101,....], -> A[4] = 36

             '*'  '+'
mean integer multiplication and addition, respectively.

Note that, if k > 128, then k is replaced by k - (a * 128), where
a is an integer chosen so that
             0 < k - (a * 128) < 129

List of Variables Used
=======================
128 bits            - K,K'
 64 bits            - KO,ciphertext,plaintext
 48 bits            - KI
 32 bits            - KL,leftmod,X32,left,right
 16 bits            - keyl,keyr,Xl,Xr,Xl',Xr',l16,r16,KO16,KI16,X16,K16
  9 bits            - key9,l9
  7 bits            - key7,r7
 integer            - n,firstround,lastround,i,roundindex,r
{1,2,....,127,128}  - k
  binary            - f,FLlayer,s
```

1.3 Test vectors

To aid in verification of a software or hardware implementation of MISTY1, we here provide a few test vectors comprising, as input (K, plaintext) and, as output from the key schedule and encryption (K', ciphertext).

K	00112233445566778899AABBCCDDEEFF$_\mathrm{x}$	0123456789ABCDEF$_\mathrm{x}$	plain
K'	CF518E7F5E29673ACDBC07D6BF355E11$_\mathrm{x}$	8B1DA5F56AB3D07C$_\mathrm{x}$	cipher
K	414AFD99BB577EE69DF58CC8FB4E6888$_\mathrm{x}$	9FC302E281310E90$_\mathrm{x}$	plain
K'	C7BD6E012268237A4389305A1B360B8C$_\mathrm{x}$	15C270974B9B9163$_\mathrm{x}$	cipher
K	3C54AED9A5389C947167DB9D97C6967A$_\mathrm{x}$	032C4A4A100EE807$_\mathrm{x}$	plain
K'	7C8E13EBFE7648050C9097934205662B$_\mathrm{x}$	3346CB8C779CF2DE$_\mathrm{x}$	cipher
K	D3F11A6D25F1B3866FDADA0B5E53FA17$_\mathrm{x}$	DB9E3218402023F3$_\mathrm{x}$	plain
K'	F011D035AC920F832F69BCF7B860D4F0$_\mathrm{x}$	B2DD1595A450BC98$_\mathrm{x}$	cipher
K	5F87F88EC7641D83AF03FD8327821046$_\mathrm{x}$	6553DE24C0DD900B$_\mathrm{x}$	plain
K'	3736172D7421C91401596DB29D3D5536$_\mathrm{x}$	60081E65CB7C2B84$_\mathrm{x}$	cipher

It is also useful to verify the test vectors for incomplete implementations of MISTY1 as, for instance, when the FL/FL^{-1} module is removed and replaced by an identity transformation, (input directly connected to output). One can similarly replace the FO and/or FI modules with appropriate identity transformations. In the following we provide one set of test vectors for each of the situations where either FL/FL^{-1} and/or FO and/or FI transformations are replaced by identity transformations. Note that, in all possible incomplete cipher configurations considered below, the action of decryption of the ciphertext results in the original plaintext.

Input:	$K = \texttt{5F87F88EC7641D83AF03FD8327821046}_{\texttt{x}}$ plaintext $= \texttt{6553DE24C0DD900B}_{\texttt{x}}$
Implementation includes	K', ciphertext
–	$K' = \texttt{5F87F88EC7641D83AF03FD8327821046}_{\texttt{x}}$ ciphertext $= \texttt{A58E4E2FC0DD900B}_{\texttt{x}}$
FI	$K' = \texttt{3736172D7421C91401596DB29D3D5536}_{\texttt{x}}$ ciphertext $= \texttt{A58E4E2FC0DD900B}_{\texttt{x}}$
FO	$K' = \texttt{5F87F88EC7641D83AF03FD8327821046}_{\texttt{x}}$ ciphertext $= \texttt{87E7E92682D978C3}_{\texttt{x}}$
FO,FI	$K' = \texttt{3736172D7421C91401596DB29D3D5536}_{\texttt{x}}$ ciphertext $= \texttt{A220257C87EA9458}_{\texttt{x}}$
FL$^{(-1)}$	$K' = \texttt{5F87F88EC7641D83AF03FD8327821046}_{\texttt{x}}$ ciphertext $= \texttt{B3EDCD6E923477CB}_{\texttt{x}}$
FL$^{(-1)}$,FI	$K' = \texttt{3736172D7421C91401596DB29D3D5536}_{\texttt{x}}$ ciphertext $= \texttt{D79D953A96A94A63}_{\texttt{x}}$
FL$^{(-1)}$,FO	$K' = \texttt{5F87F88EC7641D83AF03FD8327821046}_{\texttt{x}}$ ciphertext $= \texttt{140DE7077F1F156D}_{\texttt{x}}$
FL$^{(-1)}$,FO,FI (complete)	$K' = \texttt{3736172D7421C91401596DB29D3D5536}_{\texttt{x}}$ ciphertext $= \texttt{60081E65CB7C2B84}_{\texttt{x}}$

2. AES

2.1 Introduction

2.1.1 Overview

The Advanced Encryption Standard (AES) is the new symmetric encryption scheme adopted by the US National Institute of Standards and Technology (NIST). The standard is described in FIPS-197 [470] and specifies a symmetric block cipher called RIJNDAEL, designed by J. Daemen and V. Rijmen [183]. This cipher was selected in 2001 after a public evaluation process of more than two years and is intended to replace the existing Digital Encryption Standard (DES). The AES algorithm was not formally submitted to NESSIE but has been included in the evaluation as a widely used FIPS-approved algorithm for symmetric encryption.

2.1.2 Outline of the primitive

The RIJNDAEL block cipher was originally designed to handle a wide range of block sizes and key lengths. Only three specific combinations of block sizes and key lengths are supported by the FIPS standard, however. These three AES variants, referred to as AES-128, AES-192, and AES-256, have a fixed block size of 128 bits and accept keys of length 128, 192, and 256 bits respectively.

The AES block cipher is a Substitution Permutation Network (SPN) consisting of 10, 12, or 14 rounds, depending on the key length. The cipher takes a 128-bit plaintext block P as input and encrypts it into a 128-bit ciphertext block C, according to a secret key K. All operations during this iterative process are performed on an array of 4×4 bytes, which are interpreted as elements in the finite field $GF(2^8)$. A single round consists of a nonlinear byte substitution (`SubBytes`), a two-stage affine transformation (`ShiftRows`/`MixColumns`), and a round key addition (`AddRoundKey`). The round keys, each consisting of 128 bits, are derived from the original 128-bit, 192-bit, or 256-bit secret key by a separate key expansion routine.

The inverse process, which allows to recover the decrypted plaintext given the secret key, is very similar to the encrypting algorithm, but the different transformations are slightly modified.

2.1.3 Security and performance

The Rijndael design follows the *Wide Trail Strategy* and makes use of results from finite field and coding theory to optimize the different transformations. This guarantees the resistance against linear and differential attacks. As a result, the most successful attacks against round-reduced versions of the AES seem to be structural attacks such as improved *Square attacks* and collision attacks. These attacks cover up to 7 rounds (out of 10) for AES-128 and 9 (out of 14) for AES-256.

The AES algorithm is one of the most efficient block ciphers analyzed by NESSIE. The cipher allows fast software implementations on 32-bit platforms and the special structure of the components can be exploited to reduce the space requirements in hardware.

2.2 Description

This section intends to provide a short but complete specification of the AES algorithm. More extensive descriptions can be found in [470] and [183].

2.2.1 Cipher

The AES algorithm consists of a sequence of operations performed on an intermediate 4 × 4-byte array, called the *state* and denoted by $S = (s_{i,j})$. At the start of the encryption process, the 16 bytes of the state (128 bits) are initialized with plaintext bytes p_i, from top to bottom and from left to right as illustrated below:

$$\begin{array}{cccc} p_0 & p_4 & p_8 & p_{12} \\ p_1 & p_5 & p_9 & p_{13} \\ p_2 & p_6 & p_{10} & p_{14} \\ p_3 & p_7 & p_{11} & p_{15} \end{array} \xrightarrow{S=P} \begin{array}{cccc} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \end{array}$$

Note that in the initialization above, as well as in most of the subsequent operations, software implementations on 32-bit platforms can benefit from the fact that the 4 bytes of each *column* can be treated as a single 32-bit word.

After an initial round key addition, the state is transformed by N_r successive applications of a round function, the last round being slightly simplified. The final contents of the state is then sent to the output as ciphertext. The complete encryption algorithm can be described with the following pseudo-code:

```
$S =$ AddRoundKey($P$,$W_0$)
for $i = 1$ to $N_r - 1$ do
  $S =$ SubBytes($S$)
  $S =$ ShiftRows($S$)
```

```
    $S$ = MixColumns($S$)
    $S$ = AddRoundKey($S$,$W_i$)
  end for
  $S$ = SubBytes($S$)
  $S$ = ShiftRows($S$)
  $C$ = AddRoundKey($S$,$W_{N_r}$)
```

The number of rounds N_r depends on the key length and has been fixed to 10, 12, and 14 rounds for AES-128, AES-192, and AES-256 respectively. Note also that the MixColumns transform is skipped in the last round.

In the following subsections, each of the four transforms used above are specified separately.

2.2.1.1 The SubBytes transformation

The SubBytes operation substitutes each individual state byte $s_{i,j}$ by a new value $s'_{i,j} = S_{RD}(s_{i,j})$. The function S_{RD} is a fixed invertible nonlinear mapping (S-box), given by Table 53. Using this table, the byte $\texttt{53}_\texttt{x}$ for example would be mapped to $\texttt{ED}_\texttt{x}$.

$$\begin{array}{cccc} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \end{array} \xrightarrow{S(\cdot)} \begin{array}{cccc} s'_{0,0} & s'_{0,1} & s'_{0,2} & s'_{0,3} \\ s'_{1,0} & s'_{1,1} & s'_{1,2} & s'_{1,3} \\ s'_{2,0} & s'_{2,1} & s'_{2,2} & s'_{2,3} \\ s'_{3,0} & s'_{3,1} & s'_{3,2} & s'_{3,3} \end{array}$$

In most software environments, the S-box can simply be implemented as a lookup table, but when memory is restricted, it is possible to build more memory-efficient (but slower) implementations using the fact that the S-box is constructed from an inversion in $GF(2^8)$ followed by an affine transformation. For details on such implementations we refer to [181].

2.2.1.2 The ShiftRows transformation

The ShiftRows transformation operates on the rows of the state: the first row is kept unchanged, the remaining rows are cyclically shifted to the left over 1, 2, and 3 byte positions respectively. The operations are illustrated below:

$$\begin{array}{cccc} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \end{array} \xrightarrow{\circlearrowleft} \begin{array}{cccc} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,1} & s_{1,2} & s_{1,3} & s_{1,0} \\ s_{2,2} & s_{2,3} & s_{2,0} & s_{2,1} \\ s_{3,3} & s_{3,0} & s_{3,1} & s_{3,2} \end{array}$$

2.2.1.3 The MixColumns transformation

The MixColumns operation applies a linear transform to the columns of the state. This transform can be represented as a matrix multiplication, where each byte is interpreted as an element in the finite field $GF(2^8)$:

Table 53. The RIJNDAEL S-box S_{RD}

	$\mathtt{00}_\mathtt{x}$	$\mathtt{01}_\mathtt{x}$	$\mathtt{02}_\mathtt{x}$	$\mathtt{03}_\mathtt{x}$	$\mathtt{04}_\mathtt{x}$	$\mathtt{05}_\mathtt{x}$	$\mathtt{06}_\mathtt{x}$	$\mathtt{07}_\mathtt{x}$	$\mathtt{08}_\mathtt{x}$	$\mathtt{09}_\mathtt{x}$	$\mathtt{0A}_\mathtt{x}$	$\mathtt{0B}_\mathtt{x}$	$\mathtt{0C}_\mathtt{x}$	$\mathtt{0D}_\mathtt{x}$	$\mathtt{0E}_\mathtt{x}$	$\mathtt{0F}_\mathtt{x}$
$\mathtt{00}_\mathtt{x}$	63	7C	77	7B	F2	6B	6F	C5	30	01	67	2B	FE	D7	AB	76
$\mathtt{10}_\mathtt{x}$	CA	82	C9	7D	FA	59	47	F0	AD	D4	A2	AF	9C	A4	72	C0
$\mathtt{20}_\mathtt{x}$	B7	FD	93	26	36	3F	F7	CC	34	A5	E5	F1	71	D8	31	15
$\mathtt{30}_\mathtt{x}$	04	C7	23	C3	18	96	05	9A	07	12	80	E2	EB	27	B2	75
$\mathtt{40}_\mathtt{x}$	09	83	2C	1A	1B	6E	5A	A0	52	3B	D6	B3	29	E3	2F	84
$\mathtt{50}_\mathtt{x}$	53	D1	00	ED	20	FC	B1	5B	6A	CB	BE	39	4A	4C	58	CF
$\mathtt{60}_\mathtt{x}$	D0	EF	AA	FB	43	4D	33	85	45	F9	02	7F	50	3C	9F	A8
$\mathtt{70}_\mathtt{x}$	51	A3	40	8F	92	9D	38	F5	BC	B6	DA	21	10	FF	F3	D2
$\mathtt{80}_\mathtt{x}$	CD	0C	13	EC	5F	97	44	17	C4	A7	7E	3D	64	5D	19	73
$\mathtt{90}_\mathtt{x}$	60	81	4F	DC	22	2A	90	88	46	EE	B8	14	DE	5E	0B	DB
$\mathtt{A0}_\mathtt{x}$	E0	32	3A	0A	49	06	24	5C	C2	D3	AC	62	91	95	E4	79
$\mathtt{B0}_\mathtt{x}$	E7	C8	37	6D	8D	D5	4E	A9	6C	56	F4	EA	65	7A	AE	08
$\mathtt{C0}_\mathtt{x}$	BA	78	25	2E	1C	A6	B4	C6	E8	DD	74	1F	4B	BD	8B	8A
$\mathtt{D0}_\mathtt{x}$	70	3E	B5	66	48	03	F6	0E	61	35	57	B9	86	C1	1D	9E
$\mathtt{E0}_\mathtt{x}$	E1	F8	98	11	69	D9	8E	94	9B	1E	87	E9	CE	55	28	DF
$\mathtt{F0}_\mathtt{x}$	8C	A1	89	0D	BF	E6	42	68	41	99	2D	0F	B0	54	BB	16

$$\begin{bmatrix} s'_{0,i} \\ s'_{1,i} \\ s'_{2,i} \\ s'_{3,i} \end{bmatrix} = A \cdot \begin{bmatrix} s_{0,i} \\ s_{1,i} \\ s_{2,i} \\ s_{3,i} \end{bmatrix} = \begin{bmatrix} \mathtt{02}_\mathtt{x} & \mathtt{03}_\mathtt{x} & \mathtt{01}_\mathtt{x} & \mathtt{01}_\mathtt{x} \\ \mathtt{01}_\mathtt{x} & \mathtt{02}_\mathtt{x} & \mathtt{03}_\mathtt{x} & \mathtt{01}_\mathtt{x} \\ \mathtt{01}_\mathtt{x} & \mathtt{01}_\mathtt{x} & \mathtt{02}_\mathtt{x} & \mathtt{03}_\mathtt{x} \\ \mathtt{03}_\mathtt{x} & \mathtt{01}_\mathtt{x} & \mathtt{01}_\mathtt{x} & \mathtt{02}_\mathtt{x} \end{bmatrix} \cdot \begin{bmatrix} s_{0,i} \\ s_{1,i} \\ s_{2,i} \\ s_{3,i} \end{bmatrix}$$

As a result, the first byte of the column, for example, is replaced by $s'_{0,i} = (\mathtt{02}_\mathtt{x} \cdot s_{0,i}) \oplus (\mathtt{03}_\mathtt{x} \cdot s_{1,i}) \oplus s_{2,i} \oplus s_{3,i}$. The "$\oplus$" operator in this expression denotes addition in $GF(2^8)$, which corresponds to bitwise XOR (eXclusive OR). The multiplications are performed modulo the irreducible polynomial of the field. In the case of the AES algorithm the polynomial $x^8 + x^4 + x^3 + x + 1$ is used. For example:

$$\begin{aligned} \mathtt{03}_\mathtt{x} \cdot \mathtt{93}_\mathtt{x} &= \mathtt{00000011}_\mathtt{b} \cdot \mathtt{10010011}_\mathtt{b} \\ &= (x+1) \cdot (x^7 + x^4 + x + 1) && \bmod x^8 + x^4 + x^3 + x + 1 \\ &= x^8 + x^7 + x^5 + x^4 + x^2 + 1 && \bmod x^8 + x^4 + x^3 + x + 1 \\ &= x^7 + x^5 + x^3 + x^2 + x \\ &= \mathtt{10101110}_\mathtt{b} \\ &= \mathtt{AE}_\mathtt{x} \end{aligned}$$

The effect of the complete `MixColumns` operation is depicted below.

$$\begin{array}{cccc} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \end{array} \xrightarrow{A\times\cdot} \begin{array}{cccc} s'_{0,0} & s'_{0,1} & s'_{0,2} & s'_{0,3} \\ s'_{1,0} & s'_{1,1} & s'_{1,2} & s'_{1,3} \\ s'_{2,0} & s'_{2,1} & s'_{2,2} & s'_{2,3} \\ s'_{3,0} & s'_{3,1} & s'_{3,2} & s'_{3,3} \end{array}$$

2.2.1.4 The `AddRoundKey` transformation

Finally, in the `AddRoundKey` transformation, each bit of the state is XORed with the corresponding bit of a 128-bit round key W_i. Each round requires a separate round key, and the generation of these keys is performed by the key expansion routine described in the next subsection.

$$\begin{array}{cccc} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \end{array} \xrightarrow{\cdot \oplus W_i} \begin{array}{cccc} s'_{0,0} & s'_{0,1} & s'_{0,2} & s'_{0,3} \\ s'_{1,0} & s'_{1,1} & s'_{1,2} & s'_{1,3} \\ s'_{2,0} & s'_{2,1} & s'_{2,2} & s'_{2,3} \\ s'_{3,0} & s'_{3,1} & s'_{3,2} & s'_{3,3} \end{array}$$

2.2.2 Key Expansion

The purpose of the key expansion is to construct $(N_r + 1)$ 128-bit round keys W_i from a single 128-bit secret key K. Once these round keys have been derived, an unlimited number of plaintext blocks can be encrypted, i.e., the key expansion routine only needs to be repeated when the secret key is changed.

The AES key expansion routine is a recursive process in which each new round key is directly derived from the preceding keys. If the round keys W_i are represented by 4×4-bytes arrays, the routine exclusively consists of column operations. The key expansion in AES-128 proceeds according to the pseudo-code below:

```
$W_0 = K$
for $i = 1$ to 10 do
    $W_{i,0} = W_{i-1,0} \oplus \texttt{SubBytes}^*(\texttt{ShiftColumn}(W_{i-1,3})) \oplus R_C^i$
    $W_{i,1} = W_{i-1,1} \oplus W_{i,0}$
    $W_{i,2} = W_{i-1,2} \oplus W_{i,1}$
    $W_{i,3} = W_{i-1,3} \oplus W_{i,2}$
end for
```

The first round key W_0, used in the initial key addition, is directly filled with the 16 bytes of the secret key K. The 32-bit columns $W_{i,j}$ of the remaining round keys are derived recursively. The function `SubBytes`* substitutes the bytes of a single column in the same way as the `SubBytes` transformation described in Sect. 2.2.1.1. The `ShiftColumn` operation is an upward cyclic shift over one byte position. The constants R_C^i are fixed 4-byte columns defined as $(\texttt{02}_\texttt{x}^{i-1}, \texttt{00}_\texttt{x}, \texttt{00}_\texttt{x}, \texttt{00}_\texttt{x})^T$ with $\texttt{02}_\texttt{x}$ representing the element x in $GF(2^8)$ (using the same irreducible polynomial $x^8 + x^4 + x^3 + x + 1$).

The key expansion routine used in AES-192 is very similar (we refer to [470] for the details), but the expansion in AES-256 is slightly different:

```
$W_0 = K_0$
$W_1 = K_1$
for $i = 2$ to 14 do
```

```
    if $i$ is even then
        $W_{i,0} = W_{i-2,0} \oplus \texttt{SubBytes}^{*}\texttt{(ShiftColumn(}W_{i-1,3}\texttt{))} \oplus R_C^{i/2}$
    else
        $W_{i,0} = W_{i-2,0} \oplus \texttt{SubBytes}^{*}\texttt{(}W_{i-1,3}\texttt{)}$
    end if
    $W_{i,1} = W_{i-2,1} \oplus W_{i,0}$
    $W_{i,2} = W_{i-2,2} \oplus W_{i,1}$
    $W_{i,3} = W_{i-2,3} \oplus W_{i,2}$
end for
```

In the code above, K_0 and K_1 represent the first and the second half of the 256-bit key respectively.

2.2.3 Inverse Cipher

All transformations described in Sect. 2.2.1 are invertible and a straightforward implementation of the decryption algorithm would simply consist in applying the inverted operations in reverse order (without modifying the key expansion). This section presents an equivalent inverse cipher, which slightly modifies the key schedule, but has the advantage of using the same sequence of transformations as the original cipher. The pseudo-code is given below:

```
$S = \texttt{AddRoundKey(}C\texttt{,}W'_{N_r}\texttt{)}$
for $i = N_r - 1$ to 1 do
    $S = \texttt{SubBytes}^{-1}\texttt{(}S\texttt{)}$
    $S = \texttt{ShiftRows}^{-1}\texttt{(}S\texttt{)}$
    $S = \texttt{MixColumns}^{-1}\texttt{(}S\texttt{)}$
    $S = \texttt{AddRoundKey(}S\texttt{,}W'_i\texttt{)}$
end for
$S = \texttt{SubBytes}^{-1}\texttt{(}S\texttt{)}$
$S = \texttt{ShiftRows}^{-1}\texttt{(}S\texttt{)}$
$P = \texttt{AddRoundKey(}S\texttt{,}W'_0\texttt{)}$
```

The inverse transformations $\texttt{SubBytes}^{-1}$ and $\texttt{ShiftRows}^{-1}$ are easily derived from the descriptions in Sect. 2.2.1. The inverse of $\texttt{MixColumns}$ requires the matrix A to be inverted. The resulting linear transformation is given by:

$$\begin{bmatrix} s'_{0,i} \\ s'_{1,i} \\ s'_{2,i} \\ s'_{3,i} \end{bmatrix} = A^{-1} \cdot \begin{bmatrix} s_{0,i} \\ s_{1,i} \\ s_{2,i} \\ s_{3,i} \end{bmatrix} = \begin{bmatrix} \texttt{0E}_{\texttt{x}} & \texttt{0B}_{\texttt{x}} & \texttt{0D}_{\texttt{x}} & \texttt{09}_{\texttt{x}} \\ \texttt{09}_{\texttt{x}} & \texttt{0E}_{\texttt{x}} & \texttt{0B}_{\texttt{x}} & \texttt{0D}_{\texttt{x}} \\ \texttt{0D}_{\texttt{x}} & \texttt{09}_{\texttt{x}} & \texttt{0E}_{\texttt{x}} & \texttt{0B}_{\texttt{x}} \\ \texttt{0B}_{\texttt{x}} & \texttt{0D}_{\texttt{x}} & \texttt{09}_{\texttt{x}} & \texttt{0E}_{\texttt{x}} \end{bmatrix} \cdot \begin{bmatrix} s_{0,i} \\ s_{1,i} \\ s_{2,i} \\ s_{3,i} \end{bmatrix}$$

The new round keys W'_i used in the equivalent inverse cipher are computed from the original round keys as follows:

$$W'_i = \begin{cases} W_i & \text{for } i = 0 \text{ or } i = N_r, \\ \texttt{MixColumns}^{-1}\texttt{(}W_i\texttt{)} & \text{for } 1 \le i \le N_r - 1. \end{cases}$$

2.3 Test vectors

Test vectors are available in FIPS-197 [470].

3. Camellia

3.1 Introduction

3.1.1 Overview

Camellia was first published in 2000 and is developed jointly by NTT and Mitsubishi Electric. The design of Camellia is based on E2 and MISTY. E2 was designed by NTT and submitted as an AES candidate, and MISTY was designed by Mitsubishi Electric and submitted for the 3GPP's confidentiality and integrity function.

which was a previous block cipher by the same designers and was a submission to the AES. The main difference between E2 and Camellia is the adoption, for Camellia, of the 1-round Substitution-Permutation Network (SPN), not the 2-round SPN of E2, leading to an expected improvement in speed for Camellia. Camellia has a block size of 128 bits and uses 128, 192, or 256-bit keys. Camellia is aimed at a wide range of platforms, from low-power hardware applications and environments with limited resources to resource-intensive environments. It is designed to require only byte-oriented operations to ensure a strong performance on software platforms.

3.1.2 Outline of the primitive

Camellia [24] is an 18 (or 24)-round 128-bit block cipher which supports 128-, (or 192-, and 256-) bit key lengths, with a layer of 64-bit FL-blocks after the 6th, 12th (and 18th) rounds, which introduce round irregularity into the cipher. It is a byte-oriented Feistel cipher with a particular emphasis on low-cost hardware applications, and is designed to be resistant to differential and linear cryptanalysis. Camellia uses four 8×8-bit S-boxes with input and output affine transformations and logical operations. The diffusion layer uses a linear transformation based on a Maximum-Distance-Separable code with a branch number of 5. The FL functions are similar to those of MISTY, except that Camellia also uses a 1-bit rotation so as to make bytewise cryptanalysis harder. Decryption for Camellia is very similar to encryption except that the order in which the round keys are introduced is reversed [1] .

[1] In the original specification, the term ”subkey” is used instead of ”round key”.

3.1.3 Security and performance

Camellia has not only been examined by the NESSIE project but also by the ISO and CRYPTREC standardization bodies, as well as by the more general research community, and no security flaws have been found. The designers [25] claim that for Camellia no differential/linear characteristics exist with linear bias and differential probabilities $> 2^{-128}$ over 12 rounds and 2^{-132} over 15 rounds. This claim is based on the theoretical lower bounds of the number of active S-boxes and the use of S-boxe which are affine transformations of x^{-1} over GF(2^8). This particular mathematical function ensures optimal differential and linear characteristics through the S-box. The cipher also obtains high diffusion by using a linear transformation for the diffusion layer with a high branch number of 5. The designers claim that 10 rounds is indistinguishable from a random permutation with respect to truncated differential and linear cryptanalysis, and also claim security against interpolation, linear sum, and Square attacks. The introduction of the FL functions after every 6 rounds provides resistance to Slide attacks.

The security of Camellia against the Square attack is discussed by Yeom *et al.* in [627]. This attack may be extended up to 9 rounds including the first FL/FL^{-1} layer by considering the key schedule. In [578] Shirai *et al.* discuss the security of Camellia against differential and linear attack and it is shown that 10-round Camellia without FL/FL^{-1} has no differential and linear characteristic with probability higher than 2^{-128}. Shirai obtains a differential attack on 11 rounds without FL/FL^{-1} layers using an 8-round characteristic, and a linear attack on 12 rounds without FL/FL^{-1} layers using a 9-round linear approximation. Shirai [577] also proposed boomerang and rectangle attacks on 10-round Camellia with FL/FL^{-1} layers. They use a technique developed by Biham *et al.* [68, 69]. Truncated and impossible differential cryptanalysis of Camellia (without FL/FL^{-1} functions) is described by Sugita *et al.* [602]. A Square attack on Camellia is proposed by He and Qing in [292]. 3-round and 7-round iterative differential characteristics with probability 2^{-52} have been found by Biham *et al.* in [67], which can be iterated to further rounds. A higher-order differential attack on 10 rounds of 256-bit key Camellia without FL rounds is performed by Kawabata and Kaneko in [339], leading to 9 and 8 rounds on 192-bit and 128-bit key versions, respectively.

Camellia's S-boxes are based on the x^{-1} function, similar to the AES and, as pointed out in [165], the Camellia (and AES) S-box can be described by a system of 23 quadratic equations in 80 terms. Hence it is potentially open to algebraic attacks (none found yet), for instance using the Big Encryption System (BES) as described by Murphy and Robshaw [452, 454]. However Camellia also inserts FL and FL^{-1} layers every six rounds and these should make any future algebraic attacks more complicated in comparison to the AES. But, if an attack using BES and/or a system of overdefined quadratic equations was ever successful on AES, then it might also be quite successful on Camellia. Fuller and Millan [245] recently observed that the bit-output functions of the x^{-1} S-box are all affine transformations of the same function. This suggests a potential hardware saving for the Camellia S-box, although this may later also become a security weakness.

3.2 Description

Camellia is a byte-oriented Feistel cipher taking 128-bit plaintext blocks. It requires 18 rounds for a 128-bit, and 24 rounds for a 192-bit or 256-bit key. We will now describe each module.

3.2.1 Encryption - 128-bit Key

The encryption process for 128-bit keys operates over 18 rounds and is shown in Fig 53. It comprises a series of three 6-round Feistel operations separated by FL layers. The 128-bit input plaintext, M, is first split into two 64-bit halves, L_0 and R_0, where $M = L_0 \| R_0$. Then L_0 and R_0 are bitwise XOR'ed with the 64-bit round keys, kw_1 and kw_2, respectively, before passing into the first set of 6-round Feistel modules. Each round of a 6-round Feistel block passes 64-bit L_i through a keyed F-function, keyed by a 64-bit round key, k_i, with the 64-bit output of the F-function, L'_i, bitwise XOR'ed with R_i. At the end of each round, 64-bit L_i and R_i are assigned to R_{i+1} and L_{i+1}, respectively. After each of the first two 6-round Feistel blocks, 64-bit L_i and R_i are passed through an FL and FL^{-1} layer, respectively, where the FL and FL^{-1} layers are keyed by 64-bit kl_j and kl_{j+1}, respectively. After the third and last 6-round Feistel block, 64-bit L_{18} and R_{18} are interchanged before a final bitwise XOR with 64-bit kw_3 and kw_4, respectively. Finally the ciphertext, C, is constructed, where $C = (R_{18} \oplus kw_3) \| (L_{18} \oplus kw_4)$.

We can summarise the encryption inputs as,

$$\begin{aligned}
&M = L_0 \| R_0 \\
&k = \{k_1, k_2, \ldots, k_{18}\} \\
&kl = \{kl_1, kl_2, kl_3, kl_4\} \\
&kw = \{kw_1, kw_2, kw_3, kw_4\}
\end{aligned}$$

The encryption function is then summarised as,

$$\begin{aligned}
&(L_6 \| R_6) = \text{6RoundFeistel}(L_0 \oplus kw_1 \| R_0 \oplus kw_2, k_1, k_2, k_3, k_4, k_5, k_6) \\
&L_6 = \text{FL}(L_6, kl_1) \\
&R_6 = \text{FL}^{-1}(R_6, kl_2) \\
&(L_{12} \| R_{12}) = \text{6RoundFeistel}(L_6 \| R_6, k_7, k_8, k_9, k_{10}, k_{11}, k_{12}) \\
&L_{12} = \text{FL}(L_{12}, kl_3) \\
&R_{12} = \text{FL}^{-1}(R_{12}, kl_4) \\
&(L_{18} \| R_{18}) = \text{6RoundFeistel}(L_{12} \| R_{12}, k_{13}, k_{14}, k_{15}, k_{16}, k_{17}, k_{18}) \\
&C = (R_{18} \oplus kw_3) \| (L_{18} \oplus kw_4)
\end{aligned}$$

where 6RoundFeistel is described as,

```
for $j = 0$ to 5 do
    $L_{i+1+j} = \text{F}(L_{i+j}, k_{i+j+1}) \oplus R_{i+j}$
    $R_{i+1+j} = L_{i+j}$
```

and where $i = 0$, 6 or 12.

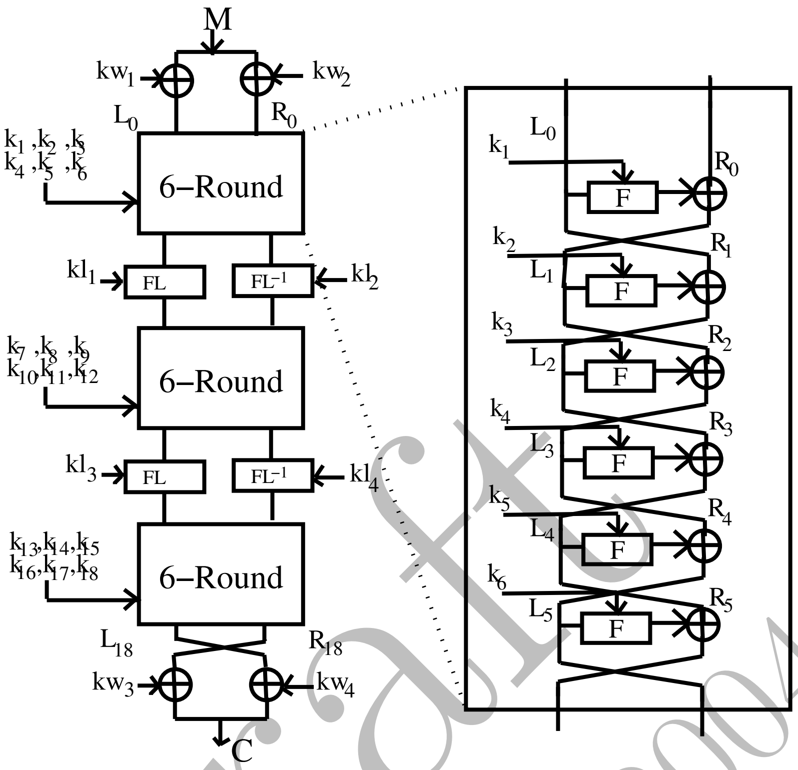


Fig. 53. Encryption for Camellia for 128-bit Key

The above variables, $L_i, R_i, k_i, kl_i, kw_j$ are all 64-bits wide, and M and C are both 128-bits wide.

The order in which the round keys are used is as follows:

$$\begin{gathered} kw_1, kw_2 \\ k_1, k_2, k_3, k_4, k_5, k_6 \\ kl_1, kl_2 \\ k_7, k_8, k_9, k_{10}, k_{11}, k_{12} \\ kl_3, kl_4 \\ k_{13}, k_{14}, k_{15}, k_{16}, k_{17}, k_{18} \\ kw_3, kw_4 \end{gathered}$$

3.2.2 Decryption - 128-bit Key

The decryption process for 128-bit keys is shown in Fig 54, and is identical in operation to encryption apart from the position and ordering of the round keys, which are reversed. To be more explicit, for decryption the round keys are input as follows:

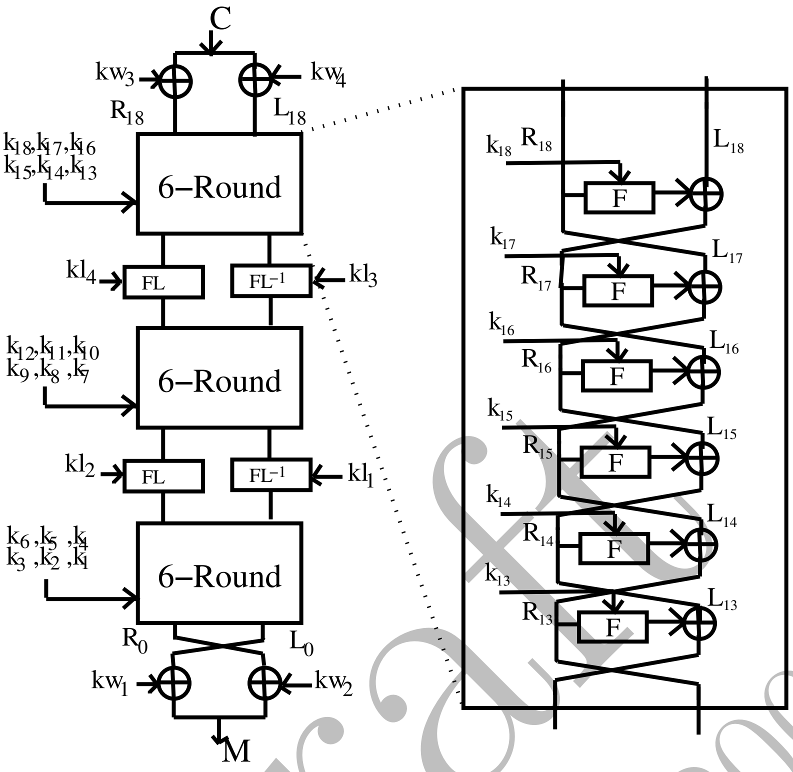


Fig. 54. Decryption for Camellia for 128-bit Key

$$\begin{gathered}
kw_3, kw_4 \\
k_{18}, k_{17}, k_{16}, k_{15}, k_{14}, k_{13} \\
kl_4, kl_3 \\
k_{12}, k_{11}, k_{10}, k_9, k_8, k_7 \\
kl_2, kl_1 \\
k_6, k_5, k_4, k_3, k_2, k_1 \\
kw_1, kw_2
\end{gathered}$$

3.2.3 Encryption - 192-bit and 256-bit Keys

The encryption process for 192-bit or 256-bit keys operates over 24 rounds and is shown in Fig 55. It is similar to the encryption process for 128-bit keys. The only difference is that an extra 6-round Feistel operation and an FL layer are inserted. It therefore comprises four 6-round Feistel operations separated by FL layers. We therefore require the generation and input of more round keys.

We can summarise the encryption inputs as,

$$\begin{aligned}
&M = L_0 \| R_0 \\
&k = \{k_1, k_2, \ldots, k_{24}\} \\
&kl = \{kl_1, kl_2, kl_3, kl_4, kl_5, kl_6\} \\
&kw = \{kw_1, kw_2, kw_3, kw_4\}
\end{aligned}$$

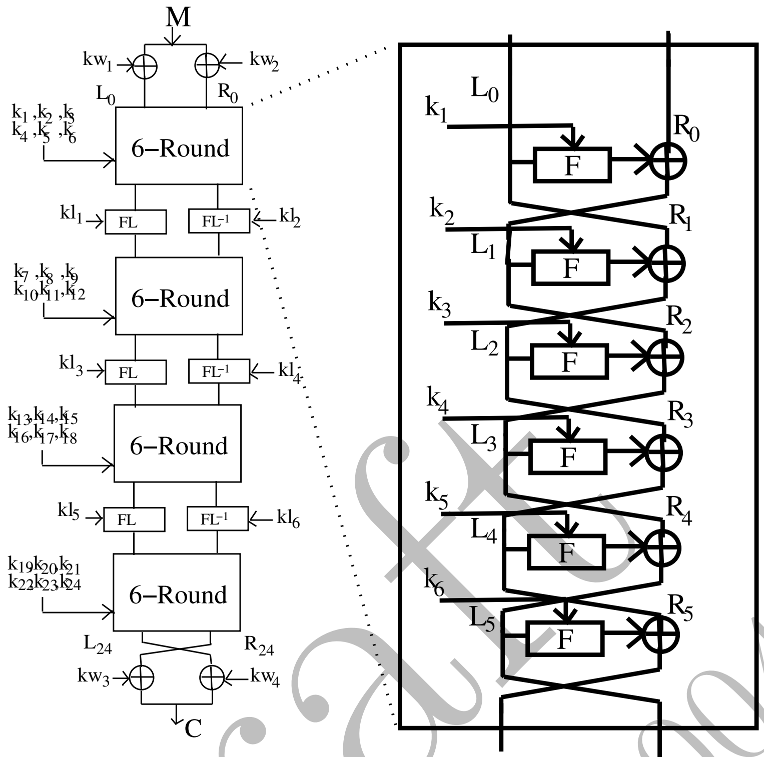


Fig. 55. Encryption for Camellia for 192-bit and 256-bit Keys

The encryption function is then summarised as,

$$
\begin{aligned}
&(L_6 \| R_6) = \text{6RoundFeistel}(L_0 \oplus kw_1 \| R_0 \oplus kw_2, k_1, k_2, k_3, k_4, k_5, k_6)\\
&L_6 = \text{FL}(L_6, kl_1)\\
&R_6 = \text{FL}^{-1}(R_6, kl_2)\\
&(L_{12} \| R_{12}) = \text{6RoundFeistel}(L_6 \| R_6, k_7, k_8, k_9, k_{10}, k_{11}, k_{12})\\
&L_{12} = \text{FL}(L_{12}, kl_3)\\
&R_{12} = \text{FL}^{-1}(R_{12}, kl_4)\\
&(L_{18} \| R_{18}) = \text{6RoundFeistel}(L_{12} \| R_{12}, k_{13}, k_{14}, k_{15}, k_{16}, k_{17}, k_{18})\\
&L_{18} = \text{FL}(L_{18}, kl_5)\\
&R_{18} = \text{FL}^{-1}(R_{18}, kl_6)\\
&C = (R_{24} \oplus kw_3) \| (L_{24} \oplus kw_4)\\
&(L_{24} \| R_{24}) = \text{6RoundFeistel}(L_{18} \| R_{18}, k_{19}, k_{20}, k_{21}, k_{22}, k_{23}, k_{24})
\end{aligned}
$$

The order in which the round keys are used is as follows:

$$kw_1, kw_2$$
$$k_1, k_2, k_3, k_4, k_5, k_6$$
$$kl_1, kl_2$$
$$k_7, k_8, k_9, k_{10}, k_{11}, k_{12}$$
$$kl_3, kl_4$$
$$k_{13}, k_{14}, k_{15}, k_{16}, k_{17}, k_{18}$$
$$kl_5, kl_6$$
$$k_{19}, k_{20}, k_{21}, k_{22}, k_{23}, k_{24}$$
$$kw_3, kw_4$$

3.2.4 Decryption - 192-bit and 256-bit Keys

The decryption process for 192-bit and 256-bit keys is shown in Fig 56, and is identical in operation to encryption apart from the position and ordering of the round keys, which are reversed. To be more explicit, for decryption the round keys are input as follows:

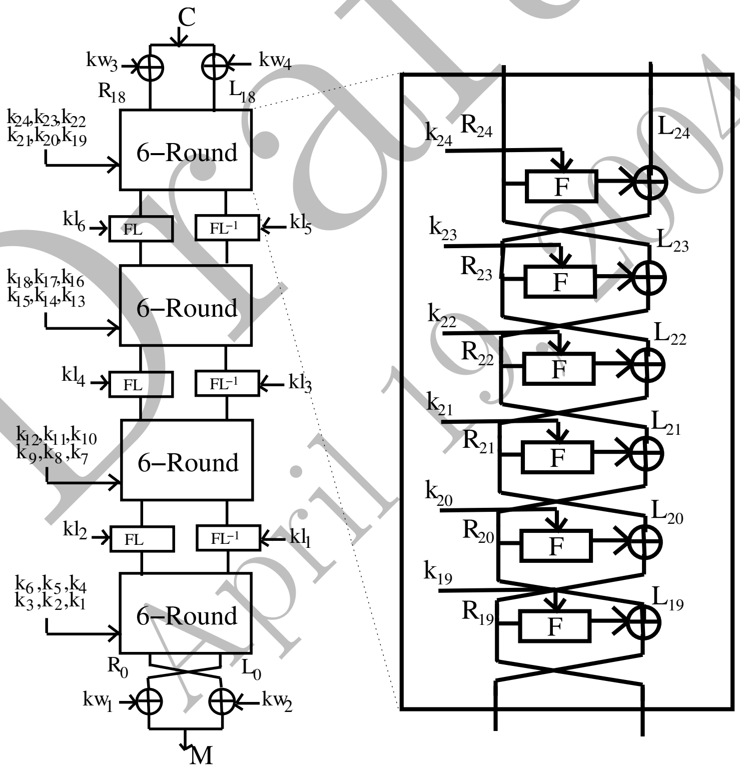


Fig. 56. Decryption for Camellia for 192-bit and 256-bit Keys

$$\begin{gathered} kw_3, kw_4 \\ k_{24}, k_{23}, k_{22}, k_{21}, k_{20}, k_{19} \\ kl_6, kl_5 \\ k_{18}, k_{17}, k_{16}, k_{15}, k_{14}, k_{13} \\ kl_4, kl_3 \\ k_{12}, k_{11}, k_{10}, k_9, k_8, k_7 \\ kl_2, kl_1 \\ k_6, k_5, k_4, k_3, k_2, k_1 \\ kw_1, kw_2 \end{gathered}$$

3.2.5 F-Function

The F-function is shown in Fig 57. The F-function comprises a bitwise XOR, followed by an application of 8 parallel 8×8 S-Boxes, followed by a diffusion layer (the P-function). To be more specific, the input is bits x_1 to x_8 where,

$$L_i = x_1 \| x_2 \| x_3 \| x_4 \| x_5 \| x_6 \| x_7 \| x_8$$

and this 64-bit input, L_i, is first bitwise XOR'ed with a 64-bit key, k_i, and is then partitioned into eight 8-bit segments, y_j, such that,

$$L_i \oplus k_i = y_1 \| y_2 \| y_3 \| y_4 \| y_5 \| y_6 \| y_7 \| y_8$$

Each y_j is then passed through an 8×8 S-box, s_t, to give eight 8-bit segments, z_j, where,

$$\begin{gathered} z_1 = s_1[y_1], z_2 = s_2[y_2], z_3 = s_3[y_3], z_4 = s_4[y_4], \\ z_5 = s_2[y_5], z_6 = s_3[y_6], z_7 = s_4[y_7], z_8 = s_1[y_8] \end{gathered}$$

The eight 8-bit segments, z_j, are then acted on by the P-function, which is a diffusion layer which outputs eight 8-bit segments, z'_j, where,

$$\begin{aligned} z'_1 &= z_1 \oplus z_3 \oplus z_4 \oplus z_6 \oplus z_7 \oplus z_8 \\ z'_2 &= z_1 \oplus z_2 \oplus z_4 \oplus z_5 \oplus z_7 \oplus z_8 \\ z'_3 &= z_1 \oplus z_2 \oplus z_3 \oplus z_5 \oplus z_6 \oplus z_8 \\ z'_4 &= z_2 \oplus z_3 \oplus z_4 \oplus z_5 \oplus z_6 \oplus z_7 \\ z'_5 &= z_1 \oplus z_2 \oplus z_6 \oplus z_7 \oplus z_8 \\ z'_6 &= z_2 \oplus z_3 \oplus z_5 \oplus z_7 \oplus z_8 \\ z'_7 &= z_3 \oplus z_4 \oplus z_5 \oplus z_6 \oplus z_8 \\ z'_8 &= z_1 \oplus z_4 \oplus z_5 \oplus z_6 \oplus z_7 \end{aligned}$$

The P-function can alternatively be represented in matrix-vector form as,

$$\begin{pmatrix} z_8 \\ z_7 \\ \vdots \\ z_1 \end{pmatrix} \rightarrow \begin{pmatrix} z'_8 \\ z'_7 \\ \vdots \\ z'_1 \end{pmatrix} = P \begin{pmatrix} z_8 \\ z_7 \\ \vdots \\ z_1 \end{pmatrix}$$

where,

$$P = \begin{pmatrix} 0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 \end{pmatrix}$$

The 64-bit output of the F-function, L_i', is then constructed by concatenating the 8-bit z_j', where,

$$L_i' = z_1' \| z_2' \| z_3' \| z_4' \| z_5' \| z_6' \| z_7' \| z_8'$$

The above variables, y_j, z_j, z_j', are 8-bits wide, and the variables, L_i, k_i, L_i' are 64-bits wide.

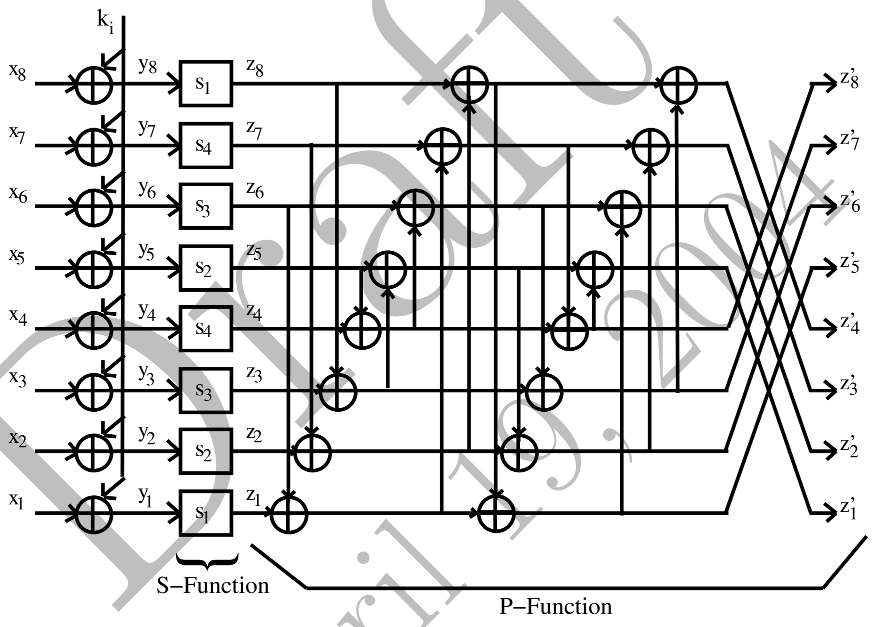


Fig. 57. F-Function for Camellia

3.2.6 FL-Function

The FL and FL^{-1}-functions are shown in Fig 58. The FL-function takes as input a 64-bit word, X, which is then split into two 32-bit parts, X_L and X_R, where,

$$X = X_L \| X_R$$

The right half, X_R, is modified by bitwise XOR'ing with a modified version of the left half after the left half, X_L, has been bitwise AND'ed with a 32-bit key,

kl_{iL}, and the 32-bit result rotated left by one bit. (AND is shown as $\wedge$ in Fig. 58). Following this, the left half is modified by bitwise XOR'ing with a modified version of the right half after the right half has been bitwise OR'ed with a 32-bit key, kl_{iR}, (OR is shown as $\vee$ in Fig. 58). Finally the 32-bit left and 32-bit right halves, Y_L and Y_R, are concatenated to form a 64-bit output, Y where,

$$Y = Y_L \| Y_R$$

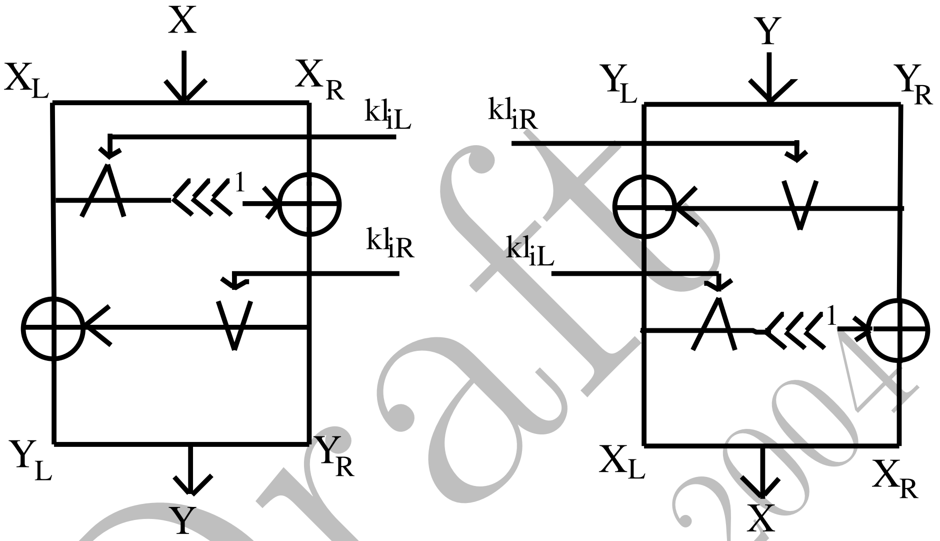


Fig. 58. FL and FL^{-1} Functions for Camellia

We can summarise the FL inputs as,

$$X = X_L \| X_R$$
$$kl_i = kl_{iL} \| kl_{iR}$$

The FL function is then defined as,

$$Y_R = ((X_L \wedge kl_{iL}) \lll_1) \oplus X_R$$
$$Y_L = (Y_R \vee kl_{iR}) \oplus X_L$$

and the output is $Y = Y_L \| Y_R$.

The above variables, $X_L, X_R, Y_L, Y_R, kl_{iL}, kl_{iR}$, are 32-bits wide, and the variables, X, Y are 64-bits wide.

3.2.7 FL^{-1}-Function

The FL and FL^{-1}-functions are shown in Fig 58. The FL^{-1}-function takes as input a 64-bit word, Y, which is then split into two 32-bit parts, Y_L and Y_R, where,

$$Y = Y_L \| Y_R$$

The left half, Y_L, is modified by bitwise XOR'ing with a modified version of the right half after the right half, Y_R, has been bitwise OR'ed with a 32-bit key, kl_{iR} (OR is shown as $\vee$ in Fig. 58). Following this, the right half is modified by bitwise XOR'ing with a modified version of the left half where the left half has been bitwise AND'ed with a 32-bit key, kl_{iL}, and the 32-bit result rotated left by one bit (AND is shown as $\wedge$ in Fig. 58). Finally the 32-bit left and 32-bit right halves, X_L and X_R, are concatenated to form a 64-bit output, X.

We can summarise the FL^{-1} inputs as,

$$\begin{aligned} Y &= Y_L \| Y_R \\ kl_i &= kl_{iL} \| kl_{iR} \end{aligned}$$

The FL^{-1} function is then defined as,

$$\begin{aligned} X_L &= (Y_R \vee kl_{iR}) \oplus Y_L \\ X_R &= ((X_L \wedge kl_{iL}) \lll_1) \oplus Y_R \end{aligned}$$

where the output is $X = X_L \| X_R$.

The above variables, $Y_L, Y_R, X_L, X_R, kl_{iL}, kl_{iR}$, are 32-bits wide, and the variables, Y, X are 64-bits wide.

3.2.8 Key Schedule

The Key Schedule is shown in Fig 59. For the 128-bit key version of Camellia, the user key, K, is the 128-bit key, K_L, with the 128-bit key, K_R, set to all zero bits. Thus,

$$K = K_L, \qquad K_R = 0$$

For the 192-bit key version of Camellia, the user key, K, is the 128-bit key, K_L, and the leftmost 64-bits of K_R, K_{RL}, with *the rightmost 64-bits of K_R, K_{RR}, set to the bitwise negation of the leftmost 64-bits of K_R, K_{RL}*. Thus,

$$K = K_L \| K_{RL}, \qquad K_{RR} = \overline{K_{RL}}, \qquad K_R = K_{RL} \| K_{RR}$$

For the 256-bit key version of Camellia, the user key, K, is the 128-bit key, K_L, and the 128-bit key, K_R. Thus,

$$K = K_L \| K_R$$

The key schedule of Camellia makes use of the F-function of the encryption module, and is the same for encryption and decryption. The user key, K, is encrypted by means of the F-function using pre-fixed constants, where these constants, Σ_i, are defined as continuous values from the hexadecimal representation of the square root of the ith prime. The round keys are then generated partly from rotated values of the user-input key, K (where $K = K_L$, $K = K_L \| K_{RL}$, or $K = K_L \| K_R$, for a 128-bit, 192-bit, or 256-bit key, K, respectively), and partly from rotated values of the encrypted keys, K_A and K_B.

For the 128-bit key version of Camellia, the output of the key schedule is the 128-bit encrypted key, K_A, with the right-hand side of Fig 59 omitted and K_B not generated or used. For the 192-bit and 256-bit key versions of Camellia, the outputs of the key schedule are the 128-bit encrypted key, K_A, and the 128-bit encrypted key, K_B. The key schedule comprises 2 or 3 2-round Feistel blocks for 128-bit or 192/256-bit key versions, respectively. Each Feistel block is 'keyed' by a pair of constants, Σ_i.

The 128-bit input to the first 2-round Feistel block on the left side of Fig 59 is $K_L \oplus K_R$, and this Feistel block is 'keyed' by two 64-bit constants, Σ_1 and Σ_2. The 128-bit output from the first 2-round Feistel block is then bitwise XOR'ed with K_L before input to the second 2-round Feistel block on the left side of Fig 59. This second Feistel block is 'keyed' by two 64-bit constants, Σ_3 and Σ_4. The 128-bit output from this second 2-round Feistel block is K_A. For 192-bit or 256-bit key versions of Camellia, K_A is then bitwise XOR'ed with the 128-bit key, K_R, before inputing the result to a third 2-round Feistel block, which is on the right side of Fig 59. This third Feistel block is 'keyed' by two 64-bit constants, Σ_5 and Σ_6. The 128-bit output from this third 2-round Feistel block is K_B.

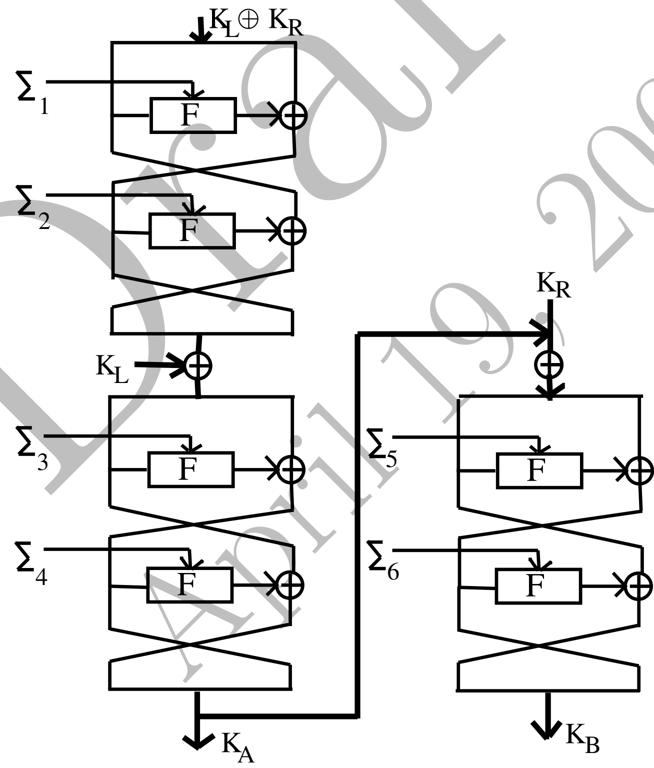


Fig. 59. Key Schedule for Camellia

The key schedule is then summarised as,

$$\begin{aligned}
K_a &= \text{2RoundFeistel}(K_L \oplus K_R, \Sigma_1, \Sigma_2)\\
K_A &= \text{2RoundFeistel}(K_a \oplus K_L, \Sigma_3, \Sigma_4)\\
K_B &= \text{2RoundFeistel}(K_A \oplus K_R, \Sigma_5, \Sigma_6) \qquad \text{(192/256-bit key only)}
\end{aligned}$$

where the 128-bit input to 2RoundFeistel is split into two 64-bit parts, $L_0 \| R_0$, the 128-bit output from 2RoundFeistel is also split into two 64-bit parts, $L_2 \| R_2$, and the two 64-bit 'key' inputs to 2RoundFeistel are Σ_i and Σ_{i+1}.

2RoundFeistel is then described as,

```
for $j = 0$ to 1 do
    $L_{j+1} = \mathrm{F}(L_j, \Sigma_{i+j}) \oplus R_j$
    $R_{j+1} = L_j$
```

where $i = 1$, 3 or 5.

The above variables, L_j, R_j, Σ_i are all 64-bits wide, and K_L, K_R, K_a, K_A, K_B are all 128-bits wide.

The order in which the constants, Σ_i, are input is as follows:

$$\begin{gathered}
\Sigma_1, \Sigma_2\\
\Sigma_3, \Sigma_4\\
\Sigma_5, \Sigma_6
\end{gathered}$$

These 64-bit key schedule constants are as follows,

$$\begin{aligned}
\Sigma_1 &= 0xa09e667f3bcc908b\\
\Sigma_2 &= 0xb67ae8584caa73b2\\
\Sigma_3 &= 0xc6ef372fe94f82be\\
\Sigma_4 &= 0x54ff53a5f1d36f1c\\
\Sigma_5 &= 0x10e527fade682d1d\\
\Sigma_6 &= 0xb05688c2b3e6c1fd
\end{aligned}$$

Finally, the 64-bit round keys, k, kw, and kl are derived from the 128-bit keys, K_L, K_R, K_A, and K_B, as shown in the following tables. Table 54 is for the 128-bit key version of Camellia, and Table 55 is for the 192- or 256-bit key version of Camellia. Note that $(X \lll_r)_L$ is the leftmost 64-bits of the 128-bit value, $(X \lll_r)$ which, in turn, is the 128-bit value, X, cyclically rotated left by r bits. Similarly, $(X \lll_r)_R$ is the rightmost 64-bits of the 128-bit value, $(X \lll_r)$.

For decryption the order of the rounds is reversed.

3.2.9 S-boxes

The four s-boxes of Camellia are affine equivalent to an inversion function, x^{-1}, over GF(2^8). They do not have compact boolean Algebraic Normal Forms (ANFs). However, they are compactly described as arithmetic operations over GF(2^8), followed by binary affine operations, where the 8-bit input, x, is partitioned into 1-bit segments where,

$$x = x_1 \| x_2 \| x_3 \| x_4 \| x_5 \| x_6 \| x_7 \| x_8$$

Table 54. Extracting Round Keys from User Key and Encrypted Keys for 128-bit Key Version of Camellia (shown for encryption only)

	round key	value
Prewhitening	kw_1	$(K_L \lll_0)_L$
	kw_2	$(K_L \lll_0)_R$
F(Round 1)	k_1	$(K_A \lll_0)_L$
F(Round 2)	k_2	$(K_A \lll_0)_R$
F(Round 3)	k_3	$(K_L \lll_{15})_L$
F(Round 4)	k_4	$(K_L \lll_{15})_R$
F(Round 5)	k_5	$(K_A \lll_{15})_L$
F(Round 6)	k_6	$(K_A \lll_{15})_R$
FL	kl_1	$(K_A \lll_{30})_L$
FL^{-1}	kl_2	$(K_A \lll_{30})_R$
F(Round 7)	k_7	$(K_L \lll_{45})_L$
F(Round 8)	k_8	$(K_L \lll_{45})_R$
F(Round 9)	k_9	$(K_A \lll_{45})_L$
F(Round 10)	k_{10}	$(K_L \lll_{60})_R$
F(Round 11)	k_{11}	$(K_A \lll_{60})_L$
F(Round 12)	k_{12}	$(K_A \lll_{60})_R$
FL	kl_3	$(K_L \lll_{77})_L$
FL^{-1}	kl_4	$(K_L \lll_{77})_R$
F(Round 13)	k_{13}	$(K_L \lll_{94})_L$
F(Round 14)	k_{14}	$(K_L \lll_{94})_R$
F(Round 15)	k_{15}	$(K_A \lll_{94})_L$
F(Round 16)	k_{16}	$(K_A \lll_{94})_R$
F(Round 17)	k_{17}	$(K_L \lll_{111})_L$
F(Round 18)	k_{18}	$(K_L \lll_{111})_R$
Postwhitening	kw_3	$(K_A \lll_{111})_L$
	kw_4	$(K_A \lll_{111})_R$

Table 55. Extracting Round Keys from User Key and Encrypted Keys for 192/256-bit Key Versions of Camellia (shown for encryption only)

	round key	value
Prewhitening	kw_1	$(K_L \lll_0)_L$
	kw_2	$(K_L \lll_0)_R$
F(Round 1)	k_1	$(K_B \lll_0)_L$
F(Round 2)	k_2	$(K_B \lll_0)_R$
F(Round 3)	k_3	$(K_R \lll_{15})_L$
F(Round 4)	k_4	$(K_R \lll_{15})_R$
F(Round 5)	k_5	$(K_A \lll_{15})_L$
F(Round 6)	k_6	$(K_A \lll_{15})_R$
FL	kl_1	$(K_R \lll_{30})_L$
FL^{-1}	kl_2	$(K_R \lll_{30})_R$
F(Round 7)	k_7	$(K_B \lll_{30})_L$
F(Round 8)	k_8	$(K_B \lll_{30})_R$
F(Round 9)	k_9	$(K_L \lll_{45})_L$
F(Round 10)	k_{10}	$(K_L \lll_{45})_R$
F(Round 11)	k_{11}	$(K_A \lll_{45})_L$
F(Round 12)	k_{12}	$(K_A \lll_{45})_R$
FL	kl_3	$(K_L \lll_{60})_L$
FL^{-1}	kl_4	$(K_L \lll_{60})_R$
F(Round 13)	k_{13}	$(K_R \lll_{60})_L$
F(Round 14)	k_{14}	$(K_R \lll_{60})_R$
F(Round 15)	k_{15}	$(K_B \lll_{60})_L$
F(Round 16)	k_{16}	$(K_B \lll_{60})_R$
F(Round 17)	k_{17}	$(K_L \lll_{77})_L$
F(Round 18)	k_{18}	$(K_L \lll_{77})_R$
FL	kl_5	$(K_A \lll_{77})_L$
FL^{-1}	kl_6	$(K_A \lll_{77})_R$
F(Round 19)	k_{19}	$(K_R \lll_{94})_L$
F(Round 20)	k_{20}	$(K_R \lll_{94})_R$
F(Round 21)	k_{21}	$(K_A \lll_{94})_L$
F(Round 22)	k_{22}	$(K_A \lll_{94})_R$
F(Round 23)	k_{23}	$(K_L \lll_{111})_L$
F(Round 24)	k_{24}	$(K_L \lll_{111})_R$
Postwhitening	kw_3	$(K_B \lll_{111})_L$
	kw_4	$(K_B \lll_{111})_R$

and where the 8-bit output, y, is partitioned into 1-bit segments where,

$$y = y_1 \| y_2 \| y_3 \| y_4 \| y_5 \| y_6 \| y_7 \| y_8$$

We now present the Galois/Affine construction of $s1, s2, s3$, and $s4$. The subsequent section gives pseudo-code for Camellia and also includes an alternative description of the four S-boxes as permutations over the integers $\{0, 1, \ldots, 254, 255\}$ which is useful for software implementations using lookup tables.

3.2.9.1 S-Box s1

$$s1: \qquad y = \mathbf{h}(\mathbf{g}(\mathbf{f}(0xc5 \oplus x))) \oplus 0x6e$$

where $\mathbf{f}$, $\mathbf{g}$, and $\mathbf{h}$ take 8-bit inputs, $a = a_1 \| a_2 \| a_3 \| a_4 \| a_5 \| a_6 \| a_7 \| a_8$, and output 8-bit values, $b = b_1 \| b_2 \| b_3 \| b_4 \| b_5 \| b_6 \| b_7 \| b_8$, where the a_i and b_i are 1-bit values. $\mathbf{f}$ is an affine permutation of the input, $\mathbf{g}$ is inversion over GF(2^8), and $\mathbf{h}$ is an affine transformation of the output. Specifically,

$$\begin{array}{l} \mathbf{f}: \\ b_1 = a_6 \oplus a_2 \\ b_2 = a_7 \oplus a_1 \\ b_3 = a_8 \oplus a_5 \oplus a_3 \\ b_4 = a_8 \oplus a_3 \\ b_5 = a_7 \oplus a_4 \\ b_6 = a_5 \oplus a_2 \\ b_7 = a_8 \oplus a_1 \\ b_8 = a_6 \oplus a_4 \end{array}$$

$$\begin{array}{l} \mathbf{g}: \\ (b_8 + b_7\alpha + b_6\alpha^2 + b_5\alpha^3) + (b_4 + b_3\alpha + b_2\alpha^2 + b_1\alpha^3)\beta \\ = 1/((a_8 + a_7\alpha + a_6\alpha^2 + a_5\alpha^3) + (a_4 + a_3\alpha + a_2\alpha^2 + a_1\alpha^3)\beta) \end{array}$$

$\mathbf{g}$ is an inversion in GF(2^8) assuming $\frac{1}{0}$ is 0, where β is an element in GF(2^8) that satisfies $\beta^8 + \beta^6 + \beta^5 + \beta^3 + 1 = 0$ and $\alpha = \beta^{238} = \beta^6 + \beta^5 + \beta^3 + \beta^2$ is an element in GF(2^4) that satisfies $\alpha^4 + \alpha + 1 = 0$.

$$\begin{array}{l} \mathbf{h}: \\ b_1 = a_5 \oplus a_6 \oplus a_2 \\ b_2 = a_6 \oplus a_2 \\ b_3 = a_7 \oplus a_4 \\ b_4 = a_8 \oplus a_2 \\ b_5 = a_7 \oplus a_3 \\ b_6 = a_8 \oplus a_1 \\ b_7 = a_5 \oplus a_1 \\ b_8 = a_6 \oplus a_3 \end{array}$$

3.2.9.2 S-Box s2

$$s2: \qquad y = s1(x) \lll_1$$

3.2.9.3 S-Box s3

$$s3: \quad y = s1(x) \ggg_1$$

3.2.9.4 S-Box s4

$$s4: \quad y = s1(x \lll_1)$$

3.2.10 Pseudo-Code

```
/* data in hexadecimal */
sigma = [0xa09e667f3bcc908b,
         0xb67ae8584caa73b2,
         0xc6ef372fe94f82be,
         0x54ff53a5f1d36f1c,
         0x10e527fade682d1d,
         0xb05688c2b3e6c1fd]

MAIN    /* an example encryption then decryption for 128-bit key */
/* produces ciphertext = 67673138549669730857065648eabe43 */
{
        K = 0x0123456789abcdeffedcba9876543210;
        plaintext = 0x0123456789abcdeffedcba9876543210;
        Keyschedule(K,k,kl,kw,128);
        ciphertext <- Encrypt(plaintext,k,kl,kw,128);
        plaintext <- Decrypt(ciphertext,k,kl,kw,128);
        /* plaintext recovered */
} /* MAIN */

Keyschedule(K,k,kl,kw,keysize)
{
        KL = K{1,128};
        if (keysize is equal to 128)
        {
                KR = ();
                KRL = 0x0000000000000000;
                KRR = 0x0000000000000000;
        }
        else
        {
                if (keysize is equal to 192)
                        KR = K{129,192} || BAR(K{129,192});
                else
                        KR = K{129,256};
                KRL = KR{1,64};
                KRR = KR{65,128};
        }
        KLL = KL{1,64};
        KLR = KL{65,128};
        L = XOR(KLL,KRL);
        R = XOR(KLR,KRR);
        for n = 1 to 2 do
        {
                R = XOR(R,F(L,sigma[n]));
                SWAP(L,R);
        }
        L = XOR(L,KLL);
        R = XOR(R,KLR);
        for n = 1 to 2 do
        {
                R = XOR(R,F(L,sigma[2+n]));
                SWAP(L,R);
        }
        KA = L || R;
        L = XOR(L,KRL);
        R = XOR(R,KRR);
        for n = 1 to 2 do
        {
                R = XOR(R,F(L,sigma[4+n]));
                SWAP(L,R);
        }
        KB = L || R;
        kw[1] = KL{1,64};
        kw[2] = KL{65,128};
        if (keysize = 128)
        {
                k[1] = KA{1,64};
                k[2] = KA{65,128};
                k[3] = (KL <<< 15){1,64};
                k[4] = (KL <<< 15){65,128};
                k[5] = (KA <<< 15){1,64};
                k[6] = (KA <<< 15){65,128};
                k[7] = (KL <<< 45){1,64};
                k[8] = (KL <<< 45){65,128};
                k[9] = (KA <<< 45){1,64};
                k[10] = (KL <<< 60){65,128};
                k[11] = (KA <<< 60){1,64};
                k[12] = (KA <<< 60){65,128};
                k[13] = (KL <<< 94){1,64};
                k[14] = (KL <<< 94){65,128};
                k[15] = (KA <<< 94){1,64};
                k[16] = (KA <<< 94){65,128};
                k[17] = (KL <<< 111){1,64};
                k[18] = (KL <<< 111){65,128};

                kl[1] = (KA <<< 30){1,64};
                kl[2] = (KA <<< 30){65,128};
                kl[3] = (KL <<< 77){1,64};
                kl[4] = (KL <<< 77){65,128};
                kw[3] = (KA <<< 111){1,64};
                kw[4] = (KA <<< 111){65,128};
        }
        else  /* keysize is 192 or 256 */
        {
                k[1] = KB{1,64};
                k[2] = KB{65,128};
                k[3] = (KR <<< 15){1,64};
                k[4] = (KR <<< 15){65,128};
                k[5] = (KA <<< 15){1,64};
```

```
                k[6] = (KA <<< 15){65,128};
                k[7] = (KB <<< 30){1,64};
                k[8] = (KB <<< 30){65,128};
                k[9] = (KL <<< 45){1,64};
                k[10] = (KL <<< 45){65,128};
                k[11] = (KA <<< 45){1,64};
                k[12] = (KA <<< 45){65,128};
                k[13] = (KR <<< 60){1,64};
                k[14] = (KR <<< 60){65,128};
                k[15] = (KB <<< 60){1,64};
                k[16] = (KB <<< 60){65,128};
                k[17] = (KL <<< 77){1,64};
                k[18] = (KL <<< 77){65,128};
                k[19] = (KR <<< 94){1,64};
                k[20] = (KR <<< 94){65,128};
                k[21] = (KA <<< 94){1,64};
                k[22] = (KA <<< 94){65,128};
                k[23] = (KL <<< 111){1,64};
                k[24] = (KL <<< 111){65,128};

                kl[1] = (KR <<< 30){1,64};
                kl[2] = (KR <<< 30){65,128};
                kl[3] = (KL <<< 60){1,64};
                kl[4] = (KL <<< 60){65,128};
                kl[5] = (KA <<< 77){1,64};
                kl[6] = (KA <<< 77){65,128};
                kw[3] = (KB <<< 111){1,64};
                kw[4] = (KB <<< 111){65,128};
        }
} /* Keyschedule */

128Bits <- Encrypt(M,k,kl,kw,keysize)
{
        L = M{1,64};
        R = M{65,128};
        L = XOR(L,kw[1]);
        R = XOR(R,kw[2]);
        if (keysize is equal to 128)
                BigRounds = 3;
        else
                BigRounds = 4;
        for N = 0 to (BigRounds - 1) do
        {
                for n = 1 to 6 do
                {
                        R = XOR(R,F(L,k[n + (6 * N)]));
                        SWAP(L,R);
                }
                if (N is not equal to (BigRounds - 1))
                {
                        L = FL(L,kl[(2 * N) + 1]);
                        R = FLINV(R,kl[(2 * N) + 2]);
                }
        }
        SWAP(L,R);
        L = XOR(L,kw[3]);
        R = XOR(R,kw[4]);
        Return(L || R);
} /* Encrypt */

128Bits <- Decrypt(C,k,kl,kw,keysize)
{
        R = C{1,64};
        L = C{65,128};
        R = XOR(R,kw[3]);
        L = XOR(L,kw[4]);
        if (keysize is equal to 128)
                BigRounds = 3;
        else
                BigRounds = 4;
        for N = (BigRounds - 1) downto 0 do
        {
                if (N is not equal to (BigRounds - 1))
                {
                        R = FL(R,kl[(2 * N) + 2)]);
                        L = FLINV(L,kl[(2 * N) + 1)]);
                }
                for n = 6 downto 1 do
                {
                        L = XOR(L,F(R,k[n + (6 * N)]));
                        SWAP(L,R);
                }
        }
        SWAP(L,R);
        R = XOR(R,kw[1]);
        L = XOR(L,kw[2]);
        Return(R || L);
} /* Decrypt */

64Bits <- F(x,ki)
{
        x = XOR(x,ki);
        z1 = s1[x{1,8}];
        z2 = s2[x{9,16}];
        z3 = s3[x{17,24}];
        z4 = s4[x{25,32}];
        z5 = s2[x{33,40}];
        z6 = s3[x{41,48}];
        z7 = s4[x{49,56}];
        z8 = s1[x{57,64}];
        z1' = XOR(z1,z3,z4,z6,z7,z8);
        z2' = XOR(z1,z2,z4,z5,z7,z8);
        z3' = XOR(z1,z2,z3,z5,z6,z8);
        z4' = XOR(z2,z3,z4,z5,z6,z7);
        z5' = XOR(z1,z2,z6,z7,z8);
        z6' = XOR(z2,z3,z5,z7,z8);
        z7' = XOR(z3,z4,z5,z6,z8);
        z8' = XOR(z1,z4,z5,z6,z7);
        Return(z1' || z2' || z3' || z4' || z5' || z6' || z7' || z8');
} /* F */
```

```
64Bits <- FL(X,kli)
{
        XL = X{1,32};
        XR = X{33,64};
        XR = XOR(XR,AND(XL,kli{1,32}) <<< 1);
        XL = XOR(XL,OR(XR,kli{33,64}));
        Return(XL || XR);
} /* FL */

64Bits <- FLINV(X,kli)
{
        XL = X{1,32};
        XR = X{33,64};
        XL = XOR(XL,OR(XR,kli{33,64}));
        XR = XOR(XR,AND(XL,kli{1,32}) <<< 1);
        Return(XL || XR);
} /* FLINV */

/* s1 is here specified in decimal */
s1 = [112,130, 44,236,179, 39,192,229,228,133, 87, 53,234, 12,174, 65,
       35,239,107,147, 69, 25,165, 33,237, 14, 79, 78, 29,101,146,189,
      134,184,175,143,124,235, 31,206, 62, 48,220, 95, 94,197, 11, 26,
      166,225, 57,202,213, 71, 93, 61,217,  1, 90,214, 81, 86,108, 77,
      139, 13,154,102,251,204,176, 45,116, 18, 43, 32,240,177,132,153,
      223, 76,203,194, 52,126,118,  5,109,183,169, 49,209, 23,  4,215,
       20, 88, 58, 97,222, 27, 17, 28, 50, 15,156, 22, 83, 24,242, 34,
      254, 68,207,178,195,181,122,145, 36,  8,232,168, 96,252,105, 80,
      170,208,160,125,161,137, 98,151, 84, 91, 30,149,224,255,100,210,
       16,196,  0, 72,163,247,117,219,138,  3,230,218,  9, 63,221,148,
      135, 92,131,  2,205, 74,144, 51,115,103,246,243,157,127,191,226,
       82,155,216, 38,200, 55,198, 59,129,150,111, 75, 19,190, 99, 46,
      233,121,167,140,159,110,188,142, 41,245,249,182, 47,253,180, 89,
      120,152,  6,106,231, 70,113,186,212, 37,171, 66,136,162,141,250,
      114,  7,185, 85,248,238,172, 10, 54, 73, 42,104, 60, 56,241,164,
       64, 40,211,123,187,201, 67,193, 21,227,173,244,119,199,128,158]

/* s2 is here specified in decimal */
s2 = [224,  5, 88,217,103, 78,129,203,201, 11,174,106,213, 24, 93,130,
       70,223,214, 39,138, 50, 75, 66,219, 28,158,156, 58,202, 37,123,
       13,113, 95, 31,248,215, 62,157,124, 96,185,190,188,139, 22, 52,
       77,195,114,149,171,142,186,122,179,  2,180,173,162,172,216,154,
       23, 26, 53,204,247,153, 97, 90,232, 36, 86, 64,225, 99,  9, 51,
      191,152,151,133,104,252,236, 10,218,111, 83, 98,163, 46,  8,175,
       40,176,116,194,189, 54, 34, 56,100, 30, 57, 44,166, 48,229, 68,
      253,136,159,101,135,107,244, 35, 72, 16,209, 81,192,249,210,160,
       85,161, 65,250, 67, 19,196, 47,168,182, 60, 43,193,255,200,165,
       32,137,  0,144, 71,239,234,183, 21,  6,205,181, 18,126,187, 41,
       15,184,  7,  4,155,148, 33,102,230,206,237,231, 59,254,127,197,
      164, 55,177, 76,145,110,141,118,  3, 45,222,150, 38,125,198, 92,
      211,242, 79, 25, 63,220,121, 29, 82,235,243,109, 94,251,105,178,
      240, 49, 12,212,207,140,226,117,169, 74, 87,132, 17, 69, 27,245,
      228, 14,115,170,241,221, 89, 20,108,146, 84,208,120,112,227, 73,
      128, 80,167,246,119,147,134,131, 42,199, 91,233,238,143,  1, 61]
```

```
/* s3 is here specified in decimal */
s3 = [ 56, 65, 22,118,217,147, 96,242,114,194,171,154,117,  6, 87,160,
      145,247,181,201,162,140,210,144,246,  7,167, 39,142,178, 73,222,
       67, 92,215,199, 62,245,143,103, 31, 24,110,175, 47,226,133, 13,
       83,240,156,101,234,163,174,158,236,128, 45,107,168, 43, 54,166,
      197,134, 77, 51,253,102, 88,150, 58,  9,149, 16,120,216, 66,204,
      239, 38,229, 97, 26, 63, 59,130,182,219,212,152,232,139,  2,235,
       10, 44, 29,176,111,141,136, 14, 25,135, 78, 11,169, 12,121, 17,
      127, 34,231, 89,225,218, 61,200, 18,  4,116, 84, 48,126,180, 40,
       85,104, 80,190,208,196, 49,203, 42,173, 15,202,112,255, 50,105,
        8, 98,  0, 36,209,251,186,237, 69,129,115,109,132,159,238, 74,
      195, 46,193,  1,230, 37, 72,153,185,179,123,249,206,191,223,113,
       41,205,108, 19,100,155, 99,157,192, 75,183,165,137, 95,177, 23,
      244,188,211, 70,207, 55, 94, 71,148,250,252, 91,151,254, 90,172,
       60, 76,  3, 53,243, 35,184, 93,106,146,213, 33, 68, 81,198,125,
       57,131,220,170,124,119, 86,  5, 27,164, 21, 52, 30, 28,248, 82,
       32, 20,233,189,221,228,161,224,138,241,214,122,187,227, 64, 79]

/* s4 is here specified in decimal */
s4 = [112, 44,179,192,228, 87,234,174, 35,107, 69,165,237, 79, 29,146,
      134,175,124, 31, 62,220, 94, 11,166, 57,213, 93,217, 90, 81,108,
      139,154,251,176,116, 43,240,132,223,203, 52,118,109,169,209,  4,
       20, 58,222, 17, 50,156, 83,242,254,207,195,122, 36,232, 96,105,
      170,160,161, 98, 84, 30,224,100, 16,  0,163,117,138,230,  9,221,
      135,131,205,144,115,246,157,191, 82,216,200,198,129,111, 19, 99,
      233,167,159,188, 41,249, 47,180,120,  6,231,113,212,171,136,141,
      114,185,248,172, 54, 42, 60,241, 64,211,187, 67, 21,173,119,128,
      130,236, 39,229,133, 53, 12, 65,239,147, 25, 33, 14, 78,101,189,
      184,143,235,206, 48, 95,197, 26,225,202, 71, 61,  1,214, 86, 77,
       13,102,204, 45, 18, 32,177,153, 76,194,126,  5,183, 49, 23,215,
       88, 97, 27, 28, 15, 22, 24, 34, 68,178,181,145,  8,168,252, 80,
      208,125,137,151, 91,149,255,210,196, 72,247,219,  3,218, 63,148,
       92,  2, 74, 51,103,243,127,226,155, 38, 55, 59,150, 75,190, 46,
      121,140,110,142,245,182,253, 89,152,106, 70,186, 37, 66,162,250,
        7, 85,238, 10, 73,104, 56,164, 40,123,201,193,227,244,199,158]

Data, Logical and Arithmetic Definitions
=========================================


             0x1fa4....
A number of this type, with 0x at the beginning, should be interpreted as
hexadecimal. The above example represents the bit string 0001111110100100.

             a || b || c ||..
outputs the concatenation of the bitstrings, a,b,c,.... with a leftmost.
Example:  a = 1011, b = 0110 -> a || b = 10110110

             XOR(a,b,...)
outputs the bitwise XOR of a with b with .....
Example:  a = 1011, b = 0110 -> XOR(a,b) = 1101

             OR(a,b)
outputs the bitwise OR of a and b.
Example:  a = 1011, b = 0110 -> OR(a,b) = 1111
```

```
             BAR(a)
outputs the bitwise negation of a.
Example:  a = 1011 -> BAR(a) = 0100

             AND(a,b)
outputs the bitwise AND of a and b.
Example:  a = 1011, b = 0110 -> AND(a,b) = 0010

             a <<< b
outputs the cyclic rotation left of a by b bits
Example:  a = 1011, b = 2 -> a <<< b = 1110

             SWAP(a,b)
swaps the contents of a and b.
Example:  a = 1011, b = 0110 -> SWAP(a,b) -> a = 0110, b = 1011

             A{a,b}
outputs the bit-segment of the bit string, A, starting from bit a of A
and up to and including bit b of A. Thus, A{a,b} has bit-length b - a + 1.
The first (leftmost) bit in A is A{1}.
Example:  A = 10110110, -> A{4,6} = 101

             A[a]
outputs the integer element stored at position a in the integer array, A.
The first integer in A is A[1].
Example:  A = [112,130,44,236,179,....], -> A[3] = 44

             ’*’  ’+’
mean integer multiplication and addition, respectively.

List of Variables Used
=======================
128-256 bits        - K
128 bits            - plaintext,ciphertext,KL,KA,KB,M,C
0-128 bits          - KR
64 bits             - KLL,KLR,L,R,sigma[n],kw[i],k[i],kl[i],x,ki,X,kli
0-64 bits           - KRL,KRR
32 bits             - XL,XR
8 bits              - z1,z2,z3,z4,z5,z6,z7,z8
integer             - keysize,n,N,BigRounds
```

3.3 Test vectors

To aid in verification of a software or hardware implementation of Camellia, we here provide a few test vectors comprising, as input (K, plaintext) and, as output from the key schedule and encryption ($KA(\|KB)$, ciphertext). All data in this section is presented in hexadecimal. We provide test vectors for keysizes 128, 192, and 256 separately.

3.3.1 Keysize 128

K	0123456789ABCDEFFEDCBA9876543210$_{x}$	0123456789ABCDEFFEDCBA9876543210$_{x}$	plain
KA	AE71C3D55BA6BF1D169240A795F89256$_{x}$	6767313854966973085706564 8EABE43$_{x}$	cipher
K	4149D2ADED9456681EC8B511D9E7EE04$_{x}$	2A9B0B74F4C5DC6239B7063A50A7946E$_{x}$	plain
KA	C501214A4E3EBDE87C7CB2849487E0AB$_{x}$	DB93BB9C0ADD5AB59ED94D467A6277F8$_{x}$	cipher
K	47E8FB063DD4FE4AB430A73AF7720206$_{x}$	0F9D74FC31CA654F921A606C024E7084$_{x}$	plain
KA	0E295B7D3F360780E68144421220764F$_{x}$	147375F650037166CA66C010CDA256A5$_{x}$	cipher
K	40BC8981241954A60A942B4A4334D1DB$_{x}$	98048DD5D98B1F8DBBC6C7B238C9B948$_{x}$	plain
KA	02596E63AA22CE0B724A216C366FEA38$_{x}$	3804E8E37A934CD71490C04AC34FD01E$_{x}$	cipher
K	3DA93F2679DECB104422E07332F7E3FE$_{x}$	CCA0F0F0BA4596C4D9C10E1CF5DFF82E$_{x}$	plain
KA	83B4F6108365A66DD87F05F25FE79D4B$_{x}$	3AC304208199AA72CCDDC42F5E6C7972$_{x}$	cipher

3.3.2 Keysize 192

K	0123456789ABCDEFFEDCBA98765432100011223344556677$_{x}$
$KA\|\|KB$	0766A2135C44E288CF62016A06BABED3$_{x}$ ‖ 8F3AFAC1CC974396C098A0B7E38B4DF2$_{x}$
plain	0123456789ABCDEFFEDCBA9876543210$_{x}$
cipher	B4993401B3E996F84EE5CEE7D79B09B9$_{x}$
K	5E89B44B505C09F156BF78055F78A83C24BFC19EDD5C94EF$_{x}$
$KA\|\|KB$	BA0B31B670B34C26026DFF7B74564087$_{x}$ ‖ 3C23619D33DD5FCBEE712953EEDA757F$_{x}$
plain	DCAC1785791E9BF611C7C7FCF3BCDFE7$_{x}$
cipher	46DB784FF79A83ADBB9A36D617BF94B2$_{x}$
K	D3E748B043DC9F66388B7D50567CC6AA2F884F3E53E4A3DD$_{x}$
$KA\|\|KB$	49667A715EC02BE735943E7A4B4ACC0C$_{x}$ ‖ 264D533063503300AADF49FE61B451CE$_{x}$
plain	54B3C1A40FCB95658A0D6BEA861326AA$_{x}$
cipher	D01DF1A0F3C44431A7D48ECABC94B25E$_{x}$
K	1E1FE47104884EF696166EB80390ADD8FB53EF43986DC268$_{x}$
$KA\|\|KB$	D976805E61B8B9EC6F7DF42B42C11239$_{x}$ ‖ 50C525B03A7575A6F061F5780EB98D91$_{x}$
plain	D46BA51747457E7FD0FBCF267796D0A6$_{x}$
cipher	069713F8BE95E9E78EB4D312D7178582$_{x}$
K	43E6B5DB547743BD09D19D312F2477AD902E2F8334A70D4F$_{x}$
$KA\|\|KB$	B4A87A3EAF5A4B27645242CF6CFEDE61$_{x}$ ‖ 83ADBB334E4BF72262ADB5E18FDE7873$_{x}$
plain	7E6BB782F305788A1BE6421F76F0F772$_{x}$
cipher	9C7C8C7148B6FAE78FBA6576D6808E92$_{x}$

3.3.3 Keysize 256

K	0123456789ABCDEFFEDCBA98765432100011223344556677889 9AABBCCDDEEFF$_x$
$KA\|KB$	17D1B5B046DF07FAC9BB914B7F1937EE$_x$‖3815214280A4D3C01848FD9AC7B1FE60$_x$
plain	0123456789ABCDEFFEDCBA9876543210$_x$
cipher	9ACC237DFF16D76C20EF7C919E3A7509$_x$
K	C940117C2EDA1D1EEA32C009D3C85421B330D6547F0D36E7AA6A2BE16D584636$_x$
$KA\|KB$	E9E4BF7F4699FC102DC4E604048338D2$_x$‖5C8D2902D7ED8F582423E290493DBB3B$_x$
plain	4DEADCB5A14F37E2679C344437032D64$_x$
cipher	96379E8CC8ECCDEE43C9A5332CE5627E$_x$
K	A8CD7528DAAB0F84153A668392ACB92A036CF1343DD64F3F7C7415EAEC0C0B95$_x$
$KA\|KB$	1013CF0907021375C6A47660CA372337$_x$‖860596FDBAC808D0B66E385332BC805D$_x$
plain	9857B3C731D0E51B02A524D66E78F721$_x$
cipher	5559E464CF71C284C2279A6BDDD8FA71$_x$
K	E9B481268AD16606457BF03188FBC6617B8315A64F4EE755ECAAED3727B08411$_x$
$KA\|KB$	8DFD62ACE469C4C31944C934F6D819CD$_x$‖E7EB9972CA9768A2F513E0E48CD147CD$_x$
plain	D980BDB42BCC3840069EC3984A7DC24D$_x$
cipher	8A9CD33A905A24A38EB0B4FBF2E7D68F$_x$
K	971803E766EA3C52942A89BCDA0ECD3E14042CEBA22107BB07545CE8685E4400$_x$
$KA\|KB$	A02F5B9A72D9CE7C7085A59258D8C8ED$_x$‖208A93B9DEC678C522574C7DC203BCA1$_x$
plain	640637EED79DF51C19E54DA1E114025C$_x$
cipher	B01EA3099F64847F8B0AD264841C64BD$_x$

It is also useful to verify the test vectors for partial implementations of Camellia as, for instance, when the FL and FL^{-1} modules are removed and replaced by identity transformations, (input directly connected to output). One can similarly replace the F,s1,s2,s3,s4, and P modules with appropriate identity transformations. In the following we provide one set of test vectors for a selected subset of scenarios where some of the above modules have been replaced by identity transformations. Note that, in all possible partial cipher configurations considered below, the action of decryption of the ciphertext (using the same partial configuration) results in the original plaintext. We provide test vectors for keysizes 128, 192, and 256 separately. Note also that for the case 'Implementation includes -', we mean that we replace the F,s1,s2,s3,s4, and P modules with identity transformations. This does NOT mean that these modules are deleted - no links in the design are broken.

3.3.4 Partial Implementations - Keysize 128

Input:	K = 0123456789ABCDEFFEDCBA9876543210$_\mathrm{x}$ plaintext = 0123456789ABCDEFFEDCBA9876543210$_\mathrm{x}$
Implementation includes	KA, ciphertext
–	KA = 0123456789ABCDEFFEDCBA9876543210$_\mathrm{x}$ ciphertext = 19080091A2B3C4D5E6F7FF6E5D4C3B2A$_\mathrm{x}$
FL/FL^{-1}	KA = 0123456789ABCDEFFEDCBA9876543210$_\mathrm{x}$ ciphertext = 60FD2E1373FB84D5B726CCED736A262F$_\mathrm{x}$
F	KA = 33AD4792AAFBB0C68E4965EFD3B1C31C$_\mathrm{x}$ ciphertext = 79AF1B549F685C53575AB532217AA79B$_\mathrm{x}$
F,s1	KA = F0AD4792AAFBB0F9214965EFD3B1C353$_\mathrm{x}$ ciphertext = 8178FADB74B78B6ACF829741D18291F0$_\mathrm{x}$
F,s2	KA = 33FB4792A9FBB0C68E5165EFFEB1C31C$_\mathrm{x}$ ciphertext = 79B09AFF3ABE5DDF5C76423D838F349C$_\mathrm{x}$
F,s3	KA = 33AD8D92AAB7B0C68E4950EFD31FC31C$_\mathrm{x}$ ciphertext = 1CBC8B22FA78B135A8F1A7B33D99B559$_\mathrm{x}$
F,s4	KA = 33AD4783AAFB69C68E4965D6D3B1FF1C$_\mathrm{x}$ ciphertext = 27A7AD0663E0C69D60EEBA55D3E15DCF$_\mathrm{x}$
F,P	KA = A1D7C1ADC77B3DA683474E5DA3344963$_\mathrm{x}$ ciphertext = 11A8E71280EFDAE6D3918AE3190BAA89$_\mathrm{x}$
FL/FL^{-1},F	KA = 33AD4792AAFBB0C68E4965EFD3B1C31C$_\mathrm{x}$ ciphertext = BEBB914476211FD2462B112452D4B6AC$_\mathrm{x}$
FL/FL^{-1},F,s1,s2,s3,s4	KA = F0FB8D83A9B769F9215150D6FE1FFF53$_\mathrm{x}$ ciphertext = 63268A2F8951DF32C9F961704443C87B$_\mathrm{x}$
FL/FL^{-1},F,s1,s2,s3,P	KA = B82919201A0BF2C69084CB4E71A64A7C$_\mathrm{x}$ ciphertext = 731FB3C2F5AD22BDA69E0AD78C0395ED$_\mathrm{x}$
FL/FL^{-1},F,s1,s2,s4,P	KA = AB017974AEAB5AC63F91169BBCC8710D$_\mathrm{x}$ ciphertext = EEFD50D90CC2F0197D63A131AB958847$_\mathrm{x}$
FL/FL^{-1},F,s1,s3,s4,P	KA = 8A00B3141C260E69724B7B0264EE5BB9$_\mathrm{x}$ ciphertext = B8BCA5B2D9481669D14FDBA2E3B3366E$_\mathrm{x}$
FL/FL^{-1},F,s2,s3,s4,P	KA = E2E1172D9353B7F3154631466CFD8EB1$_\mathrm{x}$ ciphertext = A0CF747DF45B2318BEC0DAC91DCCE888$_\mathrm{x}$
F,s1,s2,s3,s4,P	KA = AE71C3D55BA6BF1D169240A795F89256$_\mathrm{x}$ ciphertext = BB7BFEF4A3C852F56DE4FEB946BB89C0$_\mathrm{x}$
FL/FL^{-1},F,s1,s2,s3,s4,P (complete)	KA = AE71C3D55BA6BF1D169240A795F89256$_\mathrm{x}$ ciphertext = 676731385496697308570656 48EABE43$_\mathrm{x}$

3.3.5 Partial Implementations - Keysize 192

Input:	$K = \texttt{0123456789ABCDEFFEDCBA98765432100011223344556677}_{\mathrm{x}}$ plaintext $= \texttt{0123456789ABCDEFFEDCBA9876543210}_{\mathrm{x}}$
Implementation includes	$KA, KB,$ ciphertext
–	$KA = \texttt{FEDCBA9876543210FECD98AB32015467}_{\mathrm{x}}$ $KB = \texttt{0123456789ABCDEFFFEEDDCCBBAA9988}_{\mathrm{x}}$ ciphertext $= \texttt{4CC40091A2B3C4D5E6F7FFF76EE65DD5}_{\mathrm{x}}$
FL/FL^{-1}	$KA = \texttt{FEDCBA9876543210FECD98AB32015467}_{\mathrm{x}}$ $KB = \texttt{0123456789ABCDEFFFEEDDCCBBAA9988}_{\mathrm{x}}$ ciphertext $= \texttt{B23C8B3E8E37B9641924F03C913D9B96}_{\mathrm{x}}$
F	$KA = \texttt{CC52B86D55044F398E5847DC97E4A56B}_{\mathrm{x}}$ $KB = \texttt{D105352841C0D003AD1027B4E37738B0}_{\mathrm{x}}$ ciphertext $= \texttt{239F084D2F52F01F39C6AB4FBCC6D0A1}_{\mathrm{x}}$
F,s1	$KA = \texttt{1C52B86D55044F991C5847DC97E4A5CC}_{\mathrm{x}}$ $KB = \texttt{8A05352841C0D049091027B4E377383F}_{\mathrm{x}}$ ciphertext $= \texttt{CD7CB7947442B965A9A311059E057138}_{\mathrm{x}}$
F,s2	$KA = \texttt{CC5DB86D6C044F398E6047DC1FE4A56B}_{\mathrm{x}}$ $KB = \texttt{D1073528A1C0D003AD4027B4227738B0}_{\mathrm{x}}$ ciphertext $= \texttt{A52134CB16481103ABB2F3E127D5E61A}_{\mathrm{x}}$
F,s3	$KA = \texttt{CC520E6D559C4F398E5850DC9704A56B}_{\mathrm{x}}$ $KB = \texttt{D10566284141D003AD1017B4E38038B0}_{\mathrm{x}}$ ciphertext $= \texttt{F8A4F331047BFD43E9439073B0529995}_{\mathrm{x}}$
F,s4	$KA = \texttt{CC52B88455043A398E58472397E43C6B}_{\mathrm{x}}$ $KB = \texttt{D105356C41C01703AD102709E3771CB0}_{\mathrm{x}}$ ciphertext $= \texttt{3F4BAD8A55EF8C6EC7CD84A11A9A31CA}_{\mathrm{x}}$
F,P	$KA = \texttt{E582A7DA0B95F148E521283BF6523E27}_{\mathrm{x}}$ $KB = \texttt{358AA59DA93118D26E56CFB3FE250DCA}_{\mathrm{x}}$ ciphertext $= \texttt{769488327945E19BB78AE57D92378A5D}_{\mathrm{x}}$
FL/FL^{-1},F	$KA = \texttt{CC52B86D55044F398E5847DC97E4A56B}_{\mathrm{x}}$ $KB = \texttt{D105352841C0D003AD1027B4E37738B0}_{\mathrm{x}}$ ciphertext $= \texttt{1F656CC60334CD5FC679E84FEF6CC13A}_{\mathrm{x}}$
FL/FL^{-1},F,s1,s2,s3,s4	$KA = \texttt{1C5D0E846C9C3A991C6050231F043CCC}_{\mathrm{x}}$ $KB = \texttt{8A07666CA141174909401709228O1C3F}_{\mathrm{x}}$ ciphertext $= \texttt{A1D617F74B5772A9B192F139B9A0E04D}_{\mathrm{x}}$
FL/FL^{-1},F,s1,s2,s3,P	$KA = \texttt{0B624512C0B8F7848A636A3E188BEA31}_{\mathrm{x}}$ $KB = \texttt{D394E83B811432C6ACECB02D8E100B07}_{\mathrm{x}}$ ciphertext $= \texttt{C33EC891F38AC7E20CEBC53A0139E9D1}_{\mathrm{x}}$
FL/FL^{-1},F,s1,s2,s4,P	$KA = \texttt{2B691944455ED09640D25203297A7CD2}_{\mathrm{x}}$ $KB = \texttt{B1C13857457ABA851265DA48130D5669}_{\mathrm{x}}$ ciphertext $= \texttt{A8244AB4EC460EB1ACBDFB02A85041E1}_{\mathrm{x}}$
FL/FL^{-1},F,s1,s3,s4,P	$KA = \texttt{1CE4B02A1DFF66116086225F2345B610}_{\mathrm{x}}$ $KB = \texttt{E70DCA76B9A97AFCB29A626BB0E2C601}_{\mathrm{x}}$ ciphertext $= \texttt{40CB2D4629C0B493F4095828A41151C1}_{\mathrm{x}}$
FL/FL^{-1},F,s2,s3,s4,P	$KA = \texttt{D54E15652A66B668427FB9B97F945CFE}_{\mathrm{x}}$ $KB = \texttt{7D2C5D1533377968306FD045B123DDCC}_{\mathrm{x}}$ ciphertext $= \texttt{B52974CDE868248E203346F2BFB46593}_{\mathrm{x}}$
F,s1,s2,s3,s4,P	$KA = \texttt{0766A2135C44E288CF62016A06BABED3}_{\mathrm{x}}$ $KB = \texttt{8F3AFAC1CC974396C098A0B7E38B4DF2}_{\mathrm{x}}$ ciphertext $= \texttt{9E7E989406CFA12DEA3A5F2D76A35E13}_{\mathrm{x}}$
FL/FL^{-1},F,s1,s2,s3,s4,P (complete)	$KA = \texttt{0766A2135C44E288CF62016A06BABED3}_{\mathrm{x}}$ $KB = \texttt{8F3AFAC1CC974396C098A0B7E38B4DF2}_{\mathrm{x}}$ ciphertext $= \texttt{B4993401B3E996F84EE5CEE7D79B09B9}_{\mathrm{x}}$

3.3.6 Partial Implementations - Keysize 256

Input:	K = 0123456789ABCDEFFEDCBA9876543210$_x$ $|$00112233445566778899AABBCCDDEEFF$_x$ plaintext = 0123456789ABCDEFFEDCBA9876543210$_x$
Implementation includes	$KA, KB,$ ciphertext
–	KA = 89ABCDEF01234567FECD98AB32015467$_x$ KB = 76543210FEDCBA98FFEEDDCCBBAA9988$_x$ ciphertext = 4CC43B2A19087F6E5D4C7FF76EE65DD5$_x$
FL/FL^{-1}	KA = 89ABCDEF01234567FECD98AB32015467$_x$ KB = 76543210FEDCBA98FFEEDDCCBBAA9988$_x$ ciphertext = 9CF631843944869BD6C93FDDE8AC8004$_x$
F	KA = BB25CF1A2273384E8E5847DC97E4A56B$_x$ KB = A672425F36B7A774AD1027B4E37738B0$_x$ ciphertext = 17DB795C239E3CD3D7287E1AF44E6392$_x$
F,s1	KA = 2C25CF1A227338238D5847DC97E4A590$_x$ KB = D472425F36B7A7FE541027B4E377387D$_x$ ciphertext = D790603681A8BD2A27EFD0174D452960$_x$
F,s2	KA = BB14CF1ADE73384E8EA447DCF5E4A56B$_x$ KB = A6A0425F63B7A774ADD927B4CE7738B0$_x$ ciphertext = B6B7272A23F7DA6D44F5A167D3A6A6B4$_x$
F,s3	KA = BB25DE1A2293384E8E58E9DC9747A56B$_x$ KB = A672E95F36C7A774AD1062B4E34538B0$_x$ ciphertext = 1F73AD9C63CE2CFFE88CE16B41FF6D03$_x$
F,s4	KA = BB25CF542273924E8E58475D97E49F6B$_x$ KB = A672428936B73174AD102730E37757B0$_x$ ciphertext = 353C6D4016E53487CE4FEB3BFC150E44$_x$
F,P	KA = E582A7DA7CE2863FE521283BF6523E27$_x$ KB = 358AA59DDE466FA56E56CFB3FE250DCA$_x$ ciphertext = 271B47C303BAC0CF6149531EB4116295$_x$
FL/FL^{-1},F	KA = BB25CF1A2273384E8E5847DC97E4A56B$_x$ KB = A672425F36B7A774AD1027B4E37738B0$_x$ ciphertext = 74634A8596D420CA2B962CE62DEAD1B8$_x$
FL/FL^{-1},F,s1,s2,s3,s4	KA = 2C14DE54DE9392238DA4E95DF5479F90$_x$ KB = D4A0E98963C731FE54D96230CE45577D$_x$ ciphertext = D52A41E60DE3159FC0348C955908176B$_x$
FL/FL^{-1},F,s1,s2,s3,P	KA = F4CCBD694FC3B106773031528896ACC6$_x$ KB = 54CAB25F54D19B298E678F0CE61DBD57$_x$ ciphertext = 1CB480CF818A36393B581F12353FA7B8$_x$
FL/FL^{-1},F,s1,s2,s4,P	KA = 11BE956EA68AE5EB44E6A807691CF633$_x$ KB = 5BDE250A16823352375783E10162D4D5$_x$ ciphertext = 763AFDABFCB1049BA1EDA1D6B5D6FF6F$_x$
FL/FL^{-1},F,s1,s3,s4,P	KA = E7D8411C2D79CFEB32AEAABDF0869FB8$_x$ KB = 491425E38734A1C3DDCB8AEB7968E5BA$_x$ ciphertext = 3D7F37B736A081619F4853F4B56AFB99$_x$
FL/FL^{-1},F,s2,s3,s4,P	KA = 776E0ACCD74AB3DC3E6AD0D51F60F31F$_x$ KB = BA18ECDA611399AD70F36DD72F2DF753$_x$ ciphertext = 25D95763D39C04F405372AF83FF60B05$_x$
F,s1,s2,s3,s4,P	KA = 17D1B5B046DF07FAC9BB914B7F1937EE$_x$ KB = 3815214280A4D3C01848FD9AC7B1FE60$_x$ ciphertext = C93862015661CD9115B05BAA43245E36$_x$
FL/FL^{-1},F,s1,s2,s3,s4,P (complete)	KA = 17D1B5B046DF07FAC9BB914B7F1937EE$_x$ KB = 3815214280A4D3C01848FD9AC7B1FE60$_x$ ciphertext = 9ACC237DFF16D76C20EF7C919E3A7509$_x$

4. SHACAL-2

4.1 Introduction

4.1.1 Overview

SHA-1 [472] has been a NIST hash function standard (FIPS-180-1) since 1995. It has been subjected to a great deal of cryptanalytic effort, and only recently a weakness of SHA-1 was reported in [553]. This inspired the use of the compression function of SHA-1 in encryption mode [280]. This primitive was submitted to NESSIE as the candidate SHACAL [281], which is later referred to as SHACAL-1 (as it based on SHA-1).

In August 2002 NIST has added to FIPS-180 three new hash functions: SHA-256, SHA-384 and SHA-512. These inspired the submitters of SHACAL-1 to tweak their submission, and add SHACAL-2, a block cipher based on the compression function of SHA-256.

SHACAL-2 is a 256-bit block cipher which accepts keys of various lengths (between 0 and 512 bits). The cipher is well suited for applications where SHA-256 is already implemented (either by software or hardware), reducing the code size or the gate count of the final application.

4.1.2 Outline of the primitive

SHACAL-2 operates over 64 rounds on 256-bit plaintext. It supports key sizes between 0 and 512 bits, although a minimum of 128 bits is strongly recommended. It uses a composition of bitwise logical operations, addition modulo 2^{32} and two non-linear functions – the majority and the selection functions. The key schedule of SHACAL-2 is linear in nature. Keys shorter than 512 bits are padded into 512 bits, and the 512-bit key is then used to compute 2048 subkey bits.

Decryption is similar to encryption up to the order of subkeys, and the use of subtraction modulo 2^{32} instead of addition in two places.

4.1.3 Security and performance

No security flaws have been identified for SHACAL-2. The weakness in the key schedule of SHACAL-1 does not extend to SHACAL-2.

SHACAL-2 is quite efficient both in software and in hardware. The cipher is relatively fast (between 30 and 60 cycles per byte), and very well suited for usage along with SHA-256.

4.2 Description

This section intends to provide a short but complete specification of the SHACAL-2 algorithm. A more extensive description can be found in NIST's FIPS-180-2 [472] (detailing SHA-256) and in the description of SHACAL-2 [281].

4.2.1 Cipher

The SHACAL-2 encryption algorithm consists of a sequence of operations performed on eight 32-bit words. At the start of the encryption process, the 32 bytes of the plaintext (256 bits) are divided into eight words of 32 bits each. Denoting by p_i the i'th byte of the plaintext P ($0 \leq i \leq 31$, where byte 0 is the most significant byte), and by $A, B, \ldots, H$ the eight 32-bit words, we load P's first byte into the most significant byte of A. We continue to load the bytes of P into the words, till the least significant byte of H contains P_{31}. Thus, the words are loaded as follow:

Word	Byte 0 (LSB)	Byte 1	Byte 2	Byte 3 (MSB)
A	P_3	P_2	P_1	P_0
B	P_7	P_6	P_5	P_4
C	P_{11}	P_{10}	P_9	p_8
D	p_{15}	P_{14}	P_{13}	P_{12}
E	P_{19}	P_{18}	P_{17}	P_{16}
F	P_{23}	P_{22}	P_{21}	P_{20}
G	P_{27}	P_{26}	P_{25}	P_{24}
H	P_{31}	P_{30}	P_{29}	P_{28}

After the initialization, the eight words are transformed by 64 successive applications of the round function. The final contents of the words are then sent to the output as ciphertext according to the same transformation as in the initialization, i.e., the ciphertext is $A^{64}, B^{64}, \ldots, H^{64}$.

The round function is as follows:

$$\begin{aligned}
T_1 &= H^i + \Sigma_1(E^i) + Ch(E^i, F^i, G^i) + K^i + W^i \\
T_2 &= \Sigma_0(A^i) + Maj(A^i, B^i, C^i) \\
A^{i+1} &= T_1 + T_2 \\
B^{i+1} &= A^i \\
C^{i+1} &= B^i \\
D^{i+1} &= C^i \\
E^{i+1} &= D^i + T_1 \\
F^{i+1} &= E^i \\
G^{i+1} &= F^i \\
H^{i+1} &= G^i
\end{aligned}$$

Where X^i is the word X before the round i (for $i = 0, \ldots, 63$). Thus, the plaintext is loaded into $A^0, B^0, \ldots, H^0$.

All the additions are performed modulo 2^{32} (regular 32-bit unsigned addition in 32-bit processors). Each round, a different round constant K^i is used, as well as a different round subkey W^i.

4.2.2 Key Expansion

The purpose of the key expansion is to construct 2048 bits of subkey material obtained from the secret key K. The key schedule is defined for keys between 0 and 512 bits long. However, keys shorter than 128 bits are not advised, as they offer insufficient security in today's terms. Keys shorter than 512 bits are padded with as many zeroes as needed to obtain a 512-bit string.

The 512-bit key is then loaded into an array named W. This is an array of sixty four 32-bit words, and the key is loaded into the first 16 entries of this array $(W^0, W^1, \ldots, W^{15})$. The remaining 48 entries are computed as follows:

$$W^i = \sigma_1(W^{i-2}) + W^{i-7} + \sigma_0(W^{i-15}) + W^{i-16}$$

for $i = 16, 17, \ldots 63$.
The two functions $\sigma_0()$ and $\sigma_1()$ accept 32-bit input and output a 32-bit value, which is linearly dependent on the input.

4.2.3 Decryption

The decryption process of SHACAL-2 is very similar to the encryption process. The same key schedule is used to derive the 64 subkey words W^i. The same constants K^i are also used in order to decrypt.

The ciphertext is loaded into the eight 32-bit words $A, B, \ldots, H$ in a similar way to the way the plaintext is loaded. Then the following procedure is applied 64 times:

$$\begin{aligned}
R_1 &= \Sigma_0(B^{i+1}) + Maj(B^{i+1}, C^{i+1}, D^{i+1}) \\
R_2 &= A^{i+1} - R_1 \\
A^i &= B^{i+1} \\
B^i &= C^{i+1} \\
C^i &= D^{i+1} \\
D^i &= E^{i+1} - R_2 \\
E^i &= F^{i+1} \\
F^i &= G^{i+1} \\
G^i &= H^{i+1} \\
H^i &= R_2 - \Sigma_1(F^{i+1}) - Ch(F^{i+1}, G^{i+1}, H^{i+1}) - K^i - W^i
\end{aligned}$$

Where $-$ is subtraction modulo 2^{32}.

The ciphertext is loaded as $A^{64}, B^{64}, \ldots, H^{64}$. The computed $A^0, B^0, \ldots, H^0$ are then loaded into the plaintext in the inverse transformation used to load the plaintext for the encryption process.

4.2.4 The Building Functions

SHACAL-2 uses 6 building functions besides addition: $\Sigma_0, \Sigma_1, \sigma_0, \sigma_0, Ch$ and Maj. The first four functions accept 32-bit input and output a 32-bit value which is linearly computed from the input. The remaining two Ch and Maj accept three 32-bit words and output a 32-bit output.

We first define the 4 linear operations:

$$\begin{aligned}
\Sigma_0(X) &= X \ggg_2 \oplus X \ggg_{13} \oplus X \ggg_{22} \\
\Sigma_1(X) &= X \ggg_6 \oplus X \ggg_{11} \oplus X \ggg_{25} \\
\sigma_0(X) &= X \ggg_7 \oplus X \ggg_{18} \oplus X \gg_3 \\
\sigma_1(X) &= X \ggg_{17} \oplus X \ggg_{19} \oplus X \gg_{10}
\end{aligned}$$

The Ch operation accepts three 32-bit words X, Y, Z. The operation is a bitwise choose function, i.e., if the i'th bit of X is set, then the i'th bit of the output is the i'th bit of Y. Otherwise, the i'th bit of the output is the i'th bit of Z. This operation can be efficiently implemented as:

$$Ch(X, Y, Z) = (X \wedge Y) \oplus (\overline{X} \wedge Z)$$

The Maj operation also accepts three 32-bit words X, Y, Z. The operation is a bitwise majority function, i.e., if the majority of the i'th bits of X, Y and Z are set, then the i'th output bit is set, and vice versa. This operation can be efficiently implemented as:

$$Maj(X, Y, Z) = (X \wedge Y) \oplus (X \wedge Z) \oplus (Y \wedge Z)$$

4.2.5 Constants

Each round, a round constant is used in the encryption. We list the round constants in the following table:

Round i	K_i	Round i	K_i	Round i	K_i	Round i	K_i
0	$\texttt{428A2F98}_\texttt{x}$	1	$\texttt{71374491}_\texttt{x}$	2	$\texttt{B5C0FBCF}_\texttt{x}$	3	E9B5D
4	$\texttt{3956C25B}_\texttt{x}$	5	$\texttt{59F111F1}_\texttt{x}$	6	$\texttt{923F82A4}_\texttt{x}$	7	AB1C5
8	$\texttt{D807AA98}_\texttt{x}$	9	$\texttt{12835B01}_\texttt{x}$	10	$\texttt{243185BE}_\texttt{x}$	11	550C7
12	$\texttt{72BE5D74}_\texttt{x}$	13	$\texttt{80DEB1FE}_\texttt{x}$	14	$\texttt{9BDC06A7}_\texttt{x}$	15	C19BF
16	$\texttt{E49B69C1}_\texttt{x}$	17	$\texttt{EFBE4786}_\texttt{x}$	18	$\texttt{0FC19DC6}_\texttt{x}$	19	240CA
20	$\texttt{2DE92C6F}_\texttt{x}$	21	$\texttt{4A7484AA}_\texttt{x}$	22	$\texttt{5CB0A9DC}_\texttt{x}$	23	76F98
24	$\texttt{983E5152}_\texttt{x}$	25	$\texttt{A831C66D}_\texttt{x}$	26	$\texttt{B00327C8}_\texttt{x}$	27	BF597
28	$\texttt{C6E00BF3}_\texttt{x}$	29	$\texttt{D5A79147}_\texttt{x}$	30	$\texttt{06CA6351}_\texttt{x}$	31	14292
32	$\texttt{27B70A85}_\texttt{x}$	33	$\texttt{2E1B2138}_\texttt{x}$	34	$\texttt{4D2C6DFC}_\texttt{x}$	35	53380
36	$\texttt{650A7354}_\texttt{x}$	37	$\texttt{766A0ABB}_\texttt{x}$	38	$\texttt{81C2C92E}_\texttt{x}$	39	92722
40	$\texttt{A2BFE8A1}_\texttt{x}$	41	$\texttt{A81A664B}_\texttt{x}$	42	$\texttt{C24B8B70}_\texttt{x}$	43	C76C5
44	$\texttt{D192E819}_\texttt{x}$	45	$\texttt{D6990624}_\texttt{x}$	46	$\texttt{F40E3585}_\texttt{x}$	47	106AA
48	$\texttt{19A4C116}_\texttt{x}$	49	$\texttt{1E376C08}_\texttt{x}$	50	$\texttt{2748774C}_\texttt{x}$	51	34B0B
52	$\texttt{391C0CB3}_\texttt{x}$	53	$\texttt{4ED8AA4A}_\texttt{x}$	54	$\texttt{5B9CCA4F}_\texttt{x}$	55	682E6
56	$\texttt{748F82EE}_\texttt{x}$	57	$\texttt{78A5636F}_\texttt{x}$	58	$\texttt{84C87814}_\texttt{x}$	59	8CC70
60	$\texttt{90BEFFFA}_\texttt{x}$	61	$\texttt{A4506CEB}_\texttt{x}$	62	$\texttt{BEF9A3F7}_\texttt{x}$	63	C6717

4.3 Test vectors

Part C

Collision-Resistant Hash Functions

1. Whirlpool

1.1 Introduction

1.1.1 Overview

Whirlpool is a collision-resistant hash function submitted to NESSIE by Paulo Barreto and Vincent Rijmen [38]. The design of Whirlpool is based on an underlying block cipher that is used in the so-called Miyaguchi-Preneel hashing mode. This block cipher is similar in structure to the AES cipher (see Part B of the NESSIE portfolio), but it works on bigger blocks of 512 bits.

1.1.2 Outline of the primitive

Whirlpool maps a message M consisting of an arbitrary number of bits onto a 512-bit hash value Whirlpool(M). The algorithm is based on a compression function, that is used iteratively on message blocks of 512 bits. This function is based on an underlying dedicated 512-bit block cipher using a 512-bit key.

First, the message is expanded to an appropriate length and divided into message blocks $M_0 \ldots M_{t-1}$ where each block M_i $(0 \leq i \leq t-1)$ consists of 512 bits. An initial value is defined as the string consisting of 512 0-bits: $H_0 = 0$.

The compression function of Whirlpool transforms an input value H_i and a message block M_i into an output value H_{i+1}. This function depends on encryption with an internal block cipher W where M_i forms the plaintext and H_i serves as key. Furthermore, the obtained ciphertext is XOR'ed (exclusive-or operation) with both M_i and H_i (note that all string are 512 bits long). That is, $H_{i+1} = \mathrm{W}_{H_i}(M_i) \oplus M_i \oplus H_i$. In the next use of the compression function H_{i+1} forms the input together with the next message block M_{i+1}. This results in the following outline of the Whirlpool algorithm.

1. Expand the message M and divide it into message blocks $M_0 \ldots M_{t-1}$.
2. Define the initial value $H_0 = 0$.
3. For $0 \leq i \leq t-1$ use the compression function to compute $H_{i+1} = \mathrm{W}_{H_i}(M_i) \oplus M_i \oplus H_i$.
4. Set the hash value Whirlpool(M) $= H_t$.

1.1.3 Security and performance

The security of Whirlpool can be proven based on the assumption that certain ideal properties hold for the underlying block cipher W. This block cipher is very similar to AES. There are no known short-cut attacks on Whirlpool and the algorithm has a high security level because of the long hash value (512 bits).

The performance of Whirlpool depends on the performance of the underlying block cipher which is quite efficient. However one needs to recompute the key schedule for this cipher for each use of the compression function (that is for every message block of 512 bits). The structure of the algorithm is not oriented towards any particular platform but different optimisations can be made on different platforms. Also, the special structure of the components, especially the S-box that is used, allows for efficient hardware implementations.

1.2 Description

In this section we give a complete specification of Whirlpool, where we first describe the underlying block cipher, and next show how this block cipher is used to define the hash function Whirlpool.

1.2.1 Cipher

The block cipher W is a Substitution Permutation Network (SPN) consisting of 10 rounds. The cipher takes a 512-bit plaintext P as input and encrypts it into a 512-bit ciphertext C, according to a 512-bit key K. This is denoted by $C = \mathrm{W}_K(P)$. All operations during the iterative encryption procedure are performed on an array of 8×8 bytes, which are interpreted as elements in the finite field $GF(2^8)$. A single round consists of a nonlinear byte substitution (`SubBytes`), a two-stage affine transformation (`ShiftColumns/MixRows`), and a round key addition (`AddRoundKey`). The round keys, each consisting of 512 bits, are derived from the original 512-bit key by a separate key expansion routine.

The 8×8-byte array on which all operations are performed, is called the *state* and denoted by $S = (s_{i,j})$ with $0 \le i, j \le 7$. At the start of the encryption process, the 64 bytes of the state (512 bits) are initialised with plaintext bytes p_i $(0 \le i \le 63)$, from left to right and from top to bottom as illustrated below. Note that in these initialisations, as well as in most of the subsequent operations, software implementations on 64-bit platforms can benefit from the fact that the 8 bytes of each *row* can be treated as a single 64-bit word.

$$\begin{array}{cccccccc} p_0 & p_1 & p_2 & p_3 & p_4 & p_5 & p_6 & p_7 \\ p_8 & p_9 & p_{10} & p_{11} & p_{12} & p_{13} & p_{14} & p_{15} \\ p_{16} & p_{17} & p_{18} & p_{19} & p_{20} & p_{21} & p_{22} & p_{23} \\ p_{24} & p_{25} & p_{26} & p_{27} & p_{28} & p_{29} & p_{30} & p_{31} \\ p_{32} & p_{33} & p_{34} & p_{35} & p_{36} & p_{37} & p_{38} & p_{39} \\ p_{40} & p_{41} & p_{42} & p_{43} & p_{44} & p_{45} & p_{46} & p_{47} \\ p_{48} & p_{49} & p_{50} & p_{51} & p_{52} & p_{53} & p_{54} & p_{55} \\ p_{56} & p_{57} & p_{58} & p_{59} & p_{60} & p_{61} & p_{62} & p_{63} \end{array} \xrightarrow{S=P} \begin{array}{cccccccc} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} & s_{0,4} & s_{0,5} & s_{0,6} & s_{0,7} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} & s_{1,4} & s_{1,5} & s_{1,6} & s_{1,7} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} & s_{2,4} & s_{2,5} & s_{2,6} & s_{2,7} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} & s_{3,4} & s_{3,5} & s_{3,6} & s_{3,7} \\ s_{4,0} & s_{4,1} & s_{4,2} & s_{4,3} & s_{4,4} & s_{4,5} & s_{4,6} & s_{4,7} \\ s_{5,0} & s_{5,1} & s_{5,2} & s_{5,3} & s_{5,4} & s_{5,5} & s_{5,6} & s_{5,7} \\ s_{6,0} & s_{6,1} & s_{6,2} & s_{6,3} & s_{6,4} & s_{6,5} & s_{6,6} & s_{6,7} \\ s_{7,0} & s_{7,1} & s_{7,2} & s_{7,3} & s_{7,4} & s_{7,5} & s_{7,6} & s_{7,7} \end{array}$$

After an initial round key addition, the state is transformed by 10 successive applications of a round function. The final content of the state is then sent to the output as ciphertext (taking the bytes out of the state array from left to right and from top to bottom). This encryption process is described in pseudo-code as:

```
$S = P$
$S =$ AddRoundKey($S$,$K^0$)
for $r = 1$ to 10 do
  $S =$ SubBytes($S$)
  $S =$ ShiftColumns($S$)
  $S =$ MixRows($S$)
  $S =$ AddRoundKey($S$,$K^r$)
end for
$C = S$
```

1.2.1.1 The SubBytes transformation

The SubBytes operation substitutes each individual state byte $s_{i,j}$ by a new value $s'_{i,j} = S_W(s_{i,j})$. The function S_W is a fixed invertible nonlinear mapping (S-box), given by Table 56. Using this table, the byte 53_x for example is mapped to 71_x.

$$\begin{array}{cccccccc} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} & s_{0,4} & s_{0,5} & s_{0,6} & s_{0,7} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} & s_{1,4} & s_{1,5} & s_{1,6} & s_{1,7} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} & s_{2,4} & s_{2,5} & s_{2,6} & s_{2,7} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} & s_{3,4} & s_{3,5} & s_{3,6} & s_{3,7} \\ s_{4,0} & s_{4,1} & s_{4,2} & s_{4,3} & s_{4,4} & s_{4,5} & s_{4,6} & s_{4,7} \\ s_{5,0} & s_{5,1} & s_{5,2} & s_{5,3} & s_{5,4} & s_{5,5} & s_{5,6} & s_{5,7} \\ s_{6,0} & s_{6,1} & s_{6,2} & s_{6,3} & s_{6,4} & s_{6,5} & s_{6,6} & s_{6,7} \\ s_{7,0} & s_{7,1} & s_{7,2} & s_{7,3} & s_{7,4} & s_{7,5} & s_{7,6} & s_{7,7} \end{array} \xrightarrow{S_W(\cdot)} \begin{array}{cccccccc} s'_{0,0} & s'_{0,1} & s'_{0,2} & s'_{0,3} & s'_{0,4} & s'_{0,5} & s'_{0,6} & s'_{0,7} \\ s'_{1,0} & s'_{1,1} & s'_{1,2} & s'_{1,3} & s'_{1,4} & s'_{1,5} & s'_{1,6} & s'_{1,7} \\ s'_{2,0} & s'_{2,1} & s'_{2,2} & s'_{2,3} & s'_{2,4} & s'_{2,5} & s'_{2,6} & s'_{2,7} \\ s'_{3,0} & s'_{3,1} & s'_{3,2} & s'_{3,3} & s'_{3,4} & s'_{3,5} & s'_{3,6} & s'_{3,7} \\ s'_{4,0} & s'_{4,1} & s'_{4,2} & s'_{4,3} & s'_{4,4} & s'_{4,5} & s'_{4,6} & s'_{4,7} \\ s'_{5,0} & s'_{5,1} & s'_{5,2} & s'_{5,3} & s'_{5,4} & s'_{5,5} & s'_{5,6} & s'_{5,7} \\ s'_{6,0} & s'_{6,1} & s'_{6,2} & s'_{6,3} & s'_{6,4} & s'_{6,5} & s'_{6,6} & s'_{6,7} \\ s'_{7,0} & s'_{7,1} & s'_{7,2} & s'_{7,3} & s'_{7,4} & s'_{7,5} & s'_{7,6} & s'_{7,7} \end{array}$$

In most software environments, the S-box can simply be implemented as a lookup table, but when memory is restricted, it is possible to build more memory-efficient (but slower) implementations using the fact that the S-box is based on a three-layer structure using smaller 4-bit substitution boxes. For details on such implementations we refer to [38].

Table 56. The S-box S_W

	00$_x$	01$_x$	02$_x$	03$_x$	04$_x$	05$_x$	06$_x$	07$_x$	08$_x$	09$_x$	0A$_x$	0B$_x$	0C$_x$	0D$_x$	0E$_x$	0F$_x$
00$_x$	18	23	C6	E8	87	B8	01	4F	36	A6	D2	F5	79	6F	91	52
10$_x$	60	BC	9B	8E	A3	0C	7B	35	1D	E0	D7	C2	2E	4B	FE	57
20$_x$	15	77	37	E5	9F	F0	4A	DA	58	C9	29	0A	B1	A0	6B	85
30$_x$	BD	5D	10	F4	CB	3E	05	67	E4	27	41	8B	A7	7D	95	D8
40$_x$	FB	EE	7C	66	DD	17	47	9E	CA	2D	BF	07	AD	5A	83	33
50$_x$	63	02	AA	71	C8	19	49	D9	F2	E3	5B	88	9A	26	32	B0
60$_x$	E9	0F	D5	80	BE	CD	34	48	FF	7A	90	5F	20	68	1A	AE
70$_x$	B4	54	93	22	64	F1	73	12	40	08	C3	EC	DB	A1	8D	3D
80$_x$	97	00	CF	2B	76	82	D6	1B	B5	AF	6A	50	45	F3	30	EF
90$_x$	3F	55	A2	EA	65	BA	2F	C0	DE	1C	FD	4D	92	75	06	8A
A0$_x$	B2	E6	0E	1F	62	D4	A8	96	F9	C5	25	59	84	72	39	4C
B0$_x$	5E	78	38	8C	D1	A5	E2	61	B3	21	9C	1E	43	C7	FC	04
C0$_x$	51	99	6D	0D	FA	DF	7E	24	3B	AB	CE	11	8F	4E	B7	EB
D0$_x$	3C	81	94	F7	B9	13	2C	D3	E7	6E	C4	03	56	44	7F	A9
E0$_x$	2A	BB	C1	53	DC	0B	9D	6C	31	74	F6	46	AC	89	14	E1
F0$_x$	16	3A	69	09	70	B6	D0	ED	CC	42	98	A4	28	5C	F8	86

1.2.1.2 The ShiftColumns transformation

The ShiftColumns transformation operates on the columns of the state: the first column is kept unchanged, the remaining columns are cyclically shifted down over 1,2,. . . ,7 byte positions respectively. The operations are illustrated below:

$$
\begin{matrix}
s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} & s_{0,4} & s_{0,5} & s_{0,6} & s_{0,7}\\
s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} & s_{1,4} & s_{1,5} & s_{1,6} & s_{1,7}\\
s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} & s_{2,4} & s_{2,5} & s_{2,6} & s_{2,7}\\
s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} & s_{3,4} & s_{3,5} & s_{3,6} & s_{3,7}\\
s_{4,0} & s_{4,1} & s_{4,2} & s_{4,3} & s_{4,4} & s_{4,5} & s_{4,6} & s_{4,7}\\
s_{5,0} & s_{5,1} & s_{5,2} & s_{5,3} & s_{5,4} & s_{5,5} & s_{5,6} & s_{5,7}\\
s_{6,0} & s_{6,1} & s_{6,2} & s_{6,3} & s_{6,4} & s_{6,5} & s_{6,6} & s_{6,7}\\
s_{7,0} & s_{7,1} & s_{7,2} & s_{7,3} & s_{7,4} & s_{7,5} & s_{7,6} & s_{7,7}
\end{matrix}
\xrightarrow{\circlearrowleft}
\begin{matrix}
s_{0,0} & s_{7,1} & s_{6,2} & s_{5,3} & s_{4,4} & s_{3,5} & s_{2,6} & s_{1,7}\\
s_{1,0} & s_{0,1} & s_{7,2} & s_{6,3} & s_{5,4} & s_{4,5} & s_{3,6} & s_{2,7}\\
s_{2,0} & s_{1,1} & s_{0,2} & s_{7,3} & s_{6,4} & s_{5,5} & s_{4,6} & s_{3,7}\\
s_{3,0} & s_{2,1} & s_{1,2} & s_{0,3} & s_{7,4} & s_{6,5} & s_{5,6} & s_{4,7}\\
s_{4,0} & s_{3,1} & s_{2,2} & s_{1,3} & s_{0,4} & s_{7,5} & s_{6,6} & s_{5,7}\\
s_{5,0} & s_{4,1} & s_{3,2} & s_{2,3} & s_{1,4} & s_{0,5} & s_{7,6} & s_{6,7}\\
s_{6,0} & s_{5,1} & s_{4,2} & s_{3,3} & s_{2,4} & s_{1,5} & s_{0,6} & s_{7,7}\\
s_{7,0} & s_{6,1} & s_{5,2} & s_{4,3} & s_{3,4} & s_{2,5} & s_{1,6} & s_{0,7}
\end{matrix}
$$

1.2.1.3 The MixRows transformation

The MixRows operation applies a linear transform to the rows of the state. This transform can be represented as a matrix multiplication, where each byte is interpreted as an element in the finite field $GF(2^8)$:

$$\begin{bmatrix} s'_{i,0} \\ s'_{i,1} \\ s'_{i,2} \\ s'_{i,3} \\ s'_{i,4} \\ s'_{i,5} \\ s'_{i,6} \\ s'_{i,7} \end{bmatrix}^T = \begin{bmatrix} s_{i,0} \\ s_{i,1} \\ s_{i,2} \\ s_{i,3} \\ s_{i,4} \\ s_{i,5} \\ s_{i,6} \\ s_{i,7} \end{bmatrix}^T \cdot A = \begin{bmatrix} s_{i,0} \\ s_{i,1} \\ s_{i,2} \\ s_{i,3} \\ s_{i,4} \\ s_{i,5} \\ s_{i,6} \\ s_{i,7} \end{bmatrix}^T \cdot \begin{bmatrix} \mathtt{01}_\mathtt{x} & \mathtt{01}_\mathtt{x} & \mathtt{03}_\mathtt{x} & \mathtt{01}_\mathtt{x} & \mathtt{05}_\mathtt{x} & \mathtt{08}_\mathtt{x} & \mathtt{09}_\mathtt{x} & \mathtt{05}_\mathtt{x} \\ \mathtt{05}_\mathtt{x} & \mathtt{01}_\mathtt{x} & \mathtt{01}_\mathtt{x} & \mathtt{03}_\mathtt{x} & \mathtt{01}_\mathtt{x} & \mathtt{05}_\mathtt{x} & \mathtt{08}_\mathtt{x} & \mathtt{09}_\mathtt{x} \\ \mathtt{09}_\mathtt{x} & \mathtt{05}_\mathtt{x} & \mathtt{01}_\mathtt{x} & \mathtt{01}_\mathtt{x} & \mathtt{03}_\mathtt{x} & \mathtt{01}_\mathtt{x} & \mathtt{05}_\mathtt{x} & \mathtt{08}_\mathtt{x} \\ \mathtt{08}_\mathtt{x} & \mathtt{09}_\mathtt{x} & \mathtt{05}_\mathtt{x} & \mathtt{01}_\mathtt{x} & \mathtt{01}_\mathtt{x} & \mathtt{03}_\mathtt{x} & \mathtt{01}_\mathtt{x} & \mathtt{05}_\mathtt{x} \\ \mathtt{05}_\mathtt{x} & \mathtt{08}_\mathtt{x} & \mathtt{09}_\mathtt{x} & \mathtt{05}_\mathtt{x} & \mathtt{01}_\mathtt{x} & \mathtt{01}_\mathtt{x} & \mathtt{03}_\mathtt{x} & \mathtt{01}_\mathtt{x} \\ \mathtt{01}_\mathtt{x} & \mathtt{05}_\mathtt{x} & \mathtt{08}_\mathtt{x} & \mathtt{09}_\mathtt{x} & \mathtt{05}_\mathtt{x} & \mathtt{01}_\mathtt{x} & \mathtt{01}_\mathtt{x} & \mathtt{03}_\mathtt{x} \\ \mathtt{03}_\mathtt{x} & \mathtt{01}_\mathtt{x} & \mathtt{05}_\mathtt{x} & \mathtt{08}_\mathtt{x} & \mathtt{09}_\mathtt{x} & \mathtt{05}_\mathtt{x} & \mathtt{01}_\mathtt{x} & \mathtt{01}_\mathtt{x} \\ \mathtt{01}_\mathtt{x} & \mathtt{03}_\mathtt{x} & \mathtt{01}_\mathtt{x} & \mathtt{05}_\mathtt{x} & \mathtt{08}_\mathtt{x} & \mathtt{09}_\mathtt{x} & \mathtt{05}_\mathtt{x} & \mathtt{01}_\mathtt{x} \end{bmatrix}$$

As a result, the first byte of the row, for example, is replaced by (apply the first column of the matrix A):

$$s'_{i,0} = s_{i,0} \oplus (s_{i,1} \cdot \mathtt{05}_\mathtt{x}) \oplus (s_{i,2} \cdot \mathtt{09}_\mathtt{x}) \oplus (s_{i,3} \cdot \mathtt{08}_\mathtt{x}) \oplus (s_{i,4} \cdot \mathtt{05}_\mathtt{x}) \oplus s_{i,5} \oplus (s_{i,6} \cdot \mathtt{03}_\mathtt{x}) \oplus s_{i,7}\,.$$

The "$\oplus$" operator in this expression denotes addition in $GF(2^8)$, which corresponds to bitwise XOR (eXclusive OR). The multiplications are performed modulo the irreducible polynomial of the field. In the case of the cipher W the polynomial $x^8 + x^4 + x^3 + x^2 + 1$ is used. For example:

$$\begin{aligned} \mathtt{93}_\mathtt{x} \cdot \mathtt{03}_\mathtt{x} &= \mathtt{10010011}_\mathtt{b} \cdot \mathtt{00000011}_\mathtt{b} \\ &= (x^7 + x^4 + x + 1) \cdot (x + 1) && \bmod x^8 + x^4 + x^3 + x^2 + 1 \\ &= x^8 + x^7 + x^5 + x^4 + x^2 + 1 && \bmod x^8 + x^4 + x^3 + x^2 + 1 \\ &= x^7 + x^5 + x^3 \\ &= \mathtt{10101000}_\mathtt{b} \\ &= \mathtt{A8}_\mathtt{x} \end{aligned}$$

The effect of the complete `MixRows` operation is depicted below.

$$\begin{array}{cccccccc} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} & s_{0,4} & s_{0,5} & s_{0,6} & s_{0,7} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} & s_{1,4} & s_{1,5} & s_{1,6} & s_{1,7} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} & s_{2,4} & s_{2,5} & s_{2,6} & s_{2,7} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} & s_{3,4} & s_{3,5} & s_{3,6} & s_{3,7} \\ s_{4,0} & s_{4,1} & s_{4,2} & s_{4,3} & s_{4,4} & s_{4,5} & s_{4,6} & s_{4,7} \\ s_{5,0} & s_{5,1} & s_{5,2} & s_{5,3} & s_{5,4} & s_{5,5} & s_{5,6} & s_{5,7} \\ s_{6,0} & s_{6,1} & s_{6,2} & s_{6,3} & s_{6,4} & s_{6,5} & s_{6,6} & s_{6,7} \\ s_{7,0} & s_{7,1} & s_{7,2} & s_{7,3} & s_{7,4} & s_{7,5} & s_{7,6} & s_{7,7} \end{array} \xrightarrow{\cdot \times A} \begin{array}{cccccccc} s'_{0,0} & s'_{0,1} & s'_{0,2} & s'_{0,3} & s'_{0,4} & s'_{0,5} & s'_{0,6} & s'_{0,7} \\ s'_{1,0} & s'_{1,1} & s'_{1,2} & s'_{1,3} & s'_{1,4} & s'_{1,5} & s'_{1,6} & s'_{1,7} \\ s'_{2,0} & s'_{2,1} & s'_{2,2} & s'_{2,3} & s'_{2,4} & s'_{2,5} & s'_{2,6} & s'_{2,7} \\ s'_{3,0} & s'_{3,1} & s'_{3,2} & s'_{3,3} & s'_{3,4} & s'_{3,5} & s'_{3,6} & s'_{3,7} \\ s'_{4,0} & s'_{4,1} & s'_{4,2} & s'_{4,3} & s'_{4,4} & s'_{4,5} & s'_{4,6} & s'_{4,7} \\ s'_{5,0} & s'_{5,1} & s'_{5,2} & s'_{5,3} & s'_{5,4} & s'_{5,5} & s'_{5,6} & s'_{5,7} \\ s'_{6,0} & s'_{6,1} & s'_{6,2} & s'_{6,3} & s'_{6,4} & s'_{6,5} & s'_{6,6} & s'_{6,7} \\ s'_{7,0} & s'_{7,1} & s'_{7,2} & s'_{7,3} & s'_{7,4} & s'_{7,5} & s'_{7,6} & s'_{7,7} \end{array}$$

1.2.1.4 The AddRoundKey transformation

Finally, in the AddRoundKey transformation, each bit of the state is XORed with the corresponding bit of a 512-bit round key K^r. Each round requires a separate round key, and the generation of these keys is performed by the key expansion routine described in the next subsection.

$$\begin{matrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} & s_{0,4} & s_{0,5} & s_{0,6} & s_{0,7} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} & s_{1,4} & s_{1,5} & s_{1,6} & s_{1,7} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} & s_{2,4} & s_{2,5} & s_{2,6} & s_{2,7} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} & s_{3,4} & s_{3,5} & s_{3,6} & s_{3,7} \\ s_{4,0} & s_{4,1} & s_{4,2} & s_{4,3} & s_{4,4} & s_{4,5} & s_{4,6} & s_{4,7} \\ s_{5,0} & s_{5,1} & s_{5,2} & s_{5,3} & s_{5,4} & s_{5,5} & s_{5,6} & s_{5,7} \\ s_{6,0} & s_{6,1} & s_{6,2} & s_{6,3} & s_{6,4} & s_{6,5} & s_{6,6} & s_{6,7} \\ s_{7,0} & s_{7,1} & s_{7,2} & s_{7,3} & s_{7,4} & s_{7,5} & s_{7,6} & s_{7,7} \end{matrix} \xrightarrow{\cdot \oplus K^r} \begin{matrix} s'_{0,0} & s'_{0,1} & s'_{0,2} & s'_{0,3} & s'_{0,4} & s'_{0,5} & s'_{0,6} & s'_{0,7} \\ s'_{1,0} & s'_{1,1} & s'_{1,2} & s'_{1,3} & s'_{1,4} & s'_{1,5} & s'_{1,6} & s'_{1,7} \\ s'_{2,0} & s'_{2,1} & s'_{2,2} & s'_{2,3} & s'_{2,4} & s'_{2,5} & s'_{2,6} & s'_{2,7} \\ s'_{3,0} & s'_{3,1} & s'_{3,2} & s'_{3,3} & s'_{3,4} & s'_{3,5} & s'_{3,6} & s'_{3,7} \\ s'_{4,0} & s'_{4,1} & s'_{4,2} & s'_{4,3} & s'_{4,4} & s'_{4,5} & s'_{4,6} & s'_{4,7} \\ s'_{5,0} & s'_{5,1} & s'_{5,2} & s'_{5,3} & s'_{5,4} & s'_{5,5} & s'_{5,6} & s'_{5,7} \\ s'_{6,0} & s'_{6,1} & s'_{6,2} & s'_{6,3} & s'_{6,4} & s'_{6,5} & s'_{6,6} & s'_{6,7} \\ s'_{7,0} & s'_{7,1} & s'_{7,2} & s'_{7,3} & s'_{7,4} & s'_{7,5} & s'_{7,6} & s'_{7,7} \end{matrix}$$

1.2.1.5 The key expansion

The purpose of the key expansion is to construct eleven 512-bit round keys K^r from a single 512-bit key K. The key expansion routine is a recursive process in which each new round key is directly derived from the preceeding round key. It proceeds according to the pseudo-code below:

```
$K^0 = K$
for $r = 1$ to 10 do
  $S =$ SubBytes($K^{r-1}$)
  $S =$ ShiftColumns($S$)
  $S =$ MixRows($S$)
  $K^r =$ AddRoundKey($S$,$C^r$)
end for
```

The first round key K^0, used in the initial key addition, is directly filled with the 64 bytes of the key K (filling in these key bytes in an 8×8-byte array from left to right and from top to bottom). The remaining round keys K^r are derived recursively by application of the round function. The constants C^r, used in the AddRoundKey operation, are 8×8-byte arrays defined as follows (for $1 \le r \le 10$):

$$\begin{aligned} C^r_{0,j} &\equiv S_W(8(r-1)+j)\,, & 0 \le j \le 7\,, \\ C^r_{i,j} &\equiv 0\,, & 1 \le i \le 7\,, 0 \le j \le 7\,. \end{aligned}$$

1.2.2 From cipher to hash

In this section we specify how to use the block cipher W in order to define the Whirlpool hashing function. Let M be a message consisting of $Mlen$ bits with $Mlen < 2^{256}$. The hash value Whirlpool(M) is computed as described below.

1.2.2.1 Message expansion

The message M needs to be expanded so that it contains a number of bits which is an integer multiple of 512. This is done in the following way:

1. M is concatenated with a single 1-bit.
2. The result of the previous step is concatenated with a string of r 0-bits. Here $0 \leq r \leq 511$ and the length of the resultant string must be an odd multiple of 256. That is, $Mlen + 1 + r \equiv 256 \bmod 512$.
3. Concatenate the string resulting from the previous step with the 256-bit representation of $Mlen$ (the original length of M), most significant bit first.

The expanded message is then divided into message blocks with a length of 512 bits each. If the expanded message is $512t$ bits long ($Mlen + 1 + r + 256 = 512t$), this results in t message blocks $M_0, M_1, \ldots, M_{t-1}$.

1.2.2.2 Hash function

Given the 512-bit message blocks M_i and the block cipher W, WHIRLPOOL computes the hash value in the following iterative manner.

- Define an initial value H_0 as a string of 512 0-bits.

$$H_0 := \texttt{0000}\ldots\texttt{00}_{\texttt{x}} .$$

- For $i = 0, \ldots, t-1$ compute a new 512-bit string H_{i+1} from the previously obtained H_i and the message block M_i:

$$H_{i+1} := \mathrm{W}_{H_i}(M_i) \oplus M_i \oplus H_i .$$

  That is, apply the cipher W with the current message block M_i as plaintext and the previously obtained H_i as key (this means that the ciphers key schedule routine must be executed as well). Next, compute the bitwise exclusive-or (XOR) of the following three 512-bit strings: the ciphertext obtained from W, the string M_i, and the string H_i.
- Define the 512-bit hash value WHIRLPOOL(M) as the string obtained from the last step of the iteration.

$$\text{WHIRLPOOL}(M) := H_t .$$

1.2.3 The updated Whirlpool specification

After the final selection of the NESSIE portfolio algorithms, Shirai and Shibutani [579] announced a flaw in the original WHIRLPOOL diffusion matrix, that made its branch number suboptimal. Although this flaw *per se* does not seem to introduce an effective vulnerability, the designers decided to propose a replacement matrix. Besides displaying optimal branch number and thus keeping the existing security analysis unchanged, the new matrix also leads to more efficient implementation in 8-bit platforms and hardware. The updated design is included in ISO/IEC 10118-3, the international standard on dedicated hash functions.

The diffusion matrix is used in the MixRows operation. The new matrix is given by:

$$A = \begin{bmatrix} 01_x & 01_x & 04_x & 01_x & 08_x & 05_x & 02_x & 09_x \\ 09_x & 01_x & 01_x & 04_x & 01_x & 08_x & 05_x & 02_x \\ 02_x & 09_x & 01_x & 01_x & 04_x & 01_x & 08_x & 05_x \\ 05_x & 02_x & 09_x & 01_x & 01_x & 04_x & 01_x & 08_x \\ 08_x & 05_x & 02_x & 09_x & 01_x & 01_x & 04_x & 01_x \\ 01_x & 08_x & 05_x & 02_x & 09_x & 01_x & 01_x & 04_x \\ 04_x & 01_x & 08_x & 05_x & 02_x & 09_x & 01_x & 01_x \\ 01_x & 04_x & 01_x & 08_x & 05_x & 02_x & 09_x & 01_x \end{bmatrix}.$$

This is the only change in the design.

1.3 Test vectors for Whirlpool (original version)

The following test vectors are generated for WHIRLPOOL. They are denoted as pairs (message, hash). This correponds to hash = WHIRLPOOL(message).

The hash is denoted as a hexadecimal number of 64 bytes (512 bits). The message is denoted as an ASCII string (e.g., the character "a" corresponds to the hexadecimal byte 61_x).

```
message = "" (empty string)
hash = B3E1AB6EAF640A34F784593F2074416A
       CCD3B8E62C620175FCA0997B1BA23473
       39AA0D79E754C308209EA36811DFA40C
       1C32F1A2B9004725D987D3635165D3C8

message = "a"
hash = F4B620445AE62431DBD6DBCEC64D2A30
       31CD2F48DF5E755F30B3D069929ED4B4
       EDA0AE65441BC86746021FB7F2167F84
       D67566EFABA003F0ABB67A42A2CE5B13

message = "abc"
hash = 54EE18B0BBD4DD38A211699F28297931
       56E5842DF502A2A25995C6C541F28CC0
       50FF57D4AF772DEE7CEDCC4C34C3B8EC
       06446C6657F2F36C2C06464399879B86

message = "message digest"
hash = 29E158BA336CE7F930115178A6C86019
       F0F413ADB283D8F0798AF06CA0A06D6D
       6F295A333B1C24BDA2F429AC918A3748
       AEF90F7A2C8BFB084D5F979CF4E7B2B5
```

```
message = "abcdefghijklmnopqrstuvwxyz"
hash = 5AC9757E1407432DAF348A972B8AD4A6
       5C1123CF1F9B779C1AE7EE2D540F30B3
       CEFA8F98DCA5FBB42084C5C2F161A7B4
       0EB6B4A1FC7F9AAAB92A4BB6002EDC5E

message = "A...Za...z0...9"
hash = CAE4175F09753DE84974CFA968621092
       FE41EE9DE913919C2B452E6CB4240567
       21D640E563F628F29DD3BD0030837AE4
       AC14AA17308505A92E5F7A92F112BE75

message = 8 times "1234567890"
hash = E5965B4565B041A0D459610E5E48E944
       C4830CD16FEBA02D9D263E7DA8DE6A6B
       88966709BF28A5328D928312E7A172DA
       4CFF72FE6DE02277DAE4B1DBA49689A2

message = 1 million times "a"
hash = 5BC84BA27B464B4B761B5E48F314CFDB
       9F2C27B8C9BD664D26C99CF1F556D89C
       4270FC60D62340487FE8738EF2DC4168
       97CEB419F6B48335880E79D5A0046BE2
```

1.4 Test vectors for Whirlpool (updated version)

The following test vectors are generated for Whirlpool. They are denoted as pairs (message, hash). This correponds to hash = Whirlpool(message).

The hash is denoted as a hexadecimal number of 64 bytes (512 bits). The message is denoted as an ASCII string (e.g., the character "a" corresponds to the hexadecimal byte 61_x).

```
message = "" (empty string)
hash = 19FA61D75522A4669B44E39C1D2E1726
       C530232130D407F89AFEE0964997F7A7
       3E83BE698B288FEBCF88E3E03C4F0757
       EA8964E59B63D93708B138CC42A66EB3

message = "a"
hash = 8ACA2602792AEC6F11A67206531FB7D7
       F0DFF59413145E6973C45001D0087B42
       D11BC645413AEFF63A42391A39145A59
       1A92200D560195E53B478584FDAE231A

message = "abc"
hash = 4E2448A4C6F486BB16B6562C73B4020B
       F3043E3A731BCE721AE1B303D97E6D4C
       7181EEBDB6C57E277D0E34957114CBD6
       C797FC9D95D8B582D225292076D4EEF5

message = "message digest"
hash = 378C84A4126E2DC6E56DCC7458377AAC
       838D00032230F53CE1F5700C0FFB4D3B
       8421557659EF55C106B4B52AC5A4AAA6
       92ED920052838F3362E86DBD37A8903E

message = "abcdefghijklmnopqrstuvwxyz"
hash = F1D754662636FFE92C82EBB9212A484A
       8D38631EAD4238F5442EE13B8054E41B
       08BF2A9251C30B6A0B8AAE86177AB4A6
       F68F673E7207865D5D9819A3DBA4EB3B

message = "A...Za...z0...9"
hash = DC37E008CF9EE69BF11F00ED9ABA2690
       1DD7C28CDEC066CC6AF42E40F82F3A1E
       08EBA26629129D8FB7CB57211B9281A6
       5517CC879D7B962142C65F5A7AF01467

message = 8 times "1234567890"
hash = 466EF18BABB0154D25B9D38A6414F5C0
       8784372BCCB204D6549C4AFADB601429
       4D5BD8DF2A6C44E538CD047B2681A51A
       2C60481E88C5A20B2C2A80CF3A9A083B

message = 1 million times "a"
hash = 0C99005BEB57EFF50A7CF005560DDF5D
       29057FD86B20BFD62DECA0F1CCEA4AF5
       1FC15490EDDC47AF32BB2B66C34FF9AD
       8C6008AD677F77126953B226E4ED8B01
```

2. SHA-256, SHA-384 and SHA-512

2.1 Introduction

2.1.1 Overview

In 1993 NIST issued FIPS-180 [463] which presented a standard secure hash function called SHA. In FIPS-180-1 [465] from 1995 SHA was tweaked (a rotation left by 1 bit position was added) and the tweaked version was named SHA-1. Both versions have a digest size of 160 bits. Only recently a work by Saarinen [553] showed a slide attack on SHA-1.

As the security industry and community demanded larger security margins, in August 2002 NIST published three more hash functions: SHA-256, SHA-384 and SHA-512. FIPS-180-2 [472] defines these three new hash functions along with SHA-1 as the standard hash functions for US government use.

The three new functions have larger digest size, 256 bits for SHA-256, 384 bits for SHA-384, and 512 bits for SHA-512.

2.1.2 Outline of the primitive

SHA-256 is based on dividing the message into blocks of 512 bits each. Each block is then entered to the compression function, which expands the 512-bit block into 2048 bits, which are used to transform the current state value of 256 bits into the new state value.

The 256-bit initial value is loaded into eight 32-bit words. Then, for 64 rounds these words affect each other using the expanded message block and various functions. These 64 rounds can also be used in a block cipher mode (see SHACAL-2, Chapter 4 in part B (block ciphers) of the NESSIE portfolio).

SHA-384 and SHA-512 are based on dividing the message into blocks of 1024 bits each. Each block enters the compression function, and is used to affect the state of 512 bits through 80 rounds of the compression function.

Both SHA-384 and SHA-512 have a similar structure to the one used in SHA-256 but the word size is doubled and the functions are a little bit different. The only difference between SHA-384 and SHA-512 is the initial state and the fact that in SHA-384 the output is the state at the end truncated to 384 bits.

2.1.3 Security and performance

There are no reported attacks against any of the hash functions SHA-256, SHA-384 and SHA-512.

The performance figures for these hash functions show that they are very efficient. The software implementation of NESSIE achieved speeds of 30–70 cycles per byte for SHA-256, and 16–190 cycles per byte for SHA-384 and SHA-512. The simplicity of the operations used in all three submissions assures that hardware implementation will have low gate count. Moreover, SHA-256 can be used along with the encryption algorithm SHACAL-2 to save gates or the size of implementation.

We note that SHA-384 and SHA-512 are suited to environments where 64-bit registers (or longer ones) exist, as they operate on 64-bit words.

2.2 Description

All three hash functions share the same basic design. Each function appends bits to the message so that its length becomes a multiple of the block size the compression function deals with (512 bits for SHA-256, 1024 bits for SHA-384 and SHA-512).

Let l be the length of the message M which we want to hash. We append 1 followed by k zero bits, where k is the smallest non-negative value for which $l + 1 + k \equiv 448 \bmod 512$ in the case of SHA-256, and $l + 1 + k \equiv 896 \bmod 1024$ in the case of SHA-384 and SHA-512. To achieve a multiple of the block size the hash function deals with, we then add the length of the message ($\bmod 2^{64}$ for SHA-256 or $\bmod 2^{128}$ for SHA-384 and SHA-512).

Once this step is complete, we divide the message M into blocks of the same size (512 bits for SHA-256 and 1024 bits for SHA-384 and SHA-512). We denote this decomposition as $M = m_1 m_2 \ldots m_n$.

Now the same process is executed for either of the three hash functions (the constants and the exact functions might differ):

```
1. An initial hash value of 8 words is loaded into the 8 words: $A, B, C, D, E, F, G$ and $H$.
2. For $i = 1$ to $n$ do:
   – Set $AA = A, BB = B, CC = C, DD = D, EE = E, FF = F, GG = G$ and $HH = H$.
   – for $t = 0$ to 63 (79 for SHA-512 and SHA-384) do:
     $T_1 = H + \Sigma_1(E) + Ch(E, F, G) + K_t + W_t$
     $T_2 = \Sigma_0(A) + Maj(A, B, C)$
     $H = G$
     $G = F$
     $F = E$
     $E = D + T_1$
     $D = C$
     $C = B$
```

$B = A$
$A = T_1 + T_2$

- set $A = A + AA, B = B + BB, C = C + CC, D = D + DD, E = E + EE, F = F + FF, G = G + GG$ and $H = H + HH$.

3. Produce the hash value of $A|B|C|D|E|F|G|H$ for SHA-256 and SHA-512, and of $A|B|C|D|E|F$ for SHA-384.

Additions are performed modulo 2^{32} for SHA-256 and modulo 2^{64} for SHA-384 and SHA-512.

The K_t's are round constants added at each round of the compression function. The W_t's are a set of 64 or 80 words derived from the message block being processed. The $Ch(x, y, z)$ and $Maj(x, y, z)$ are the only non-linear functions used in the compression function. Σ_0 and Σ_1 are two linear functions which differ for SHA-256 and for SHA-512 and SHA-384.

Each block M_i is divided into 16 words $m_i^0, m_i^1, \ldots m_i^{15}$, then the W_t's are computed:

$$W_t = \begin{cases} m_i^t & 0 \leq t \leq 15 \\ s_1(W_{t-2}) + W_{t-7} + s_0(W_{t-15}) + W_{t-16} & 16 \leq t \leq 63 \text{(or 79)} \end{cases}$$

The s_0 and s_1 functions are linear functions, but are different for SHA-256 and for SHA-512 and SHA-384.

2.2.1 The Functions Used in SHA-256

SHA-256 uses 6 functions: $Ch(x, y, z)$ and $Maj(x, y, z)$, two non-linear functions with 96-bit input and 32-bit output, and 4 linear functions with 32-bit input and 32-bit output: $\Sigma_0, \Sigma_1, s_0, s_1$.

We first define the 4 linear operations:

$$\begin{aligned} \Sigma_0(X) &= X \ggg_2 \oplus X \ggg_{13} \oplus X \ggg_{22} \\ \Sigma_1(X) &= X \ggg_6 \oplus X \ggg_{11} \oplus X \ggg_{25} \\ \sigma_0(X) &= X \ggg_7 \oplus X \ggg_{18} \oplus X \gg_3 \\ \sigma_1(X) &= X \ggg_{17} \oplus X \ggg_{19} \oplus X \gg_{10} \end{aligned}$$

The Ch operation accepts three 32-bit words X, Y, Z. The operation is a bitwise choose function, i.e., if the i'th bit of X is set, then the i'th bit of the output is the i'th bit of Y. Otherwise, the i'th bit of the output is the i'th bit of Z. This operation can be efficiently implemented as:

$$Ch(X, Y, Z) = (X \wedge Y) \oplus (\overline{X} \wedge Z)$$

The Maj operation also accepts three 32-bit words X, Y, Z. The operation is a bitwise majority function, i.e., if the majority of the i'th bits of X, Y and

Z are set, then the i'th output bit is set, and vice versa. This operation can be efficiently implemented as:

$$Maj(X,Y,Z) \quad = \quad (X \wedge Y) \oplus (X \wedge Z) \oplus (Y \wedge Z)$$

2.2.2 The Constants of SHA-256

Each round, a round constant is used in the compression function. We list the round constants in the following table:

Round i	K_i	Round i	K_i	Round i	K_i	Round i	K_i
0	428A2F98$_x$	1	71374491$_x$	2	B5C0FBCF$_x$	3	E9B5DBA5
4	3956C25B$_x$	5	59F111F1$_x$	6	923F82A4$_x$	7	AB1C5ED5
8	D807AA98$_x$	9	12835B01$_x$	10	243185BE$_x$	11	550C7DC3
12	72BE5D74$_x$	13	80DEB1FE$_x$	14	9BDC06A7$_x$	15	C19BF174
16	E49B69C1$_x$	17	EFBE4786$_x$	18	0FC19DC6$_x$	19	240CA1CC
20	2DE92C6F$_x$	21	4A7484AA$_x$	22	5CB0A9DC$_x$	23	76F988DA
24	983E5152$_x$	25	A831C66D$_x$	26	B00327C8$_x$	27	BF597FC7
28	C6E00BF3$_x$	29	D5A79147$_x$	30	06CA6351$_x$	31	14292967
32	27B70A85$_x$	33	2E1B2138$_x$	34	4D2C6DFC$_x$	35	53380D13
36	650A7354$_x$	37	766A0ABB$_x$	38	81C2C92E$_x$	39	92722C85
40	A2BFE8A1$_x$	41	A81A664B$_x$	42	C24B8B70$_x$	43	C76C51A3
44	D192E819$_x$	45	D6990624$_x$	46	F40E3585$_x$	47	106AA070
48	19A4C116$_x$	49	1E376C08$_x$	50	2748774C$_x$	51	34B0BCB5
52	391C0CB3$_x$	53	4ED8AA4A$_x$	54	5B9CCA4F$_x$	55	682E6FF3
56	748F82EE$_x$	57	78A5636F$_x$	58	84C87814$_x$	59	8CC70208
60	90BEFFFA$_x$	61	A4506CEB$_x$	62	BEF9A3F7$_x$	63	C67178F2

At the beginning of the hash computation the words A, B, C, D, E, F, G and H are loaded with an initial value. For SHA-256 the following values are loaded:

$$\begin{aligned} A &= \texttt{6A09E667}_x \\ B &= \texttt{BB67AE85}_x \\ C &= \texttt{3C6EF372}_x \\ D &= \texttt{A54FF53A}_x \\ E &= \texttt{519E527F}_x \\ F &= \texttt{9B05688C}_x \\ G &= \texttt{1F83D9AB}_x \\ H &= \texttt{5BE0CD19}_x \end{aligned}$$

2.2.3 The Functions Used in SHA-384 and SHA-512

SHA-512 and SHA-384 uses 6 functions: $Ch(x, y, z)$ and $Maj(x, y, z)$, two non-linear functions with 192-bit input and 64-bit output, and 4 linear functions with 64-bit input and 64-bit output: $\Sigma_0, \Sigma_1, s_0, s_1$.

We first define the 4 linear operations:

$$\begin{aligned}
\Sigma_0(X) &= X \ggg_{28} \oplus X \ggg_{34} \oplus X \ggg_{39} \\
\Sigma_1(X) &= X \ggg_{14} \oplus X \ggg_{18} \oplus X \ggg_{41} \\
\sigma_0(X) &= X \ggg_{1} \oplus X \ggg_{8} \oplus X \gg_{7} \\
\sigma_1(X) &= X \ggg_{19} \oplus X \ggg_{19} \oplus X \gg_{6}
\end{aligned}$$

The Ch operation accepts three 64-bit words X, Y, Z. The operation is a bitwise choose function, i.e., if the i'th bit of X is set, then the i'th bit of the output is the i'th bit of Y. Otherwise, the i'th bit of the output is the i'th bit of Z. This operation can be efficiently implemented as:

$$Ch(X, Y, Z) = (X \wedge Y) \oplus (\overline{X} \wedge Z)$$

The Maj operation also accepts three 64-bit words X, Y, Z. The operation is a bitwise majority function, i.e., if the majority of the i'th bits of X, Y and Z are set, then the i'th output bit is set, and vice versa. This operation can be efficiently implemented as:

$$Maj(X, Y, Z) = (X \wedge Y) \oplus (X \wedge Z) \oplus (Y \wedge Z)$$

2.2.4 The Constants of SHA-384 and SHA-512

Each round, a round constant is used in the compression function. We list the round constants in the following table:

Round i	K_i	Round i	K_i	Round i	K_i
0	428A2F98D728AE22$_\mathrm{x}$	1	7137449123EF65CD$_\mathrm{x}$	2	B5C0FBCFEC4
3	E9B5DBA58189DBBC$_\mathrm{x}$	4	3956C25BF348B538$_\mathrm{x}$	5	59F111F1B60
6	923F82A4AF194F9B$_\mathrm{x}$	7	AB1C5ED5DA6D8118$_\mathrm{x}$	8	D807AA98A30
9	12835B0145706FBE$_\mathrm{x}$	10	243185BE4EE4B28C$_\mathrm{x}$	11	550C7DC3D5F
12	72BE5D74F27B896F$_\mathrm{x}$	13	80DEB1FE3B1696B1$_\mathrm{x}$	14	9BDC06A725C
15	C19BF174CF692694$_\mathrm{x}$	16	E49B69C19EF14AD2$_\mathrm{x}$	17	EFBE4786384
18	0FC19DC68B8CD5B5$_\mathrm{x}$	19	240CA1CC77AC9C65$_\mathrm{x}$	20	2DE92C6F592
21	4A7484AA6EA6E483$_\mathrm{x}$	22	5CB0A9DCBD41FBD4$_\mathrm{x}$	23	76F988DA831
24	983E5152EE66DFAB$_\mathrm{x}$	25	A831C66D2DB43210$_\mathrm{x}$	26	B00327C898F
27	BF597FC7BEEF0EE4$_\mathrm{x}$	28	C6E00BF33DA88FC2$_\mathrm{x}$	29	D5A79147930
30	06CA6351E003826F$_\mathrm{x}$	31	142929670A0E6E70$_\mathrm{x}$	32	27B70A8546D
33	2E1B21385C26C926$_\mathrm{x}$	34	4D2C6DFC5AC42AED$_\mathrm{x}$	35	53380D139D9
36	650A73548BAF63D2$_\mathrm{x}$	37	766A0ABB3C77B2A8$_\mathrm{x}$	38	81C2C92E47E
39	92722C851482353B$_\mathrm{x}$	40	A2BFE8A14CF10364$_\mathrm{x}$	41	A81A664BBC4
42	C24B8B70D0F89791$_\mathrm{x}$	43	C76C51A30654BE30$_\mathrm{x}$	44	D192E819D6E
45	D69906245565A910$_\mathrm{x}$	46	F40E35855771202A$_\mathrm{x}$	47	106AA07032B
48	19A4C116B8D2D0C8$_\mathrm{x}$	49	1E376C085141AB53$_\mathrm{x}$	50	2748774CDF8
51	34B0BCB5E19B48A8$_\mathrm{x}$	52	391C0CB3C5C95A63$_\mathrm{x}$	53	4ED8AA4AE34
54	5B9CCA4F7763E373$_\mathrm{x}$	55	682E6FF3D6B2B8A3$_\mathrm{x}$	56	748F82EE5DE
57	78A5636F43172F60$_\mathrm{x}$	58	84C87814A1F0AB72$_\mathrm{x}$	59	8CC702081A6
60	90BEFFFA23631E28$_\mathrm{x}$	61	A4506CEBDE82BDE9$_\mathrm{x}$	62	BEF9A3F7B2C
63	C67178F2E372532B$_\mathrm{x}$	64	CA273ECEEA26619C$_\mathrm{x}$	65	D186B8C721C
66	EADA7DD6CDE0EB1E$_\mathrm{x}$	67	F57D4F7FEE6ED178$_\mathrm{x}$	68	06F067AA721
69	0A637DC5A2C898A6$_\mathrm{x}$	70	113F9804BEF90DAE$_\mathrm{x}$	71	1B710B35131
72	28DB77F523047D84$_\mathrm{x}$	73	32CAAB7B40C72493$_\mathrm{x}$	74	3C9EBE0A15C
75	431D67C49C100D4C$_\mathrm{x}$	76	4CC5D4BECB3E42B6$_\mathrm{x}$	77	597F299CFC6
78	5FCB6FAB3AD6FAEC$_\mathrm{x}$	79	6C44198C4A475817$_\mathrm{x}$		

At the beginning of the hash computation the words A, B, C, D, E, F, G and H are loaded with an initial value. For SHA-384 the following values are loaded:

$$\begin{aligned}
A &= \texttt{CBBB9D4DC1059ED8}_\mathrm{x}\\
B &= \texttt{629A292A367CD507}_\mathrm{x}\\
C &= \texttt{9159015A3070DD17}_\mathrm{x}\\
D &= \texttt{152FECD8F70E5939}_\mathrm{x}\\
E &= \texttt{67332667FFC00B31}_\mathrm{x}\\
F &= \texttt{8EB44A8768581511}_\mathrm{x}\\
G &= \texttt{DB0C2E0D64F98FA7}_\mathrm{x}\\
H &= \texttt{47B5481DBEFA4FA4}_\mathrm{x}
\end{aligned}$$

For SHA-512 the following values are loaded:

$$
\begin{aligned}
A &= \texttt{6A09E667F3BCC908}_{\texttt{x}}\\
B &= \texttt{BB67AE8584CAA73B}_{\texttt{x}}\\
C &= \texttt{3C6EF372FE94F82B}_{\texttt{x}}\\
D &= \texttt{A54FF53A5F1D36F1}_{\texttt{x}}\\
E &= \texttt{510E527FADE682D1}_{\texttt{x}}\\
F &= \texttt{9B05688C2B3E6C1F}_{\texttt{x}}\\
G &= \texttt{1F83D9ABFB41BD6B}_{\texttt{x}}\\
H &= \texttt{5BE0CD19137E2179}_{\texttt{x}}
\end{aligned}
$$

2.3 Test vectors

Part D

Message Authentication Codes

1. UMAC

1.1 Introduction

1.1.1 Overview

UMAC is a message authentication code submitted to NESSIE by Ted Krovetz, John Black, Shai Halevi, Hugo Krawczyk and Phillip Rogaway [379]. The design of UMAC is based on families of universal hash functions, and it uses the AES block cipher (described in Part B of the NESSIE portfolio) as a component. Many options are available in an implementation of UMAC, resulting in different versions of the algorithm, but in the descriptions provided in this chapter we will focus on two specific versions named UMAC32 and UMAC16.

1.1.2 Outline of the primitive

The UMAC message authentication code maps a message M consisting of an arbitrary number of bits onto a fixed length output under control of a secret key K. An unusual feature of UMAC is that the authentication also depends on a nonce (in addition to depending on the message and key). This nonce is a value that should not be repeated when authenticating messages under the same key (it is usually implemented by a counter). Note that the nonce does not have to be secret, it may be transmitted along with the message.

The design of UMAC is based on a universal hash function family, named UHASH. The procedure of authentication can be summarised by the following formula:

$$\mathsf{UMAC}(K, M, Nonce) = \mathsf{UHASH}(K, M) \oplus \mathsf{PDF}(K, Nonce)\,.$$

That is, UHASH is used to compress the message M to a fixed length, and a pad derivation function (PDF) is used to transform the nonce into a pad value of the same length. The message authentication value is then computed by bitwise addition (exclusive-or) of the compressed message and the pad value. Note that both UHASH and the PDF function are keyed: UHASH uses a key to choose a particular hash function from the universal hash family; PDF uses a key for internal encryption with the AES block cipher. A number of subkeys need to be generated from the user key K and this is done by means of a key derivation function.

UHASH itself consists of three seperate layers, where each layer is based on a different universal hash function family:

1. Compression: The first layer uses the fast NH hash family to compress the message by a fixed ratio.
2. Hash-to-fixed-length: The second layer uses a polynomial hash family, which generates an output of a fixed length.
3. Strengthen-and-fold: The third layer uses an inner-product hash family, which reduces the length of its input to a more appropriate size.

1.1.3 Security and performance

There is a security proof for UMAC. The core of the design, the UHASH function, does not depend on any cryptographic assumptions, and its strength is specified by a purely mathematical property stated in terms of collision probability (this property is proven in an absolute sense). Therefore, the security of UMAC depends on the cryptographic function, AES, that is used for the derivation of key material and for the computation of the pad value. This means that an attack that breaks UMAC (finding a forgery with probability significantly higher than the established collision probability) would lead to an attack of comparable complexity that breaks AES (in the sense of distinguishing AES from a family of random permutations).

Because UHASH relies on simple operations (additions and multiplications of 16-bit, 32-bit and 64-bit numbers), the UMAC algorithm is very efficient in software implementations, especially when it is used to authenticate long data streams. Several parameters can be chosen for UMAC in order to optimise an implementation for a particular platform. For maximum speed, the environment should provide sufficient space for the storage of internal keys of approximately 1.5 KB and it should support the multiplication of 32-bit operands into a 64-bit result. Note that the efficiency of UMAC comes at the cost of a greater complexity (and less compact implementations) compared to other MAC algorithms. Also, the procedure for key-setup consumes significant time so UMAC is not suited for applications where the key needs to be changed frequently (i.e., where each key is used to authenticate only a small amount of data).

1.2 Description

As noted in the introduction to this chapter many options are available in the implementation of UMAC and this is reflected by a number of parameters, where each choice for this set of parameters results in a different version of the algorithm. However, two named parameter sets have been specified, UMAC32 and UMAC16. The parameters for these two versions have been chosen to give good efficiency on a wide variety of platforms. The main difference between the two versions is in the word length which is equal to 32 bits for UMAC32 and 16 bits for UMAC16. In the following we give detailed specifications for both UMAC32 and UMAC16. The default output length of these algorithms is equal to 64 bits, but we will generally describe how to produce outputs of length equal to 32, 64, 96 or 128 bits in order to provide different security levels. We conclude with a discussion on the parameters of UMAC and other versions of the algorithm.

1.2.1 Definitions and Notations

In this section we introduce some notations specific to this chapter. See also Part A of the NESSIE portfolio for generic notations used in this book.

Operations on integers.

- $\mathsf{prime}(x)$ is the largest prime smaller than 2^x

Operations on strings.

- $S[i]$ is the i-th bit of the string S (indices begin at 1)
- $S[i \dots j]$ is the substring of S consisting of bits i through j
- $\mathsf{zeroes}(n)$ is the string made of n 0-bits
- $\mathsf{ones}(n)$ is the string made of n 1-bits
- $\mathsf{padzero}(S, w)$ is the string $S \,\|\, \mathsf{zeroes}(n)$ where n is the smallest number so that the bitlength of $S \,\|\, \mathsf{zeroes}(n)$ is divisible by w
- $\mathsf{padonezero}(S, w)$ is the string $S \,\|\, \mathsf{ones}(1) \,\|\, \mathsf{zeroes}(n)$ where n is the smallest number so that the bitlength of $S \,\|\, \mathsf{ones}(1) \,\|\, \mathsf{zeroes}(n)$ is divisible by w
- $\mathsf{bytereverse}(S, w)$ computes a new w-bit string from the original w-bit string S, by reversing the order of the bytes. That is, if $S = S_1 \,\|\, \cdots \,\|\, S_{w/8}$ then $\mathsf{bytereverse}(S, w) = S_{w/8} \,\|\, \cdots \,\|\, S_1$, where the bitsize of S is w and the bitsize of each S_i is 8.

Conversions between strings and integers.

- $\mathsf{str2uint}(S)$ is the non-negative integer x such that the binary representation of x corresponds to the string S. More formally, if S is n bits long then $\mathsf{str2uint}(S) = S[1] \times 2^{t-1} + S[2] \times 2^{t-2} + \cdots + S[t-1] \times 2^1 + S[t]$
- $\mathsf{uint2str}(x, w)$ is the w-bit string S such that $\mathrm{str2uint}(S) = x$
- $\mathsf{str2sint}(S)$ is the integer x such that the binary representation of x in two's complement corresponds to the string S. More formally, if S is n bits long then $\mathsf{str2sint}(S) = -S[1] \times 2^{t-1} + S[2] \times 2^{t-2} + \cdots + S[t-1] \times 2^1 + S[t]$
- $\mathsf{sint2str}(x, w)$ is the w-bit string S such that $\mathrm{str2sint}(S) = x$

Mathematical operations on strings.

- For UMAC32 we interpret
  - $S +_w T$ as $\mathsf{uint2str}(\mathsf{str2uint}(S) + \mathsf{str2uint}(T) \bmod 2^w, w)$
  - $S \times_w T$ as $\mathsf{uint2str}(\mathsf{str2uint}(S) \times \mathsf{str2uint}(T) \bmod 2^w, w)$
- For UMAC16 we interpret
  - $S +_w T$ as $\mathsf{uint2str}(\mathsf{str2sint}(S) + \mathsf{str2sint}(T) \bmod 2^w, w)$
  - $S \times_w T$ as $\mathsf{uint2str}(\mathsf{str2sint}(S) \times \mathsf{str2sint}(T) \bmod 2^w, w)$

1.2.2 Encryption and Key derivation functions

In this section we describe the encryption and key derivation functions that are used by both UMAC32 and UMAC16.

Encryption function: AES.
The block cipher AES is used as encryption function. AES takes a plaintext T and key K, both strings of 128 bits, to compute a ciphertext string $Y = \mathsf{AES}(K, T)$ of 128 bits. Note that in general other key lengths are possible for AES (192 or 256 bits) but the default key length specified in UMAC32 and UMAC16 is 128 bits. For a description of AES we refer to Part B of the NESSIE portfolio.

Key derivation function: KDF.
We describe below the key derivation function that is used. This function expands the user-supplied key into the subkeys used internally by UMAC32 and UMAC16. It uses the AES cipher in output-feedback mode to produce the required pseudorandom bits. Note that the *index* parameter determines the initial plaintext. Using the same key but different values for *index* generates different pseudorandom outputs.

```
Description:
    Input:    a string $K$ of 128 bits
              an integer $index$, with $0 \leq index < 256$
              a positive integer $Subkeylen$
    Output:   a string $Y$ of $Subkeylen$ bits

    1. Let $t := \lceil Subkeylen/128 \rceil$
    2. Set $T := \mathsf{zeroes}(120) \parallel \mathsf{uint2str}(index, 8)$
    3. Let the initial value of $Y$ be the empty string
    4. For $i := 1$ to $t$ do the following steps
         a) $T := \mathsf{AES}(K, T)$
         b) $Y := Y \parallel T$
    5. $Y := Y[1 \ldots Subkeylen]$
    6. Return the final value of $Y$
```

1.2.3 UMAC32

UMAC32 is a universal hash based message authentication code with 32-bit word length. It is targeted to processors with good 32- and 64-bit support. We give a full specification of the algorithm, where we first describe the individual components and conclude by showing how all these components fit together in the UMAC32 construction. Some readers may prefer to read these sections backwards, in order to get a top-down description.

1.2.3.1 Components of UHASH32

In this section we describe the components of UHASH32. There are three levels of hashing in UHASH32, each of them based on a different universal hash family.

NH hash with 32-bit wordsize: NH32.
We describe below a universal hash family that hashes an input string M using a key K by considering M and K to be arrays of 32-bit integers, and performing a sequence of arithmetic operations on them. Note that in order to accommodate processors with small-scale vector parallelism, NH32 accesses data-words in pairs which are four words apart.

Description:

```
Input:   a string $K$ of 8192 bits
         a string $M$, with $Mlen \le 8192$ and divisible by 256
Output:  a string $Y$ of 64 bits

1. Divide $M$ and $K$ into substrings of 32 bits, according to little-endian convention
   a) Set $t := Mlen/32$ (number of 32-bit words in $M$)
   b) Let $M_1, \ldots, M_t$ be strings of 32 bits such that $M = M_1 \parallel \cdots \parallel M_t$
   c) For $i = 1$ to $t$ do $M_i := \mathsf{bytereverse}(M_i, 32)$
   d) Let $K_1, K_2, \ldots, K_t$ be strings of 32 bits such that $K[1 \ldots Mlen] = K_1 \parallel K_2 \parallel \cdots \parallel K_t$
   e) For $i = 1$ to $t$ do $K_i := \mathsf{bytereverse}(K_i, 32)$
2. Set the initial value of $Y := 0$
3. Process the substrings $M_i$ and $K_i$ $(1 \le i \le t)$
   a) Set $i := 1$
   b) While $i < t$ do the following five steps
        i. $Y := Y +_{64} ((M_{i+0} +_{32} K_{i+0}) \times_{64} (M_{i+4} +_{32} K_{i+4}))$
       ii. $Y := Y +_{64} ((M_{i+1} +_{32} K_{i+1}) \times_{64} (M_{i+5} +_{32} K_{i+5}))$
      iii. $Y := Y +_{64} ((M_{i+2} +_{32} K_{i+2}) \times_{64} (M_{i+6} +_{32} K_{i+6}))$
       iv. $Y := Y +_{64} ((M_{i+3} +_{32} K_{i+3}) \times_{64} (M_{i+7} +_{32} K_{i+7}))$
        v. $i := i + 8$
4. Return the final value of $Y$
```

First-layer hash: L1HASH32.

We describe below the first level of hashing in UHASH32. It is based on the NH32 hash function. This function requires a key which is just as long as the message being hashed. Therefore, in order to limit the amount of key material that is needed, we use a key of fixed length (8192 bits) and process the message in blocks of this length (or shorter). Each block is hashed with NH32 and length information is included. The results are concatenated to form the output of L1HASH32. Note that the blocks of 8192 bits are compressed to 64-bit strings, which corresponds to a compression ratio of 128.

Description:

```
Input:   a string $K$ of 8192 bits
         a string $M$ of $Mlen$ bits, with $Mlen < 2^{67}$
Output:  a string $Y$ of $64 \times \lceil Mlen/8192 \rceil$ bits

1. Divide $M$ into substrings of 8192 bits (the final substring may be shorter)
   a) Set $t := \lceil Mlen/8192 \rceil$
   b) Let $M_1, M_2, \ldots, M_t$ be strings such that $M = M_1 \parallel M_2 \parallel \cdots \parallel M_t$, where the length of $M_i$ is 8192 bits for $1 \le i < t$ and the length of $M_t$ is $tlen$ bits with $tlen \le 8192$
2. Let the initial value of $Y$ be the empty string
3. Process the substrings $M_i$ $(1 \le i < t)$
   a) $Len :=$ uint2str(8192,64)
```

```
    b) For $i := 1$ to $(t-1)$ do $Y := Y \,\|\, (\mathsf{NH32}(K, M_i) +_{64} Len)$
4. Process the final substring $M_t$
    a) $Len :=$ uint2str($tlen$,64)
    b) $M_t :=$ padzero($M_t$,256)
    c) $Y := Y \,\|\, (\mathsf{NH32}(K, M_t) +_{64} Len)$
5. Return the final value of $Y$
```

Polynomial hash with 64-bit wordsize: POLY64.

We describe below a polynomial hashing scheme which treats an input message as a sequence of coefficients of a polynomial, and the key as the point at which this polynomial is evaluated. The largest prime smaller than 2^{64} is used as modulus, and the message is divided in 64-bit words. Words that are larger than $2^{64} - 2^{32}$ are split into two words. This guarantees that all words are smaller than the prime modulus.

Description:

```
Input:   an integer $k$,with $0 \le k \le \mathsf{prime}(64) - 1$
         a string $M$ of $Mlen$ bits, with $Mlen$ divisible by 64
Output:  an integer $y$, with $0 \le y \le \mathsf{prime}(64) - 1$

1. Set $p := \mathsf{prime}(64) = 2^{64} - 59$
2. Divide $M$ into substrings of 64 bits
    a) Set $t := Mlen/64$
    b) Let $M_1, \ldots, M_t$ be strings of 64 bits such that $M = M_1 \,\|\, \cdots \,\|\, M_t$
3. Set the initial value of $y := 1$
4. Process the substrings $M_i$. That is, for $i := 1$ to $t$ do
    a) $m := \mathsf{str2uint}(M_i)$
    b) If $(m < 2^{64} - 2^{32})$ then $y := (k \times y + m) \bmod p$
       else do the following two steps
         i. $y := (k \times y + (p-1)) \bmod p$
        ii. $y := (k \times y + (m - 59)) \bmod p$
5. Return the final value of $y$
```

Polynomial hash with 128-bit wordsize: POLY128.

We describe below a 128-bit variant of the polynomial hashing scheme. The largest prime smaller than 2^{128} is used as modulus, and the message is divided in 128-bit words. Words that are larger than $2^{128} - 2^{96}$ are split into two words. This guarantees that all words are smaller than the prime modulus.

Description:

```
Input:   an integer $k$,with $0 \le k \le \mathsf{prime}(128) - 1$
         a string $M$ of $Mlen$ bits, with $Mlen$ divisible by 128
Output:  an integer $y$, with $0 \le y \le \mathsf{prime}(128) - 1$

1. Set $p := \mathsf{prime}(128) = 2^{128} - 159$
2. Divide $M$ into substrings of 128 bits
    a) Set $t := Mlen/128$
    b) Let $M_1, \ldots, M_t$ be strings of 128 bits such that $M = M_1 \,\|\, \cdots \,\|\, M_t$
```

```
3. Set the initial value of $y := 1$
4. Process the substrings $M_i$. That is, for $i := 1$ to $t$ do
   a) $m := \mathsf{str2uint}(M_i)$
   b) If $(m < 2^{128} - 2^{96})$ then $y := (k \times y + m) \bmod p$
      else do the following two steps
       i. $y := (k \times y + (p - 1)) \bmod p$
      ii. $y := (k \times y + (m - 159)) \bmod p$
5. Return the final value of $y$
```

Second-layer hash: L2HASH32.
We describe below the second level of hashing in UHASH32. It is based on the POLY64 and POLY128 hash functions. The security guarantee of a polynomial hashing scheme degrades linearly in the length of the message being hashed: if two messages of n words are hashed the collision probability is no more than n/p (where p denotes the prime modulus that is used). The scheme described below hashes n words under the modulus $\mathsf{prime}(64)$ until $n/\mathsf{prime}(64)$ reaches a certain bound. Then the result obtained so far is prepended to the remaining message and hashing continues under the modulus $\mathsf{prime}(128)$ which is substantially larger than $\mathsf{prime}(64)$. Note that the dynamic use of POLY64 and POLY128 gives good performance for short messages while still accomodating longer ones. The keys used for the polynomial hashing are restricted to particular subsets to allow for optimisations (some potential arithmetic carries can be disregarded during the computation). L2HASH32 produces an output of a fixed length of 128 bits.

Description:

```
Input:   a string $K$ of 192 bits
         a string $M$ of $Mlen$ bits, with $Mlen < 2^{67}$
Output:  a string $Y$ of 128 bits

1. Extract key values and restrict them to special key-sets
   a) Let $K_{64}$ be a string of 64 bits, and let $K_{128}$ be a string of 128 bits,
      such that $K = K_{64} \,\|\, K_{128}$
   b) Define two strings:
      $Mask_{64} := \mathsf{uint2str}(\mathtt{01FFFFFF01FFFFFF}_{\mathtt{x}}, 64)$
      $Mask_{128} := \mathsf{uint2str}(\mathtt{01FFFFFF01FFFFFF01FFFFFF01FFFFFF}_{\mathtt{x}}, 128)$
   c) Compute two integers:
      $k_{64} := \mathsf{str2uint}(K_{64} \wedge Mask_{64})$
      $k_{128} := \mathsf{str2uint}(K_{128} \wedge Mask_{128})$
2. If $Mlen \le 2^{20}$ then $y := \mathsf{POLY64}(k_{64}, M)$
   else do the following four steps
   a) Let $M_1$ be a string of $2^{20}$ bits, and let $M_2$ be a string of less than
      $2^{67} - 2^{20}$ bits, such that $M = M_1 \,\|\, M_2$
   b) $M_2 := \mathsf{padonezero}(M_2, 128)$
   c) $y := \mathsf{POLY64}(k_{64}, M_1)$
   d) $y := \mathsf{POLY128}(k_{128}, \mathsf{uint2str}(y, 128) \,\|\, M_2)$
3. Return $Y := \mathsf{uint2str}(y, 128)$
```

Third-layer hash: L3HASH32.

We describe below the third level of hashing in UHASH32. It hashes a 128-bit input to a fixed length of 32 bits using a simple inner-product hash with affine translation. A 36-bit prime modulus is used to improve security.

Description:

```
Input:   a string $K$ of 512 bits
         a string $T$ of 32 bits
         a string $M$ of 128 bits
Output:  a string $Y$ of 32 bits

1. Set $p := \mathsf{prime}(36) = 2^{36} - 5$
2. Divide $M$ and $K$ into substrings of 16 bits and 64 bits respectively
   a) Let $M_1, \ldots, M_8$ be strings of 16 bits such that $M = M_1 \parallel \cdots \parallel M_8$
   b) Let $K_1, \ldots, K_8$ be strings of 64 bits such that $K = K_1 \parallel \cdots \parallel K_8$
3. Convert the substrings into numbers. That is, for $i := 1$ to 8 do
   a) $m_i := \mathsf{str2uint}(M_i)$
   b) $k_i := \mathsf{str2uint}(K_i) \bmod p$
4. Compute the output as follows
   a) $y := ((m_1 \times k_1 + m_2 \times k_2 + \cdots + m_8 \times k_8) \bmod p) \bmod 2^{32}$
   b) Return $Y := \mathsf{uint2str}(y, 32) \oplus T$
```

1.2.3.2 Three-layer hashing scheme: UHASH32

We describe below the universal hash function UHASH32 that is used by UMAC32 to compress the message input to a result of a fixed length. The components described in the previous section are combined in a straightforward manner. A message is first hashed with L1HASH32, its output is then hashed with L2HASH32, whose output is then hashed with L3HASH32. If the message is no longer than 8192 bits to begin with, L2HASH32 is skipped as an optimisation. Because the output of L3HASH32 is only 32 bits long, multiple iterations of the three-layer hash scheme are used with different keys each time. All of these subkeys are derived from the user-supplied key with the key derivation function KDF. To reduce memory requirements L1HASH32 reuses most of its key material between iterations.

Description:

```
Input:   a string $K$ of 128 bits
         a string $M$ of $Mlen$ bits, with $Mlen < 2^{67}$
         an integer $Outlen$ = 32, 64, 96, or 128
Output:  a string $Y$ of $Outlen$ bits

1. Set $t := Outlen/32$
2. Derive the needed key material from $K$
   a) $K1 := \mathsf{KDF}(K, 0, 8192 + (t-1) \times 128)$
   b) $K2 := \mathsf{KDF}(K, 1, t \times 192)$
   c) $K3 := \mathsf{KDF}(K, 2, t \times 512)$
   d) $T3 := \mathsf{KDF}(K, 3, t \times 32)$
```

```
3. Let the initial value of $Y$ be the empty string
4. Iterate the three-layer hashing scheme. That is, for $i := 1$ to $t$ do
   a) Define the subkeys to be used in this iteration
        i. $K1_i := K1[(i-1) \times 128 + 1 \ldots (i-1) \times 128 + 8192]$
       ii. $K2_i := K2[(i-1) \times 192 + 1 \ldots i \times 192]$
      iii. $K3_i := K3[(i-1) \times 512 + 1 \ldots i \times 512]$
       iv. $T3_i := T3[(i-1) \times 32 + 1 \ldots i \times 32]$
   b) $A := \mathsf{L1HASH32}(K1_i, M)$
   c) If $Mlen \leq 8192$ then $B := \mathsf{zeroes}(64) \parallel A$
      else $B := \mathsf{L2HASH32}(K2_i, A)$
   d) $C := \mathsf{L3HASH32}(K3_i, T3_i, B)$
   e) $Y := Y \parallel C$
5. Return the final value of $Y$
```

1.2.3.3 Message authentication code: UMAC32

The UMAC32 algorithm computes a message authentication value when given a key, message and nonce. It uses UHASH32 to compress the message and AES to encrypt the nonce. The exclusive-or (XOR) of the two resulting strings forms the output message authentication value. In the description below we consider output lengths of 32, 64, 96 or 128 bits, determined by the parameter *Outlen*.

Description:

```
Input:   a string $K$ of 128 bits
         a string $M$ of $Mlen$ bits, with $Mlen < 2^{67}$
         a string $Nonce$ of $Nlen$ bits, with $8 \leq Nlen \leq 128$
                    and $Nlen$ divisible by 8
         an integer $Outlen$ = 32, 64, 96, or 128
Output:  a string $Y$ of $Outlen$ bits

1. $Hash := \mathsf{UHASH32}(K, M, Outlen)$
2. $K_{Enc} := \mathsf{KDF}(K, 128, 128)$
3. $Nonce := Nonce \parallel \mathsf{zeroes}(128 - Nlen)$
4. $Pad := \mathsf{AES}(K_{Enc}, Nonce)$
5. $Pad := Pad[1 \ldots Outlen]$
6. Return $Y := Hash \oplus Pad$
```

Reuse of the encrypted nonce for short output lengths. When the parameter *Outlen* in the description above is equal to 32 or 64 bits, the AES output $Pad := \mathsf{AES}(K_{Enc}, Nonce)$ can be reused for several authentications (using a different substring of the AES output each time). This reduces the average time spent by AES for each authentication. We refer to [379] for more information on this optimisation.

1.2.4 UMAC16

UMAC16 is a universal hash based message authentication code with 16-bit word length. It is targeted to processors with good 16- and 32-bit support. We give a full specification of the algorithm, where we first describe the individual components and conclude by showing how all these components fit together in the UMAC16 construction. Some readers may prefer to read these sections backwards, in order to get a top-down description.

1.2.4.1 Components of UHASH16

In this section we describe the components of UHASH16. There are three levels of hashing in UHASH16, each of them based on a different universal hash family.

NH hash with 16-bit wordsize: NH16.
We describe below a universal hash family that hashes an input string M using a key K by considering M and K to be arrays of 16-bit integers, and performing a sequence of arithmetic operations on them. Note that in order to accommodate processors with small-scale vector parallelism, NH16 accesses data-words in pairs which are eight words apart.

Description:

```
Input:   a string $K$ of 8192 bits
         a string $M$, with $Mlen \leq 8192$ and divisible by 256
Output:  a string $Y$ of 32 bits

1. Divide $M$ and $K$ into substrings of 16 bits, according to little-endian
   convention
   a) Set $t := Mlen/16$ (number of 16-bit words in $M$)
   b) Let $M_1, \ldots, M_t$ be strings of 16 bits
      such that $M = M_1 \parallel \cdots \parallel M_t$
   c) For $i = 1$ to $t$ do $M_i := \mathsf{bytereverse}(M_i, 16)$
   d) Let $K_1, \ldots, K_t$ be strings of 16 bits
      such that $K[1 \ldots Mlen] = K_1 \parallel \cdots \parallel K_t$
   e) For $i = 1$ to $t$ do $K_i := \mathsf{bytereverse}(K_i, 16)$
2. Set the initial value of $Y := 0$
3. Process the substrings $M_i$ and $K_i$ $(1 \leq i \leq t)$
   a) Set $i := 1$
   b) While $i < t$ do the following nine steps
         i. $Y := Y +_{32} ((M_{i+0} +_{16} K_{i+0}) \times_{32} (M_{i+8} +_{16} K_{i+8}))$
        ii. $Y := Y +_{32} ((M_{i+1} +_{16} K_{i+1}) \times_{32} (M_{i+9} +_{16} K_{i+9}))$
       iii. $Y := Y +_{32} ((M_{i+2} +_{16} K_{i+2}) \times_{32} (M_{i+10} +_{16} K_{i+10}))$
        iv. $Y := Y +_{32} ((M_{i+3} +_{16} K_{i+3}) \times_{32} (M_{i+11} +_{16} K_{i+11}))$
         v. $Y := Y +_{32} ((M_{i+4} +_{16} K_{i+4}) \times_{32} (M_{i+12} +_{16} K_{i+12}))$
        vi. $Y := Y +_{32} ((M_{i+5} +_{16} K_{i+5}) \times_{32} (M_{i+13} +_{16} K_{i+13}))$
       vii. $Y := Y +_{32} ((M_{i+6} +_{16} K_{i+6}) \times_{32} (M_{i+14} +_{16} K_{i+14}))$
      viii. $Y := Y +_{32} ((M_{i+7} +_{16} K_{i+7}) \times_{32} (M_{i+15} +_{16} K_{i+15}))$
        ix. $i := i + 16$
4. Return the final value of $Y$
```

First-layer hash: L1HASH16.

We describe below the first level of hashing in UHASH16. It is based on the NH16 hash function. This function requires a key which is just as long as the message being hashed. Therefore, in order to limit the amount of key material that is needed, we use a key of fixed length (8192 bits) and process the message in blocks of this length (or shorter). Each block is hashed with NH16 and length information is included. The results are concatenated to form the output of L1HASH16. Note that the blocks of 8192 bits are compressed to 32-bit strings, which corresponds to a compression ratio of 256.

Description:

```
Input:   a string $K$ of 8192 bits
         a string $M$ of $Mlen$ bits, with $Mlen < 2^{67}$
Output:  a string $Y$ of $32 \times \lceil Mlen/8192 \rceil$ bits

1. Divide $M$ into substrings of 8192 bits (the final substring may be shorter)
   a) Set $t := \lceil Mlen/8192 \rceil$
   b) Let $M_1, M_2, \ldots, M_t$ be strings such that $M = M_1 \,\|\, M_2 \,\|\, \cdots \,\|\, M_t$,
      where the length of $M_i$ is 8192 bits for $1 \leq i < t$ and the length of
      $M_t$ is $tlen$ bits with $tlen \leq 8192$
2. Let the initial value of $Y$ be the empty string
3. Process the substrings $M_i$ $(1 \leq i < t)$
   a) $Len :=$ uint2str(8192,32)
   b) For $i := 1$ to $(t-1)$ do $Y := Y \,\|\, (\mathsf{NH16}(K, M_i) +_{32} Len)$
4. Process the final substring $M_t$
   a) $Len :=$ uint2str($tlen$,32)
   b) $M_t :=$ padzero($M_t$,256)
   c) $Y := Y \,\|\, (\mathsf{NH16}(K, M_t) +_{32} Len)$
5. Return the final value of $Y$
```

Polynomial hash with 32-bit wordsize: POLY32.

We describe below a polynomial hashing scheme which treats an input message as a sequence of coefficients of a polynomial, and the key as the point at which this polynomial is evaluated. The largest prime smaller than 2^{32} is used as modulus, and the message is divided in 32-bit words. Words that are larger than $2^{32} - 6$ are split into two words. This guarantees that all words are smaller than the prime modulus.

Description:

```
Input:   an integer $k$,with $0 \leq k \leq \mathsf{prime}(32) - 1$
         a string $M$ of $Mlen$ bits, with $Mlen$ divisible by 32
Output:  an integer $y$, with $0 \leq y \leq \mathsf{prime}(32) - 1$

1. Set $p := \mathsf{prime}(32) = 2^{32} - 5$
2. Divide $M$ into substrings of 32 bits
   a) Set $t := Mlen/32$
   b) Let $M_1, \ldots, M_t$ be strings of 32 bits such that $M = M_1 \,\|\, \cdots \,\|\, M_t$
3. Set the initial value of $y := 1$
```

```
4. Process the substrings $M_i$. That is, for $i := 1$ to $t$ do
   a) $m := \mathsf{str2uint}(M_i)$
   b) If $(m < 2^{32} - 6)$ then $y := (k \times y + m) \bmod p$
      else do the following two steps
        i. $y := (k \times y + (p - 1)) \bmod p$
       ii. $y := (k \times y + (m - 5)) \bmod p$
5. Return the final value of $y$
```

Polynomial hash with 64-bit wordsize: POLY64.
See the description of UMAC32 for a specification of this function.

Polynomial hash with 128-bit wordsize: POLY128.
See the description of UMAC32 for a specification of this function.

Second-layer hash: L2HASH16.
We describe below the second level of hashing in UHASH16. It is based on the POLY32, POLY64 and POLY128 hash functions. The security guarantee of a polynomial hashing scheme degrades linearly in the length of the message being hashed: if two messages of n words are hashed the collision probability is no more than n/p (where p denotes the prime modulus that is used). The scheme described below hashes n_1 words under the modulus $\mathsf{prime}(32)$ until $n_1/\mathsf{prime}(32)$ reaches a certain bound. Then the result obtained so far is prepended to the remaining message and hashing continues under the modulus $\mathsf{prime}(64)$ which is substantially larger than $\mathsf{prime}(32)$. The hashing continues for n_2 more words until $n_2/\mathsf{prime}(64)$ also reaches a certain bound, at which time a new larger prime modulus $\mathsf{prime}(128)$ is used for the remaining message. Note that the dynamic use of the different polynomial hash functions gives good performance for short messages while still accomodating longer ones. The keys used for the polynomial hashing are restricted to particular subsets to allow for optimisations (some potential arithmetic carries can be disregarded during the computation). L2HASH16 produces an output of a fixed length of 128 bits.

Description:

```
Input:    a string $K$ of 224 bits
          a string $M$ of $Mlen$ bits, with $Mlen < 2^{67}$
Output:   a string $Y$ of 128 bits

1. Extract key values and restrict them to special key-sets
   a) Let $K_{32}$ be a string of 32 bits, let $K_{64}$ be a string of 64 bits, and let
      $K_{128}$ be a string of 128 bits, such that $K = K_{32} \| K_{64} \| K_{128}$
   b) Define three strings:
      $Mask_{32} := \mathsf{uint2str}(\mathtt{1FFFFFFF}_{\mathtt{x}}, 32)$
      $Mask_{64} := \mathsf{uint2str}(\mathtt{01FFFFFF01FFFFFF}_{\mathtt{x}}, 64)$
      $Mask_{128} := \mathsf{uint2str}(\mathtt{01FFFFFF01FFFFFF01FFFFFF01FFFFFF}_{\mathtt{x}}, 128)$
   c) Compute three integers:
      $k_{32} := \mathsf{str2uint}(K_{32} \wedge Mask_{32})$
      $k_{64} := \mathsf{str2uint}(K_{64} \wedge Mask_{64})$
      $k_{128} := \mathsf{str2uint}(K_{128} \wedge Mask_{128})$
```

```
2. If $Mlen \le 2^{14}$ then $y := \mathsf{POLY32}(k_{32}, M)$
   else if $Mlen \le 2^{36}$ then do the following four steps
     a) Let $M_1$ be a string of $2^{14}$ bits, and let $M_2$ be a string of no more
        than $2^{36} - 2^{14}$ bits, such that $M = M_1 \parallel M_2$
     b) $M_2 := \mathsf{padonezero}(M_2, 64)$
     c) $y := \mathsf{POLY32}(k_{32}, M_1)$
     d) $y := \mathsf{POLY64}(k_{64}, \mathsf{uint2str}(y, 64) \parallel M_2)$
   else do the following five steps
     a) Let $M_1$ be a string of $2^{14}$ bits, let $M_2$ be a string of $2^{36} - 2^{14}$ bits,
        and let $M_3$ be a string of less than $2^{67} - 2^{36}$ bits, such that $M =$
        $M_1 \parallel M_2 \parallel M_3$
     b) $M_3 := \mathsf{padonezero}(M_3, 128)$
     c) $y := \mathsf{POLY32}(k_{32}, M_1)$
     d) $y := \mathsf{POLY64}(k_{64}, \mathsf{uint2str}(y, 64) \parallel M_2)$
     e) $y := \mathsf{POLY128}(k_{128}, \mathsf{uint2str}(y, 128) \parallel M_3)$
3. Return $Y := \mathsf{uint2str}(y, 128)$
```

Third-layer hash: L3HASH16.

We describe below the third level of hashing in UHASH16. It hashes a 128-bit input to a fixed length of 16 bits using a simple inner-product hash with affine translation. A 19-bit prime modulus is used to improve security.

```
Description:
   Input:    a string $K$ of 256 bits
             a string $T$ of 16 bits
             a string $M$ of 128 bits
   Output:   a string $Y$ of 16 bits

   1. Set $p := \mathsf{prime}(19) = 2^{19} - 1$
   2. Divide $M$ and $K$ into substrings of 16 bits and 32 bits respectively
      a) Let $M_1, \ldots, M_8$ be strings of 16 bits such that $M = M_1 \parallel \cdots \parallel M_8$
      b) Let $K_1, \ldots, K_8$ be strings of 32 bits such that $K = K_1 \parallel \cdots \parallel K_8$
   3. Convert the substrings into numbers. That is, for $i := 1$ to 8 do
      a) $m_i := \mathsf{str2uint}(M_i)$
      b) $k_i := \mathsf{str2uint}(K_i) \bmod p$
   4. Compute the output as follows
      a) $y := ((m_1 \times k_1 + m_2 \times k_2 + \cdots + m_8 \times k_8) \bmod p) \bmod 2^{16}$
      b) Return $Y := \mathsf{uint2str}(y, 16) \oplus T$
```

1.2.4.2 Three-layer hashing scheme: UHASH16

We describe below the universal hash function UHASH16 that is used by UMAC16 to compress the message input to a result of a fixed length. The components described in the previous section are combined in a straightforward manner. A message is first hashed with L1HASH16, its output is then hashed with L2HASH16, whose output is then hashed with L3HASH16. If the message is no longer than 8192 bits to start with, L2HASH16 is skipped as an optimisation. Because the output of L3HASH16 is only 16 bits long, multiple iterations of the three-layer hash scheme are used with different keys each time. All of these subkeys are derived from the user-supplied key with the key derivation function KDF. To reduce memory requirements L1HASH16 reuses most of its key material between iterations.

Description:

```
Input:   a string $K$ of 128 bits
         a string $M$ of $Mlen$ bits, with $Mlen < 2^{67}$
         an integer $Outlen = 32$, 64, 96, or 128
Output:  a string $Y$ of $Outlen$ bits

1. Set $t := Outlen/16$
2. Derive the needed key material from $K$
   a) $K1 := \mathsf{KDF}(K, 0, 8192 + (t-1) \times 128)$
   b) $K2 := \mathsf{KDF}(K, 1, t \times 224)$
   c) $K3 := \mathsf{KDF}(K, 2, t \times 256)$
   d) $T3 := \mathsf{KDF}(K, 3, t \times 16)$
3. Let the initial value of $Y$ be the empty string
4. Iterate the three-layer hashing scheme. That is, for $i := 1$ to $t$ do
   a) Define the subkeys to be used in this iteration
        i. $K1_i := K1[(i-1) \times 128 + 1 \ldots (i-1) \times 128 + 8192]$
       ii. $K2_i := K2[(i-1) \times 224 + 1 \ldots i \times 224]$
      iii. $K3_i := K3[(i-1) \times 256 + 1 \ldots i \times 256]$
       iv. $T3_i := T3[(i-1) \times 16 + 1 \ldots i \times 16]$
   b) $A := \mathsf{L1HASH16}(K1_i, M)$
   c) If $Mlen \leq 8192$ then $B := \mathsf{zeroes}(96) \parallel A$
      else $B := \mathsf{L2HASH16}(K2_i, A)$
   d) $C := \mathsf{L3HASH16}(K3_i, T3_i, B)$
   e) $Y := Y \parallel C$
5. Return the final value of $Y$
```

1.2.4.3 Message authentication code: UMAC16

The UMAC16 algorithm computes a message authentication value when given a key, message and nonce. It uses UHASH16 to compress the message and AES to encrypt the nonce. The exclusive-or (XOR) of the two resulting strings forms the output message authentication value. In the description below we consider output lengths of 32, 64, 96 or 128 bits, determined by the parameter *Outlen*.

Description:

```
Input:   a string $K$ of 128 bits
         a string $M$ of $Mlen$ bits, with $Mlen < 2^{67}$
         a string $Nonce$ of $Nlen$ bits, with $8 \leq Nlen \leq 128$
                       and $Nlen$ divisible by 8
         an integer $Outlen = 32$, 64, 96, or 128
Output:  a string $Y$ of $Outlen$ bits

1. $Hash := \mathsf{UHASH16}(K, M, Outlen)$
```

```
2. $K_{Enc} := \mathsf{KDF}(K, 128, 128)$
3. $Nonce := Nonce \,\|\, \mathsf{zeroes}(128 - Nlen)$
4. $Pad := \mathsf{AES}(K_{Enc}, Nonce)$
5. $Pad := Pad[1 \ldots Outlen]$
6. Return $Y := Hash \oplus Pad$
```

Reuse of the encrypted nonce for short output lengths. When the parameter *Outlen* in the description above is equal to 32 or 64 bits, the AES output $Pad := \mathsf{AES}(K_{Enc}, Nonce)$ can be reused for several authentications (using a different substring of the AES output each time). This reduces the average time spent by AES for each authentication. We refer to [379] for more information on this optimisation.

1.2.5 Parameters of UMAC and other versions of the algorithm

We have given specifications for two particular versions of UMAC (named UMAC32 and UMAC16). More generally, other versions of UMAC can be specified according to the parameters described below. For each parameter we discuss its role, the values chosen for it in UMAC32 and UMAC16, and other values that may be used. For more information on the use of UMAC with different parameters than in UMAC32 and UMAC16 see [379].

- *Word length*: defines the bitsize of a "word". Allowed values are 16 bits (as in UMAC16) and 32 bits (as in UMAC32). Generally, an implementation of UMAC will be efficient if the processor on which it runs has good support for operations on data types of this size. This is the case if the word length chosen is equal to the native word size of the target machine, but in some cases a word length smaller than the native word size turns out to be preferrable (e.g., when this allows to exploit small-scale vector parallelism supported by the processor).
- *Key length*: defines the bitsize of the user-supplied key. For both UMAC32 and UMAC16, the key length is 128 bits. Alternatively, a key length of 256 bits may be chosen. In this case the KDF function is based on AES with 256-bit key. Furthermore the derived key K_{Enc}, used for AES-encryption of the nonce, should then have a length of 256 bits (instead of 128).
- *Output length*: defines the bitsize of the UMAC output. The default output length is 64 bits, although we have allowed output lengths of 32, 64, 96 or 128 bits in the specifications of UMAC32 and UMAC16 in order to provide different security levels. More generally, any output length up to 256 bits that is a multiple of 8 bits may be defined for UMAC.
- *Block length*: defines the bitsize of the blocks in which the message is divided in the first layer L1HASH. This length is equal to $2^{13} = 8192$ bits for both UMAC32 and UMAC16. More generally, lengths of $2^8, 2^9, \ldots, 2^{28}$ bits are allowed. Larger block lengths mean greater compression and higher performance, but more storage needed for internal keys. When the data being hashed and the key used no longer simultaneously fit the processor cache, the performance

will decrease. Note that the second layer L2HASH is skipped when the message is not longer than one block.
- *Little-endian or big-endian*: specifies which endian-orientation will be followed in the reading of data to be hashed. Both UMAC32 and UMAC16 favour processors with little-endian architecture, because these processors automatically carry out the bytereverse operation, as specified in the functions NH32 and NH16. For implementations targeted to big-endian processors one may use a version of UMAC without this byte-reversal in the function NH.
- *Signed or unsigned*: specifies whether the strings manipulated in the hash function are to be considered initially as signed or unsigned integers. For UMAC16 the signed representation is used, and for UMAC32 the unsigned representation (see the interpretation of the operations $+_w$ and $\times_w$).

1.3 Test vectors for UMAC32

The following test vectors are generated for UMAC32 with an output length of 64 bits. The test vectors are denoted as quadruplets (`message`, `nonce`, `key`, `MAC`). This correponds to `MAC` = UMAC32(`key`, `message`, `nonce`, 64).

Two sets of test vectors are given, for two different values of the `key`. One particular `nonce` value is used for all test vectors. Note that in practice one must use a random secret `key` that is generated by a secure key generation algorithm. The `nonce` must be a number that doesn't repeat when authenticating messages under the same key; these numbers can be provided by a counter or a random number generator and do not have to be secret.

The `key` and `MAC` are denoted as hexadecimal numbers of 16 bytes (128 bits) and 8 bytes (64 bits) respectively. The `message` and `nonce` are denoted as ASCII strings (e.g., the character `"1"` corresponds to the hexadecimal byte $\texttt{31}_\texttt{x}$ and the character `"a"` corresponds to the hexadecimal byte $\texttt{61}_\texttt{x}$).

1.3.1 Set 1

```
message = "" (empty string)
nonce = "123456789"
key = 00112233445566778899AABBCCDDEEFF
MAC = B4B4A647B14B68A2

message = "a"
nonce = "123456789"
key = 00112233445566778899AABBCCDDEEFF
MAC = 4E4AC79DB8D94582

message = "abc"
nonce = "123456789"
key = 00112233445566778899AABBCCDDEEFF
MAC = 66BAA8C9F523155A
```

```
message = "message digest"
nonce = "123456789"
key = 00112233445566778899AABBCCDDEEFF
MAC = 8418F6C20938F3F3

message = "abcdefghijklmnopqrstuvwxyz"
nonce = "123456789"
key = 00112233445566778899AABBCCDDEEFF
MAC = BBE67587E1CFF5AD

message = "abcdbcdecdefdefgefghfghighij
           hijkijkljklmklmnlmnomnopnopq"
nonce = "123456789"
key = 00112233445566778899AABBCCDDEEFF
MAC = 7203A61AC5892CF8

message = "A...Za...z0...9"
nonce = "123456789"
key = 00112233445566778899AABBCCDDEEFF
MAC = 8B7079B2341BFDEA

message = 8 times "1234567890"
nonce = "123456789"
key = 00112233445566778899AABBCCDDEEFF
MAC = 3315478D6D287209

message = "Now is the time for all "
nonce = "123456789"
key = 00112233445566778899AABBCCDDEEFF
MAC = 9CD818CCE76C6070

message = "Now is the time for it"
nonce = "123456789"
key = 00112233445566778899AABBCCDDEEFF
MAC = 346DB75B3328BAC1

message = 1 million times "a"
nonce = "123456789"
key = 00112233445566778899AABBCCDDEEFF
MAC = 83F416954359BF09
```

1.3.2 Set 2

```
message = "" (empty string)
nonce = "123456789"
key = 0123456789ABCDEF0123456789ABCDEF
MAC = FF2ECE16C08698D3
```

```
message = "a"
nonce = "123456789"
key = 0123456789ABCDEF0123456789ABCDEF
MAC = 1A6F8D49C06E45E4

message = "abc"
nonce = "123456789"
key = 0123456789ABCDEF0123456789ABCDEF
MAC = 256ABE68D8C28A02

message = "message digest"
nonce = "123456789"
key = 0123456789ABCDEF0123456789ABCDEF
MAC = 4232400D2FF7A5DD

message = "abcdefghijklmnopqrstuvwxyz"
nonce = "123456789"
key = 0123456789ABCDEF0123456789ABCDEF
MAC = D599E5F2A4D5434A

message = "abcdbcdecdefdefgefghfghighij
           hijkijkljklmklmnlmnomnopnopq"
nonce = "123456789"
key = 0123456789ABCDEF0123456789ABCDEF
MAC = D735E16B8AD0587F

message = "A...Za...z0...9"
nonce = "123456789"
key = 0123456789ABCDEF0123456789ABCDEF
MAC = FD7A926ABAD80321

message = 8 times "1234567890"
nonce = "123456789"
key = 0123456789ABCDEF0123456789ABCDEF
MAC = 5DAD9004B3DB0280

message = "Now is the time for all "
nonce = "123456789"
key = 0123456789ABCDEF0123456789ABCDEF
MAC = 09DC488E3E93B941

message = "Now is the time for it"
nonce = "123456789"
key = 0123456789ABCDEF0123456789ABCDEF
MAC = AA09663BDE4C24B1

message = 1 million times "a"
nonce = "123456789"
key = 0123456789ABCDEF0123456789ABCDEF
MAC = 930A8AD2BAF5C7A1
```

1.4 Test vectors for UMAC16

The following test vectors are generated for UMAC16 with an output length of 64 bits. The test vectors are denoted as quadruplets (message, nonce, key, MAC). This correponds to MAC = UMAC16(key, message, nonce, 64).

Two sets of test vectors are given, for two different values of the key. One particular nonce value is used for all test vectors. Note that in practice one must use a random secret key that is generated by a secure key generation algorithm. The nonce must be a number that doesn't repeat when authenticating messages under the same key; these numbers can be provided by a counter or a random number generator and do not have to be secret.

The key and MAC are denoted as hexadecimal numbers of 16 bytes (128 bits) and 8 bytes (64 bits) respectively. The message and nonce are denoted as ASCII strings (e.g., the character "1" corresponds to the hexadecimal byte $\mathtt{31}_\mathtt{x}$ and the character "a" corresponds to the hexadecimal byte $\mathtt{61}_\mathtt{x}$).

1.4.1 Set 1

```
message = "" (empty string)
nonce = "123456789"
key = 00112233445566778899AABBCCDDEEFF
MAC = ADC025707C249BEB

message = "a"
nonce = "123456789"
key = 00112233445566778899AABBCCDDEEFF
MAC = A6E03B4542C7CD2E

message = "abc"
nonce = "123456789"
key = 00112233445566778899AABBCCDDEEFF
MAC = 85D682506D04F881

message = "message digest"
nonce = "123456789"
key = 00112233445566778899AABBCCDDEEFF
MAC = 2196A961BC198425

message = "abcdefghijklmnopqrstuvwxyz"
nonce = "123456789"
key = 00112233445566778899AABBCCDDEEFF
MAC = 2975651EB8FE20CC

message = "abcdbcdecdefdefgefghfghighij
           hijkijkljklmklmnlmnomnopnopq"
nonce = "123456789"
key = 00112233445566778899AABBCCDDEEFF
MAC = 453066F118536AC6
```

```
message = "A...Za...z0...9"
nonce = "123456789"
key = 00112233445566778899AABBCCDDEEFF
MAC = 3563645DF1F3A0B6

message = 8 times "1234567890"
nonce = "123456789"
key = 00112233445566778899AABBCCDDEEFF
MAC = CF07D9A8F4B32DC7

message = "Now is the time for all "
nonce = "123456789"
key = 00112233445566778899AABBCCDDEEFF
MAC = 6070AB5BB7559135

message = "Now is the time for it"
nonce = "123456789"
key = 00112233445566778899AABBCCDDEEFF
MAC = FC8C75AD9776F587

message = 1 million times "a"
nonce = "123456789"
key = 00112233445566778899AABBCCDDEEFF
MAC = 336121DAA5C135C7
```

1.4.2 Set 2

```
message = "" (empty string)
nonce = "123456789"
key = 0123456789ABCDEF0123456789ABCDEF
MAC = 2511CCA2391013F4

message = "a"
nonce = "123456789"
key = 0123456789ABCDEF0123456789ABCDEF
MAC = 4ABD61A21EC4C65D

message = "abc"
nonce = "123456789"
key = 0123456789ABCDEF0123456789ABCDEF
MAC = 0B77207ACFBEBA28

message = "message digest"
nonce = "123456789"
key = 0123456789ABCDEF0123456789ABCDEF
MAC = 46362C34678676DF
```

```
message = "abcdefghijklmnopqrstuvwxyz"
nonce = "123456789"
key = 0123456789ABCDEF0123456789ABCDEF
MAC = 34EEFA9A6DF25298

message = "abcdbcdecdefdefgefghfghighij
           hijkijkljklmklmnlmnomnopnopq"
nonce = "123456789"
key = 0123456789ABCDEF0123456789ABCDEF
MAC = F187F5BC514A5E05

message = "A...Za...z0...9"
nonce = "123456789"
key = 0123456789ABCDEF0123456789ABCDEF
MAC = C869A08C7F73EF24

message = 8 times "1234567890"
nonce = "123456789"
key = 0123456789ABCDEF0123456789ABCDEF
MAC = A844FC0885A7238C

message = "Now is the time for all "
nonce = "123456789"
key = 0123456789ABCDEF0123456789ABCDEF
MAC = B56F03EB69DADF21

message = "Now is the time for it"
nonce = "123456789"
key = 0123456789ABCDEF0123456789ABCDEF
MAC = 90287B810E096BBB

message = 1 million times "a"
nonce = "123456789"
key = 0123456789ABCDEF0123456789ABCDEF
MAC = 397BD9D499A54416
```

2. TTMAC

2.1 Introduction

2.1.1 Overview

TTMAC (also known as Two-Track-MAC) is a message authentication code submitted to NESSIE by Bert den Boer and Bart Van Rompay [609]. TTMAC is used to compress messages of arbitrary length into a 160-bit output, under control of a 160-bit secret key; its design is based on the (unkeyed) hash function RIPEMD-160 with some (small) modifications. RIPEMD-160 is one of the hash functions included in the standard ISO/IEC 10118-3 [306]. It was proposed in 1996 by Hans Dobbertin, Bart Preneel and Antoon Bosselaers [531] as a strengthened version of RIPEMD, a hash function developed in the framework of the EU project RIPE [116]. Both RIPEMD-160 and TTMAC are constructed from an underlying compression function that uses two parallel trails of computation.

2.1.2 Outline of the primitive

TTMAC maps a message M consisting of an arbitrary number of bits into a 160-bit block $\mathbf{TTMAC}(K, M)$ under control of a 160-bit key K. The algorithm is based on the iteration of a compression function that tranforms an input block of 320 bits and a message block of 512 bits into an output block of 320 bits.

First, the message is expanded to an appropriate length and divided into message blocks $M_0 \ldots M_{t-1}$ where each block M_i $(0 \leq i \leq t-1)$ consists of 512 bits. Next, an initial value is defined as follows: two duplicate copies of the key are concatenated and the resulting string of 320 bits forms the initial value: $H_0 = (K \| K)$.

The compression function $\mathbf{F}$ is used to transform a 320-bit input block H_i and a 512-bit message block M_i into a 320-bit output block $H_{i+1} = \mathbf{F}(H_i, M_i)$. H_{i+1} then forms the input to the next iteration of the compression function together with the next message block M_{i+1}.

The last message block M_{t-1} is processed by a slightly modified variant of the compression function $\mathbf{F}'$, and an output transformation $\mathbf{G}$ finalises the computation, where $\mathbf{G}$ takes a 320-bit input and produces a 160-bit output. This results in the following outline of the TTMAC algorithm:

1. Expand the message M and divide it into message blocks $M_0 \ldots M_{t-1}$.

2. Use the key K to define $H_0 = (K \| K)$.
3. For $0 \leq i \leq t-2$ compute $H_{i+1} = \mathbf{F}(H_i, M_i)$.
4. Compute $H_t = \mathbf{F}'(H_{t-1}, M_{t-1})$.
5. Compute the message authentication value $\mathbf{TTMAC}(K, M) = \mathbf{G}(H_t)$.

2.1.3 Security and performance

The security of TTMAC can be proven based on the assumption that the underlying compression function is pseudo-random. This function is very similar to the compression function used by RIPEMD-160. There are no known short-cut attacks on TTMAC and the security level depends on the length of the key (160 bits) and on the length of the message authentication value (the default is 160 bits but this can be truncated to 32, 64, 96, or 128 bits). Note that the large size of the internal state in TTMAC (320 bits) gives the algorithm a high level of security against attacks based on internal collisions.

The performance of TTMAC is close to the performance of the unkeyed hash function RIPEMD-160 and this is already the case for the shortest possible message (512 bits after expansion). Furthermore, since the key is used only to determine the initial value for the computation, TTMAC has optimal key-agility. Therefore TTMAC is very useful for applications that require the authentication of short messages, where the key is changed for every authentication.

The design of TTMAC is oriented towards a fast software implementation on 32-bit architectures: it is based on a simple set of operations on 32-bit words. Moreover, since no substitution tables are used the implementation can be quite compact as well. The large input block size is unfavorable for compact hardware implementations, but a high speed in hardware is certainly possible. Parallelism can only be used to run both halves of the compression function simultaneously, as every register in each half gets updated sequentially.

2.2 Description

2.2.1 High level description

Let $M = (m_0, m_1, \ldots, m_{n-1})$ be a message consisting of n bits, and let $K = (k_0, k_1, \ldots, k_{159})$ denote the 160-bit secret key. The message authentication value $\mathbf{TTMAC}(K, M)$ is computed as described below. Note that the wordlength in TTMAC is 32 bits. This means that all operations are performed on 32-bit words.

key representation. The key K is represented as a sequence of five words $(K_0, K_1, K_2, K_3, K_4)$.

message expansion. The message M is expanded to a message W consisting of $16t$ words $W_0, W_1, \ldots, W_{16t-1}$, where $t = ((n + 64) \text{ div } 512) + 1$. That is, the message is expanded such that it becomes a multiple of 512 bits long, and then it is represented as a sequence of words. Note that the expansion is done even if M already is a multiple of 512 bits long.

compression. The resulting message W is compressed as follows using a compression function $\mathbf{F}$ and a modified compression function $\mathbf{F}'$.

- Define ten words $H_{0,0}, H_{0,1}, \ldots, H_{0,9}$ as

$$\begin{array}{c} H_{0,0} := K_0 \ , \ H_{0,1} := K_1 \ , \ H_{0,2} := K_2 \ , \ H_{0,3} := K_3 \ , \ H_{0,4} := K_4 \ , \\ H_{0,5} := K_0 \ , \ H_{0,6} := K_1 \ , \ H_{0,7} := K_2 \ , \ H_{0,8} := K_3 \ , \ H_{0,9} := K_4 \ . \end{array}$$

- For $i = 0, 1, \ldots, t-2$ the words $H_{i+1,0}, H_{i+1,1}, \ldots, H_{i+1,9}$ are computed as follows from the words $H_{i,0}, H_{i,1}, \ldots, H_{i,9}$ and sixteen message words $W_{16i}, W_{16i+1}, \ldots, W_{16i+15}$:

$$(H_{i+1,0}, \ldots, H_{i+1,9}) := \mathbf{F}((H_{i,0}, \ldots, H_{i,9}); (W_{16i}, \ldots, W_{16i+15})) \ .$$

  That is, each 16-word message block is used to transform $(H_{i,0}, H_{i,1}, \ldots, H_{i,9})$ into $(H_{i+1,0}, H_{i+1,1}, \ldots, H_{i+1,9})$ for the current value of i.

- For $i = t-1$ the words $H_{t,0}, H_{t,1}, \ldots, H_{t,9}$ are computed in the same manner but with a different compression function:

$$(H_{t,0}, \ldots, H_{t,9}) := \mathbf{F}'((H_{t-1,0}, \ldots, H_{t-1,9}); (W_{16t-16}, \ldots, W_{16t-1})) \ .$$

output transformation. The ten-word output from the previous step is transformed into a five-word result using an output transformation $\mathbf{G}$:

$$(H_{out,0}, H_{out,1}, H_{out,2}, H_{out,3}, H_{out,4}) := \mathbf{G}(H_{t,0}, H_{t,1}, \ldots, H_{t,9}) \ .$$

optional truncation. One out of four optional functions $\mathbf{T}_1, \mathbf{T}_2, \mathbf{T}_3$, or $\mathbf{T}_4$ may be used to truncate the final result of the computation:

$$(H_{trunc,0}, \ldots, H_{trunc,i-1}) := \mathbf{T}_i(H_{out,0}, H_{out,1}, H_{out,2}, H_{out,3}, H_{out,4}) \ ,$$

where $i = 1$, 2, 3, or 4 and the truncated output consists of i words.

message authentication value. The default message authentication value $\mathbf{TTMAC}(K, M)$ is the 160-bit string that consists of the concatenation of the five 32-bit strings corresponding to the five words $H_{out,0}, H_{out,1}, H_{out,2}, H_{out,3}$ and $H_{out,4}$. That is,

$$\mathbf{TTMAC}(K, M) := H_{out,0} \,\|\, H_{out,1} \,\|\, H_{out,2} \,\|\, H_{out,3} \,\|\, H_{out,4} \ .$$

Here the first byte corresponds to the least significant byte of $H_{out,0}$ and the last byte corresponds to the most significant byte of $H_{out,4}$ according to the little-endian convention.

If desired, message authentication values of a shorter length (32, 64, 96 or 128 bits) can be generated. In this case, the truncation function $\mathbf{T}_i$ ($i = 1$, 2, 3, or 4) is applied, and the truncated message authentication value $\mathbf{TTMAC}_{32i}(K, M)$ is the $32i$-bit string that consists of the concatenation of the i 32-bit strings corresponding to the words $H_{trunc,0}, \ldots, H_{trunc,i-1}$. That is,

$$\mathbf{TTMAC}_{32i}(K, M) := H_{trunc,0} \,\|\, \ldots \,\|\, H_{trunc,i-1} \ ,$$

(according to the little-endian convention).

2.2.2 Representation of the key

The 160-bit key $K = (k_0, k_1, \ldots, k_{159})$ is transformed into a sequence of five words $(K_0, K_1, K_2, K_3, K_4)$ according to little-endian convention:

$$K_i := \sum_{j=0}^{3} \sum_{l=0}^{7} k_{(32i+8j+l)} 2^{8j+(7-l)} , \quad 0 \leq i \leq 4 .$$

2.2.3 Expanding the message

Let $t = ((n + 64) \text{ div } 512) + 1$. The n-bit message $M = (m_0, m_1, \ldots, m_{n-1})$ is expanded to the $16t$-word message $W = (W_0, W_1, \ldots, W_{16t-1})$ in the following three steps.

1. First of all, the following r bits are appended to the message M, where $1 \leq r \leq 512$ and $n + r \equiv 448 \mod 512$:

   $$\begin{aligned} m_n &:= 1 , \\ m_{n+1} := m_{n+2} := \cdots := m_{n+r-1} &:= 0 . \end{aligned}$$

   In other words, append a single 1-bit and a number of 0-bits until the expanded message is 64 bits shorter than a multiple of 512 bits. Note that the padding is always done, even if the message is already 64 bits shorter than a multiple of 512 bits. In that case, the message is expanded by 512 bits.
2. This $(n + r)$-bit extended message is transformed into $\frac{n+r}{32} = 16t - 2$ words $W_0, W_1, \ldots, W_{16t-3}$ according to little-endian convention:

   $$W_i := \sum_{j=0}^{3} \sum_{l=0}^{7} m_{(32i+8j+l)} 2^{8j+(7-l)} , \quad i = 0, 1, \ldots, 16t - 3 .$$
3. Finally, the expansion is completed by appending the length n of the original message in the following way:

   $$\begin{aligned} W_{16t-2} &:= n \mod 2^{32} , \\ W_{16t-1} &:= (n \mod 2^{64}) \text{ div } 2^{32} . \end{aligned}$$

2.2.4 The compression function F

Given a 10-word input block $(H_0, H_1, \ldots, H_9)$ and a 16-word message block $(W_0, W_1, \ldots, W_{15})$, the compression function $\mathbf{F}$ computes a 10-word output block $(R_0, R_1, \ldots, R_9)$ in the manner described below. That is,

$$(R_0, R_1, \ldots, R_9) := \mathbf{F}((H_0, H_1, \ldots, H_9); (W_0, W_1, \ldots, W_{15})) \ .$$

1. Use two functions $\mathbf{F_{LT}}$ and $\mathbf{F_{RT}}$ (the left and right trail) in the following way (the details of the functions are given below).

   $$\begin{aligned} (P_0, P_1, \ldots, P_4) &:= \mathbf{F_{LT}}((H_0, H_1, \ldots, H_4); (W_0, W_1, \ldots, W_{15})) \ , \\ (P_5, P_6, \ldots, P_9) &:= \mathbf{F_{RT}}((H_5, H_6, \ldots, H_9); (W_0, W_1, \ldots, W_{15})) \ . \end{aligned}$$

2. Compute a new set of ten words by subtracting the input words from the results of the previous step.

$$Q_j := P_j - H_j \mod 2^{32} \ , \ 0 \leq j \leq 9 \ .$$

3. To finish the compression function, the ten words $Q_i (0 \leq i \leq 9)$ are mixed in two linear transformations $\mathbf{F_{LM}}$ and $\mathbf{F_{RM}}$ (the details are given below).

$$\begin{aligned} (R_0, R_1, \ldots, R_4) &:= \mathbf{F_{LM}}(Q_0, Q_1, \ldots, Q_9) \ , \\ (R_5, R_6, \ldots, R_9) &:= \mathbf{F_{RM}}(Q_0, Q_1, \ldots, Q_9) \ . \end{aligned}$$

The functions $\mathbf{F_{LT}}$ and $\mathbf{F_{RT}}$

The functions $\mathbf{F_{LT}}$ and $\mathbf{F_{RT}}$, which are known as the left and right trail of the compression function, are identical to the functions used in the compression function of RIPEMD-160. They consist of 80 sequential steps which we describe below. We first define the functions and constants that are used.

Non-linear functions at bit level:

$$\begin{aligned} f_j(X,Y,Z) &= X \oplus Y \oplus Z \, , & 0 \leq j \leq 15 \, , \\ f_j(X,Y,Z) &= (X \wedge Y) \vee (\bar{X} \wedge Z) \, , & 16 \leq j \leq 31 \, , \\ f_j(X,Y,Z) &= (X \vee \bar{Y}) \oplus Z \, , & 32 \leq j \leq 47 \, , \\ f_j(X,Y,Z) &= (X \wedge Z) \vee (Y \wedge \bar{Z}) \, , & 48 \leq j \leq 63 \, , \\ f_j(X,Y,Z) &= X \oplus (Y \vee \bar{Z}) \, , & 64 \leq j \leq 79 \, . \end{aligned}$$

Additive constants (hexadecimal notation):

$$\begin{aligned} U_{Lj} &= \mathtt{00000000}_{\mathtt{x}} \, , & U_{Rj} &= \mathtt{50A28BE6}_{\mathtt{x}} \, , & 0 \leq j \leq 15 \, , \\ U_{Lj} &= \mathtt{5A827999}_{\mathtt{x}} \, , & U_{Rj} &= \mathtt{5C4DD124}_{\mathtt{x}} \, , & 16 \leq j \leq 31 \, , \\ U_{Lj} &= \mathtt{6ED9EBA1}_{\mathtt{x}} \, , & U_{Rj} &= \mathtt{6D703EF3}_{\mathtt{x}} \, , & 32 \leq j \leq 47 \, , \\ U_{Lj} &= \mathtt{8F1BBCDC}_{\mathtt{x}} \, , & U_{Rj} &= \mathtt{7A6D76E9}_{\mathtt{x}} \, , & 48 \leq j \leq 63 \, , \\ U_{Lj} &= \mathtt{A953FD4E}_{\mathtt{x}} \, , & U_{Rj} &= \mathtt{00000000}_{\mathtt{x}} \, , & 64 \leq j \leq 79 \, . \end{aligned}$$

Selection of message word:

$$\begin{aligned} sL[\ 0 \ldots 15] &= [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15] \\ sL[16 \ldots 31] &= [7,4,13,1,10,6,15,3,12,0,9,5,2,14,11,8] \\ sL[32 \ldots 47] &= [3,10,14,4,9,15,8,1,2,7,0,6,13,11,5,12] \\ sL[48 \ldots 63] &= [1,9,11,10,0,8,12,4,13,3,7,15,14,5,6,2] \\ sL[64 \ldots 79] &= [4,0,5,9,7,12,2,10,14,1,3,8,11,6,15,13] \\ sR[\ 0 \ldots 15] &= [5,14,7,0,9,2,11,4,13,6,15,8,1,10,3,12] \\ sR[16 \ldots 31] &= [6,11,3,7,0,13,5,10,14,15,8,12,4,9,1,2] \\ sR[32 \ldots 47] &= [15,5,1,3,7,14,6,9,11,8,12,2,10,0,4,13] \\ sR[48 \ldots 63] &= [8,6,4,1,3,11,15,0,5,12,2,13,9,7,10,14] \\ sR[64 \ldots 79] &= [12,15,10,4,1,5,8,7,6,2,13,14,0,3,9,11] \end{aligned}$$

Rotation constants:

$$\begin{aligned}
vL[\ 0\ldots15] &= [\,11,14,15,12,5,8,7,9,11,13,14,15,6,7,9,8\,]\\
vL[16\ldots31] &= [\,7,6,8,13,11,9,7,15,7,12,15,9,11,7,13,12\,]\\
vL[32\ldots47] &= [\,11,13,6,7,14,9,13,15,14,8,13,6,5,12,7,5\,]\\
vL[48\ldots63] &= [\,11,12,14,15,14,15,9,8,9,14,5,6,8,6,5,12\,]\\
vL[64\ldots79] &= [\,9,15,5,11,6,8,13,12,5,12,13,14,11,8,5,6\,]\\
vR[\ 0\ldots15] &= [\,8,9,9,11,13,15,15,5,7,7,8,11,14,14,12,6\,]\\
vR[16\ldots31] &= [\,9,13,15,7,12,8,9,11,7,7,12,7,6,15,13,11\,]\\
vR[32\ldots47] &= [\,9,7,15,11,8,6,6,14,12,13,5,14,13,13,7,5\,]\\
vR[48\ldots63] &= [\,15,5,8,11,14,14,6,14,6,9,12,9,12,5,15,8\,]\\
vR[64\ldots79] &= [\,8,5,12,9,12,5,14,6,8,13,6,5,15,13,11,11\,]
\end{aligned}$$

1. Suppose that the input block $(H_0, H_1, \ldots, H_9)$ and the 16-word message block $(W_0, W_1, \ldots, W_{15})$ are given. First copy the words of the input block into registers $(A_L, B_L, C_L, D_L, E_L)$ and $(A_R, B_R, C_R, D_R, E_R)$:

$$\begin{aligned}
&A_L := H_0\ ,\ B_L := H_1\ ,\ C_L := H_2\ ,\ D_L := H_3\ ,\ E_L := H_4\ ,\\
&A_R := H_5\ ,\ B_R := H_6\ ,\ C_R := H_7\ ,\ D_R := H_8\ ,\ E_R := H_9\ .
\end{aligned}$$

2. The function $\mathbf{F_{LT}}$ (left trail of the compression function) consists of the following steps for $0 \le j \le 79$ (additions are mod 2^{32}):

$$\begin{aligned}
temp &:= (A_L + f_j(B_L, C_L, D_L) + W_{sL[j]} + U_{Lj}) \lll_{vL[j]} + E_L\ ,\\
A_L &:= E_L\ ,\\
E_L &:= D_L\ ,\\
D_L &:= C_L \lll_{10}\ ,\\
C_L &:= B_L\ ,\\
B_L &:= temp\ .
\end{aligned}$$

3. The function $\mathbf{F_{RT}}$ (right trail of the compression function) consists of the following steps for $0 \le j \le 79$ (additions are mod 2^{32}):

$$\begin{aligned}
temp &:= (A_R + f_{79-j}(B_R, C_R, D_R) + W_{sR[j]} + U_{Rj}) \lll_{vR[j]} + E_R\ ,\\
A_R &:= E_R\ ,\\
E_R &:= D_R\ ,\\
D_R &:= C_R \lll_{10}\ ,\\
C_R &:= B_R\ ,\\
B_R &:= temp\ .
\end{aligned}$$

4. Finally, set $(P_0, P_1, P_2, P_3, P_4)$ equal to $(A_L, B_L, C_L, D_L, E_L)$ and similarly set $(P_5, P_6, P_7, P_8, P_9)$ equal to $(A_R, B_R, C_R, D_R, E_R)$. Then $(P_0, P_1, \ldots, P_4)$ and $(P_5, P_6, \ldots, P_9)$ form the outputs of the functions $\mathbf{F_{LT}}$ and $\mathbf{F_{RT}}$ respectively.

The functions $\mathbf{F_{LM}}$ and $\mathbf{F_{RM}}$

The linear transformations $\mathbf{F_{LM}}$ and $\mathbf{F_{RM}}$ are used to mix the outputs of the two trails of the compression function. For an input block $(Q_0, Q_1, \ldots, Q_9)$, the function $\mathbf{F_{LM}}$ computes five words $R_0, R_1, \ldots, R_4$ as follows (operations are mod 2^{32}):

$$\begin{aligned} R_0 &:= (Q_1 + Q_4) - Q_8 \ , \\ R_1 &:= Q_2 - Q_9 \ , \\ R_2 &:= Q_3 - Q_5 \ , \\ R_3 &:= Q_4 - Q_6 \ , \\ R_4 &:= Q_0 - Q_7 \ . \end{aligned}$$

For an input block $(Q_0, Q_1, \ldots, Q_9)$, the function $\mathbf{F_{RM}}$ computes five words $R_5, R_6, \ldots, R_9$ as follows (operations are mod 2^{32}):

$$\begin{aligned} R_5 &:= Q_3 - Q_9 \ , \\ R_6 &:= (Q_4 + Q_2) - Q_5 \ , \\ R_7 &:= Q_0 - Q_6 \ , \\ R_8 &:= Q_1 - Q_7 \ , \\ R_9 &:= Q_2 - Q_8 \ . \end{aligned}$$

2.2.5 The modified compression function $\mathbf{F}'$

Given a 10-word input block $(H_0, H_1, \ldots, H_9)$ and a 16-word message block $(W_0, W_1, \ldots, W_{15})$, the modified compression function $\mathbf{F}'$ computes a 10-word output block $(R_0, R_1, \ldots, R_9)$ in the manner described below. That is,

$$(R_0, R_1, \ldots, R_9) := \mathbf{F}'((H_0, H_1, \ldots, H_9); (W_0, W_1, \ldots, W_{15})) \ .$$

1. Use the two functions $\mathbf{F_{LT}}$ and $\mathbf{F_{RT}}$ in the following way. These functions are identical to the ones used in the compression function $\mathbf{F}$, however their role is reversed in the modified compression function $\mathbf{F}'$.

$$\begin{aligned} (P_0, P_1, \ldots, P_4) &:= \mathbf{F_{RT}}((H_0, H_1, \ldots, H_4); (W_0, W_1, \ldots, W_{15})) \ , \\ (P_5, P_6, \ldots, P_9) &:= \mathbf{F_{LT}}((H_5, H_6, \ldots, H_9); (W_0, W_1, \ldots, W_{15})) \ . \end{aligned}$$

2. Compute a new set of ten words by subtracting the input words from the results of the previous step.

$$R_j := P_j - H_j \mod 2^{32} \ , \ 0 \leq j \leq 9 \ .$$

Note that the modified compression function $\mathbf{F}'$ does not use the functions $\mathbf{F_{LM}}$ and $\mathbf{F_{RM}}$.

Use of the functions $\mathbf{F_{LT}}$ and $\mathbf{F_{RT}}$

The modified compression function $\mathbf{F}'$ uses the same functions $\mathbf{F_{LT}}$ and $\mathbf{F_{RT}}$ as the compression function $\mathbf{F}$, but their role is reversed. This results in the following changes to the description (compared to Sect. 2.2.4):

1. Suppose that the input block $(H_0, H_1, \ldots, H_9)$ and the 16-word message block $(W_0, W_1, \ldots, W_{15})$ are given. First copy the words of the input block into registers $(A_R, B_R, C_R, D_R, E_R)$ and $(A_L, B_L, C_L, D_L, E_L)$:

$$\begin{gathered} A_R := H_0 \ , \ B_R := H_1 \ , \ C_R := H_2 \ , \ D_R := H_3 \ , \ E_R := H_4 \ , \\ A_L := H_5 \ , \ B_L := H_6 \ , \ C_L := H_7 \ , \ D_L := H_8 \ , \ E_L := H_9 \ . \end{gathered}$$

4. Finally, set $(P_0, P_1, P_2, P_3, P_4)$ equal to $(A_R, B_R, C_R, D_R, E_R)$ and similarly set $(P_5, P_6, P_7, P_8, P_9)$ equal to $(A_L, B_L, C_L, D_L, E_L)$. Then $(P_0, P_1, \ldots, P_4)$ and $(P_5, P_6, \ldots, P_9)$ form the outputs of the functions $\mathbf{F_{RT}}$ and $\mathbf{F_{LT}}$ respectively.

2.2.6 The output transformation G

Given a 10-word input block $(H_0, H_1, \ldots, H_9)$, the output transformation $\mathbf{G}$ computes a 5-word output block $(R_0, R_1, R_2, R_3, R_4)$ in the manner described below. That is,

$$(R_0, R_1, R_2, R_3, R_4) := \mathbf{G}(H_0, H_1, \ldots, H_9) \ .$$

Compute five words R_0, R_1, R_2, R_3, R_4 as follows:

$$R_j := H_j - H_{j+5} \mod 2^{32} \ , \ 0 \leq j \leq 4 \ .$$

2.2.7 The truncation functions $\mathbf{T}_i$

Given a 5-word input block $(H_0, H_1, H_2, H_3, H_4)$, the truncation function $\mathbf{T}_i$ computes an i-word output block $(R_0, \ldots, R_{i-1})$ in the manner described below. That is,

$$(R_0, \ldots, R_{i-1}) := \mathbf{T}_i(H_0, H_1, H_2, H_3, H_4) \ .$$

– The function $\mathbf{T}_1$ computes one word R_0 as follows:

$$R_0 := H_0 + H_1 + H_2 + H_3 + H_4 \ .$$

– The function $\mathbf{T}_2$ computes two words R_0, R_1 as follows:

$$\begin{aligned} R_0 &:= H_0 + H_1 + H_3 \ , \\ R_1 &:= H_1 + H_2 + H_4 \ . \end{aligned}$$

– The function $\mathbf{T}_3$ computes three words R_0, R_1, R_2 as follows:

$$\begin{aligned} R_0 &:= H_0 + H_1 + H_3 \ , \\ R_1 &:= H_1 + H_2 + H_4 \ , \\ R_2 &:= H_2 + H_3 + H_0 \ . \end{aligned}$$

– The function $\mathbf{T}_4$ computes four words R_0, R_1, R_2, R_3 as follows:

$$\begin{aligned} R_0 &:= H_0 + H_1 + H_3 \ , \\ R_1 &:= H_1 + H_2 + H_4 \ , \\ R_2 &:= H_2 + H_3 + H_0 \ , \\ R_3 &:= H_3 + H_4 + H_1 \ . \end{aligned}$$

Note that all additions are mod 2^{32}.

2.3 Test vectors for TTMAC

The following test vectors are generated for TTMAC with the default output length of 160 bits. The test vectors are denoted as triples (`message`, `key`, `MAC`). This correponds to `MAC` = **TTMAC**(`key`, `message`). Two sets of test vectors are given, for two different values of the `key`. Note that in practice one should use a random secret `key` that is generated by a secure key generation algorithm.

The `key` and `MAC` are denoted as hexadecimal numbers of 20 bytes (160 bits). These numbers are the concatenation of five hexadecimal words consisting of four bytes (32 bits) each. The `message` is denoted as an ASCII string (e.g., the character `"a"` corresponds to the hexadecimal byte $\texttt{61}_\texttt{x}$). Note that the little-endian convention is used for conversion between bytes and four-byte words (e.g., the message string `"abcd"` corresponds to the hexadecimal word $\texttt{64636261}_\texttt{x}$).

2.3.1 Set 1

```
message = "" (empty string)
key = 00112233445566778899AABBCCDDEEFF01234567
MAC = 2DEC8ED4A0FD712ED9FBF2AB466EC2DF21215E4A

message = "a"
key = 00112233445566778899AABBCCDDEEFF01234567
MAC = 5893E3E6E306704DD77AD6E6ED432CDE321A7756

message = "abc"
key = 00112233445566778899AABBCCDDEEFF01234567
MAC = 70BFD1029797A5C16DA5B557A1F0B2779B78497E

message = "message digest"
key = 00112233445566778899AABBCCDDEEFF01234567
MAC = 8289F4F19FFE4F2AF737DE4BD71C829D93A972FA

message = "abcdefghijklmnopqrstuvwxyz"
key = 00112233445566778899AABBCCDDEEFF01234567
MAC = 2186CA09C5533198B7371F245273504CA92BAE60
```

```
message = "abcdbcdecdefdefgefghfghighij
           hijkijkljklmklmnlmnomnopnopq"
key = 00112233445566778899AABBCCDDEEFF01234567
MAC = 8A7BF77AEF62A2578497A27C0D6518A429E7C14D

message = "A...Za...z0...9"
key = 00112233445566778899AABBCCDDEEFF01234567
MAC = 54BAC392A886806D169556FCBB6789B54FB364FB

message = 8 times "1234567890"
key = 00112233445566778899AABBCCDDEEFF01234567
MAC = 0CED2C9F8F0D9D03981AB5C8184BAC43DD54C484

message = "Now is the time for all "
key = 00112233445566778899AABBCCDDEEFF01234567
MAC = DD77E545DD4552C9A3D441A11ADB9A7178780BD3

message = "Now is the time for it"
key = 00112233445566778899AABBCCDDEEFF01234567
MAC = A9D7E53597BBF8B187AE5B32E96C2BE5F63A69B4

message = 1 million times "a"
key = 00112233445566778899AABBCCDDEEFF01234567
MAC = 27B3AEDB5DF8B629F0142194DAA3846E1895F3D2
```

2.3.2 Set 2

```
message = "" (empty string)
key = 0123456789ABCDEF0123456789ABCDEF01234567
MAC = AAB071094FD5843B8509D4202CC8D50D98676EE9

message = "a"
key = 0123456789ABCDEF0123456789ABCDEF01234567
MAC = CEF8E42E78BC879C81579A48B8190B8E71E5832C

message = "abc"
key = 0123456789ABCDEF0123456789ABCDEF01234567
MAC = 6B4514D4F0AA0496DB4B6BD4352D8C778F6AC3DC

message = "message digest"
key = 0123456789ABCDEF0123456789ABCDEF01234567
MAC = EA1048408269CD0D10EB58F53878DF03E4D966FE

message = "abcdefghijklmnopqrstuvwxyz"
key = 0123456789ABCDEF0123456789ABCDEF01234567
MAC = DC4A53AB697D0F3579EF8C2A6073D421BEA8D1E5

message = "abcdbcdecdefdefgefghfghighij
           hijkijkljklmklmnlmnomnopnopq"
key = 0123456789ABCDEF0123456789ABCDEF01234567
MAC = 1C346FF07021E5655E74E3D4B914768105793C28
```

```
message = "A...Za...z0...9"
key = 0123456789ABCDEF0123456789ABCDEF01234567
MAC = C870219D72D51FDC8A5C63977D60BF393C50738D

message = 8 times "1234567890"
key = 0123456789ABCDEF0123456789ABCDEF01234567
MAC = BC737B0864B58E2FF0164D9732860C3F9AC4CF75

message = "Now is the time for all "
key = 0123456789ABCDEF0123456789ABCDEF01234567
MAC = 56244897814A3A1D893CBDAEE398A422C638DBAE

message = "Now is the time for it"
key = 0123456789ABCDEF0123456789ABCDEF01234567
MAC = 207040D6BA62517EDD2CD32E15E9EE0479EA448D

message = 1 million times "a"
key = 0123456789ABCDEF0123456789ABCDEF01234567
MAC = CAB10B780471AA8EDB4C3C02565624D6A4D4209A
```

3. EMAC

3.1 Introduction

3.1.1 Overview

EMAC is a generic construction for a block cipher-based message authentication scheme, proposed in 1997 by Erez Petrank and Charles Rackoff [518]. EMAC was not formally submitted to NESSIE but has been included in the evaluation as existing standard. It is one of the message authentication schemes included in the ISO/IEC standard 9797-1 [303]. The EMAC scheme is a variant of the CBC-MAC, that is a message authentication code based on the CBC-mode (Cipher Block Chaining) of a block cipher. EMAC uses a block cipher as a black box, so that existing code can be reused and the underlying block cipher can be easily replaced. NESSIE recommends the use of one of the 128-bit block ciphers described in Part B of the NESSIE portfolio (AES or Camellia) as underlying block cipher for EMAC.

3.1.2 Outline of the primitive

Suppose that a 128-bit block cipher is given and let $\mathbf{E}_K$ denote encryption with the block cipher using a q-bit key K. Then the EMAC scheme based on this block cipher maps a message M consisting of an arbitrary number of bits onto an 128-bit block $\mathbf{EMAC}(K, M)$, under control of a key K of q bits long.

First, the message is expanded to an appropriate length and divided into message blocks $M_0 \ldots M_{t-1}$ where each block M_i ($0 \leq i \leq t-1$) consists of 128 bits. These message blocks M_i are encrypted by the block cipher (with key K) in CBC-mode and there is an additional encryption at the end using a different key K'. The outline of EMAC is as follows.

1. Expand the message M and divide it into message blocks $M_0 \ldots M_{t-1}$.
2. Derive a secondary key K' from the key K by complementing the first four bits of every byte of K.
3. Set the initial value to zero: $H_0 = 0$.
4. For $0 \leq i \leq t-1$: compute $H_{i+1} = \mathbf{E}_K(H_i \oplus M_i)$.
5. Compute the output transformation: $\mathbf{EMAC}(K, M) = \mathbf{E}_{K'}(H_t)$.

3.1.3 Security and performance

The security of EMAC is closely related to the security of the underlying block cipher. In [518] Petrank and Rackoff give a security proof for EMAC based on the assumption that the underlying block cipher is pseudo-random. The security level depends on the size of the internal state (this is equal to the block size of the cipher) and on the lengths of the key and message authentication value.

The EMAC construction has the advantage that it allows the reuse of an existing block cipher implementation. The performance of EMAC is close to the performance of the block cipher that is used, except in the case of short messages when there is a significant overhead (this is for messages of a few blocks, where the block length is 128 bits). The overhead occurs because of the extra encryption in the output transformation. Changing the key for EMAC requires two executions of the key schedule of the block cipher.

3.2 Description

Let $\mathbf{E}_K$ denote encryption with a 128-bit block cipher using a q-bit key K. Both the input and output of $\mathbf{E}_K$ are 128 bits long. We suggest to use a 128-bit block cipher from the NESSIE portfolio, that is $\mathbf{E}$ = AES or Camellia. We refer to Part B for the specification of these block ciphers. Note that both AES and Camellia have $q = 128$, 192, or 256.

Let $M = (m_0, m_1, \ldots, m_{n-1})$ be a message consisting of n bits, and let $K = (k_0, k_1, \ldots, k_{q-1})$ denote a q-bit secret key. The message authentication value $\mathbf{EMAC}(K, M)$ is computed as follows.

Message expansion. Let $t = (n \text{ div } 128) + 1$. First of all, the n-bit message M is expanded to a $128t$-bit message $M = (m_0, m_1, \ldots, m_{128t-1})$. This is done by appending the following r bits to the original message, where $1 \le r \le 128$ and $n + r = 128t$, a multiple of 128 bits:

$$\begin{aligned} m_n &:= 1\,, \\ m_{n+1} := m_{n+2} := \cdots := m_{n+r-1} &:= 0\,. \end{aligned}$$

In other words, append a single 1-bit and a number of 0-bits until the length of the expanded message is a multiple of 128 bits. Note that the padding is always done, even if the message length is already a multiple of 128 bits. In that case, the message is expanded by 128 bits.

Next, the expanded message is divided into t message blocks $M_0, M_1, \ldots, M_{t-1}$ where each block M_i $(0 \le i \le t-1)$ consists of 128 bits:

$$M_i = (m_{128i}, m_{128i+1}, \ldots, m_{128i+127})\,.$$

Secondary key. Compute a secondary key $K' = (k'_0, k'_1, \ldots, k'_{q-1})$ consisting of q bits, by complementing alternate substrings of four bits of K starting with the first (leftmost) four bits:

$$k'_{4i+j} := \bar{k}_{4i+j} \quad \text{for } j = 0,1,2,3 \text{ and } i = 0,2,\ldots \text{ (even)},$$
$$k'_{4i+j} := k_{4i+j} \quad \text{for } j = 0,1,2,3 \text{ and } i = 1,3,\ldots \text{ (odd)}.$$

In hexadecimal notation this is equivalent to:

$$K' := K \oplus \texttt{F0F0...F0}_{\texttt{x}}\,.$$

Compression. The message M is compressed using the block cipher $\mathbf{E}$ with the key K in CBC-mode (Cipher Block Chaining), according to the following steps:

– Define a 128-bit initial value H_0 by setting this value equal to zero.

$$H_0 := \texttt{0000...00}_{\texttt{x}}\,.$$

– For $i = 0,\ldots,t-1$ compute a new 128-bit value H_{i+1} from the value H_i and the message block M_i:

$$H_{i+1} := \mathbf{E}_K(H_i \oplus M_i)\,.$$

That is, compute the bitwise exclusive-or (XOR) of the previously obtained value H_i and the message block M_i and encrypt $H_i \oplus M_i$ with the block cipher $\mathbf{E}$ using the key K. The resulting value H_{i+1} is 128 bits long.

Output transformation. Encrypt the final value H_t obtained from the compression stage with the block cipher $\mathbf{E}$ using the secondary key K'. The result H_{out} is 128 bits long.

$$H_{out} := \mathbf{E}_{K'}(H_t)\,.$$

Optional truncation. If desired one may optionally truncate the output of the previous step to p bits, by selecting the p leftmost bits of H_{out} ($p < 128$).

$$H_{trunc} := p \vdash H_{out}\,.$$

Message authentication value. The default message authentication value $\mathbf{EMAC}(K,M)$ is the 128-bit string corresponding to the value H_{out}.

$$\mathbf{EMAC}(K,M) := H_{out}\,.$$

If desired, message authentication values of a shorter length $p < 128$ can be generated by applying the truncation operation. In this case, the truncated message authentication value $\mathbf{EMAC}_p(K,M)$ is the p-bit string corresponding to the value H_{trunc}.

$$\mathbf{EMAC}_p(K,M) := H_{trunc}\,.$$

3.3 Test vectors for EMAC using AES

The following test vectors are generated for EMAC using AES (128-bit key) as underlying block cipher, with the default output length of 128 bits. The test vectors are denoted as triples (`message`, `key`, `MAC`). This correponds to `MAC` $=$ $\mathbf{EMAC}$(`key`, `message`). Two sets of test vectors are given, for two different values of the key. Note that in practice one should use a random secret key that is generated by a secure key generation algorithm.

The key and MAC are denoted as hexadecimal numbers of 16 bytes (128 bits). The message is denoted as an ASCII string (e.g., the character "a" corresponds to the hexadecimal byte $\texttt{61}_{\texttt{x}}$).

3.3.1 Set 1

```
message = "" (empty string)
key = 00112233445566778899AABBCCDDEEFF
MAC = CAC6EB80AD0FC891CCE693B8BF587063

message = "a"
key = 00112233445566778899AABBCCDDEEFF
MAC = 644A572D2F05EB7FA77290ABFCF048A7

message = "abc"
key = 00112233445566778899AABBCCDDEEFF
MAC = D55DD2CA1C89E488F62A5F694357B517

message = "message digest"
key = 00112233445566778899AABBCCDDEEFF
MAC = 1B62443A4740E470A6FC858F1F7B9053

message = "abcdefghijklmnopqrstuvwxyz"
key = 00112233445566778899AABBCCDDEEFF
MAC = 691A07CA6E0CD6EEF39524DCB0434361

message = "abcdbcdecdefdefgefghfghighij
           hijkijkljklmklmnlmnomnopnopq"
key = 00112233445566778899AABBCCDDEEFF
MAC = 0582D1FB4C7E830C976EED9116DBAD49

message = "A...Za...z0...9"
key = 00112233445566778899AABBCCDDEEFF
MAC = BF6C86A90D71B083045EEE09AD7D706D

message = 8 times "1234567890"
key = 00112233445566778899AABBCCDDEEFF
MAC = 444D6404975C32D3DF715A8C50FCB20F

message = "Now is the time for all "
key = 00112233445566778899AABBCCDDEEFF
MAC = 08B8167B1A02BECC09B1E43974F5CEB0

message = "Now is the time for it"
key = 00112233445566778899AABBCCDDEEFF
MAC = C0953563FB7C979CC389EF2F49521B85

message = 1 million times "a"
key = 00112233445566778899AABBCCDDEEFF
MAC = 579EDDB29886AC068C08D4EEFB1AB076
```

3.3.2 Set 2

```
message = "" (empty string)
key = 0123456789ABCDEF0123456789ABCDEF
MAC = 33BD8D7DE983CD8D452695152A53AE8A

message = "a"
key = 0123456789ABCDEF0123456789ABCDEF
MAC = DC525ADE636062644FA0B53468FFAB7E

message = "abc"
key = 0123456789ABCDEF0123456789ABCDEF
MAC = BDF207BB71B862988C3ED0DCED005460

message = "message digest"
key = 0123456789ABCDEF0123456789ABCDEF
MAC = B6D4B0F97C0B5AD662CAC18BBD5514DA

message = "abcdefghijklmnopqrstuvwxyz"
key = 0123456789ABCDEF0123456789ABCDEF
MAC = 41256D33451F4CFDB1845BD7F26C8CA3

message = "abcdbcdecdefdefgefghfghighij
           hijkijkljklmklmnlmnomnopnopq"
key = 0123456789ABCDEF0123456789ABCDEF
MAC = EE83D79574167C7A139047156D2B26BD

message = "A...Za...z0...9"
key = 0123456789ABCDEF0123456789ABCDEF
MAC = 497C2A4E4DA3C237F69779E084489A6D

message = 8 times "1234567890"
key = 0123456789ABCDEF0123456789ABCDEF
MAC = 4C76C4509DBE50FAD2B32D969C082677

message = "Now is the time for all "
key = 0123456789ABCDEF0123456789ABCDEF
MAC = D1F8838FDE9D149D897C0470478774A5

message = "Now is the time for it"
key = 0123456789ABCDEF0123456789ABCDEF
MAC = CEBB575BDA40FF9BBF53557DC352D328

message = 1 million times "a"
key = 0123456789ABCDEF0123456789ABCDEF
MAC = 3DDFDF81FA36B6A5E7A2DCFD50F590C3
```

4. HMAC

4.1 Introduction

4.1.1 Overview

HMAC is a generic construction for a hash-based message authentication scheme, proposed in 1996 by Mihir Bellare, Ran Canetti and Hugo Krawczyk [44]. HMAC was not formally submitted to NESSIE but has been included in the evaluation as existing standard. The HMAC scheme using SHA-1 as underlying hash function has been standardised by NIST as FIPS-198 [473], and the generic scheme is included in the ISO/IEC standard 9797-2 [304]. HMAC uses a hash function (or its compression function) as a black box, so that existing code can be reused and the underlying hash function can be easily replaced. NESSIE recommends the use of one of the collision-resistant hash functions described in Part C of the NESSIE portfolio (Whirlpool, SHA-256 or SHA-512) as underlying hash function for HMAC.

4.1.2 Outline of the primitive

Suppose that a hash function $\mathbf{H}$ with l-bit output is given, and that this hash function is based on the iteration of a compression function $\mathbf{F}$ that processes message blocks of b bits. Then the HMAC scheme based on this hash function maps a message M consisting of an arbitrary number of bits onto an l-bit block $\mathbf{HMAC}(K, M)$, under control of a key K up to b bits long (and not shorter than l bits).

1. If necessary, expand the key K so that it is b bits long. Compute K_1 by bitwise exclusive-or of K and a b-bit constant C_1; compute K_2 by bitwise exclusive-or of K and a b-bit constant C_2. Note that K_1 and K_2 are b bits long.
2. Use the hash function $\mathbf{H}$ to compress the concatenation of K_1 and the message M: $\mathbf{H}(K_1 \| M)$. The result is l bits long.
3. Use the hash function $\mathbf{H}$ to compress the concatenation of K_2 and the result from step 2: $\mathbf{HMAC}(K, M) = \mathbf{H}(K_2 \,\|\, \mathbf{H}(K_1 \,\|\, M))$. The resulting message authentication value is l bits long.

4.1.3 Security and performance

The security of HMAC is closely related to the security of the underlying hash function **H**. In [44] Bellare, Canetti and Krawczyk give a security proof for HMAC based on the following assumptions on the underlying hash function **H** and the compression function **F** (where **H** is an iterated construction applying **F** on message blocks of a fixed length):

- The hash function **H** is collision-resistant when the initial value is secret.
- The compression function **F** keyed by the initial value is a strong MAC algorithm (this means that its output is hard to predict).
- The values $\mathbf{F}(K \oplus C_1)$ and $\mathbf{F}(K \oplus C_2)$ cannot be distinguished from truly random values. This means that the compression function **F** is a 'weak' pseudorandom function ('weak' because the opponent has no direct access to K).

The security level depends on the size of the internal state (this is equal to the output size of the hash function), on the length of the key and on the length of the message authentication value.

The HMAC construction has the advantage that it allows the reuse of an existing hash function implementation. The performance of HMAC is close to the performance of the hash function that is used, except in the case of short messages when there is a significant overhead (this is for messages of a few blocks; the block length depends on the compression function and is 512 bits for Whirlpool and SHA-256, and 1024 bits for SHA-512). The overhead occurs because of step 3 in the outline of Sect. 4.1.2. Changing the key for HMAC requires two computations of the compression function.

4.2 Description

Let **H** be a collision-resistant hash function with an output length of l bits, and based on the iteration of a compression function **F** that works on message blocks of b bits long. We suggest to use a collision-resistant hash function from the NESSIE portfolio, that is **H** = Whirlpool, SHA-256 or SHA-512. We refer to Part C for the specification of these hash functions. Note that Whirlpool has $l = 512, b = 512$; SHA-256 has $l = 256, b = 512$; SHA-512 has $l = 512, b = 1024$.

Let $M = (m_0, m_1, \ldots, m_{n-1})$ be a message consisting of n bits, and let $K = (k_0, k_1, \ldots, k_{q-1})$ denote a q-bit secret key with $l \leq q \leq b$. The message authentication value $\mathbf{HMAC}(K, M)$ is computed as follows.

key expansion. If the length of the key is shorter than one message block for the compression function ($q < b$), $b - q$ 0-bits are appended to the key. This results in a key $K = (k_0, k_1, \ldots, k_{b-1})$ of b bits long.

$$k_q := k_{q+1} := \cdots := k_{b-1} := 0\,.$$

additive constants. Define the following two b-bit constants (hexadecimal notation):

$$C_1 := \texttt{3636...36}_{\mathrm{x}}\ ,\ C_2 := \texttt{5C5C...5C}_{\mathrm{x}}\,.$$

inner computation of H. Use the hash function **H** to compute

$$H_1 := \mathbf{H}(K \oplus C_1 \,\|\, M)\,.$$

That is, compute the bitwise exclusive-or (XOR) of the key K and the constant C_1, concatenate $K \oplus C_1$ (this corresponds to a b-bit string) with the message M (an n-bit string) and apply the hash function **H** on the concatenated string of $b + n$ bits. The result H_1 is l bits long.

outer computation of H. Use the hash function **H** to compute

$$H_2 := \mathbf{H}(K \oplus C_2 \,\|\, H_1)\,.$$

That is, compute the bitwise exclusive-or (XOR) of the key K and the constant C_2, concatenate $K \oplus C_2$ (this corresponds to a b-bit string) with the value H_1 (this corresponds to an l-bit string) and apply the hash function **H** on the concatenated string of $b + l$ bits. The result H_2 is l bits long.

optional truncation. If desired one may optionally truncate the output of the previous step to p bits, by selecting the p leftmost bits of H_2 ($p < l$).

$$H_{trunc} := p \vdash H_2\,.$$

message authentication value. The default message authentication value $\mathbf{HMAC}(K, M)$ is the l-bit string corresponding to the value H_2.

$$\mathbf{HMAC}(K, M) := H_2\,.$$

If desired, message authentication values of a shorter length $p < l$ can be generated by applying the truncation operation. In this case, the truncated message authentication value $\mathbf{HMAC}_p(K, M)$ is the p-bit string corresponding to the value H_{trunc}.

$$\mathbf{HMAC}_p(K, M) := H_{trunc}\,.$$

4.2.1 Implementation note

Both the inner computation and the outer computation of the hash function **H** start with a message block (b bits) depending on the key K and a constant C_1 or C_2. This message block will be the first input to the compression function **F** used by the hash function in an iterative manner. To speed up the computation of HMAC, the values $\mathbf{F}(K \oplus C_1)$ and $\mathbf{F}(K \oplus C_2)$ can be precomputed once (at the time of generation of the key K), stored in memory and used to initialise the inner and outer computation of **H** for every message that is authenticated with the key K. This method saves the application of the compression function on two b-byte blocks $K \oplus C_1$ and $K \oplus C_2$ for each message authenticated with this key. Here we assume that the implementation has access to the code of the compression function, and not only to the code of the hash function.

4.3 Test vectors for HMAC using SHA-256

The following test vectors are generated for HMAC using SHA-256 as underlying hash function and a key of 512 bits, with the default output length of 256 bits. The test vectors are denoted as triples (`message`, `key`, `MAC`). This correponds to `MAC` = **HMAC**(`key`, `message`). Two sets of test vectors are given, for two different values of the `key`. Note that in practice one should use a random secret `key` that is generated by a secure key generation algorithm.

The `key` and `MAC` are denoted as hexadecimal numbers of 64 bytes (512 bits) and 32 bytes (256 bits) respectively. The `message` is denoted as an ASCII string (e.g., the character `"a"` corresponds to the hexadecimal byte $\texttt{61}_{\texttt{x}}$). Note that in SHA-256 the big-endian convention is used for conversion between bytes and four-byte words (e.g., the message string `"abcd"` corresponds to the hexadecimal word $\texttt{61626364}_{\texttt{x}}$).

4.3.1 Set 1

```
message = "" (empty string)
key = 00112233445566778899AABBCCDDEEFF
      0123456789ABCDEF0011223344556677
      8899AABBCCDDEEFF0123456789ABCDEF
      00112233445566778899AABBCCDDEEFF
MAC = B379757F089F3EC3B41BFD184048AB43
      6FAE7E09F8C7E3461C825ED37E544303

message = "a"
key = 00112233445566778899AABBCCDDEEFF
      0123456789ABCDEF0011223344556677
      8899AABBCCDDEEFF0123456789ABCDEF
      00112233445566778899AABBCCDDEEFF
MAC = FAED6AEB172995459B58DBAF70AF44A1
      D31E94D6EB45A7A33631AB5DE415C459

message = "abc"
key = 00112233445566778899AABBCCDDEEFF
      0123456789ABCDEF0011223344556677
      8899AABBCCDDEEFF0123456789ABCDEF
      00112233445566778899AABBCCDDEEFF
MAC = 742CB1AE5D0949615F9D2866EFB1BE21
      2C766C96C44AAF1902E10740B19DEBF5

message = "message digest"
key = 00112233445566778899AABBCCDDEEFF
      0123456789ABCDEF0011223344556677
      8899AABBCCDDEEFF0123456789ABCDEF
      00112233445566778899AABBCCDDEEFF
MAC = 300EC2BDCFECD094802B31B20A8CA9CF
      E3BDDAB867963BE2F39B87D70BC02817
```

```
message = "abcdefghijklmnopqrstuvwxyz"
key = 00112233445566778899AABBCCDDEEFF
      0123456789ABCDEF0011223344556677
      8899AABBCCDDEEFF0123456789ABCDEF
      00112233445566778899AABBCCDDEEFF
MAC = 038045116F21FEBF34FDC678B8E87982
      9DF1AB83DBBFC1105A762029F3CD934C

message = "abcdbcdecdefdefgefghfghighij
           hijkijkljklmklmnlmnomnopnopq"
key = 00112233445566778899AABBCCDDEEFF
      0123456789ABCDEF0011223344556677
      8899AABBCCDDEEFF0123456789ABCDEF
      00112233445566778899AABBCCDDEEFF
MAC = AB65232F3BD2450297264E3FD828E1DD
      4666BEB965C54CF842A4AD5E453192A6

message = "A...Za...z0...9"
key = 00112233445566778899AABBCCDDEEFF
      0123456789ABCDEF0011223344556677
      8899AABBCCDDEEFF0123456789ABCDEF
      00112233445566778899AABBCCDDEEFF
MAC = 76FA502215729DB8B77D3799B221A8FD
      B538393947A16730B5E84FC2837635C6

message = 8 times "1234567890"
key = 00112233445566778899AABBCCDDEEFF
      0123456789ABCDEF0011223344556677
      8899AABBCCDDEEFF0123456789ABCDEF
      00112233445566778899AABBCCDDEEFF
MAC = 5DEDB7B986F4A9B2ECA41D8BA42A4723
      6DA659D76A4DFCAAE4E82E1BED9D6068

message = "Now is the time for all "
key = 00112233445566778899AABBCCDDEEFF
      0123456789ABCDEF0011223344556677
      8899AABBCCDDEEFF0123456789ABCDEF
      00112233445566778899AABBCCDDEEFF
MAC = 37781AF322155A96B9D902CED73A71B7
      13C8A2E26F67E38DCB7A269BAD1F4184

message = "Now is the time for it"
key = 00112233445566778899AABBCCDDEEFF
      0123456789ABCDEF0011223344556677
      8899AABBCCDDEEFF0123456789ABCDEF
      00112233445566778899AABBCCDDEEFF
MAC = CC60A0459AEA28348CBC27FF9CDD4204
      19A7872107C0F5DFE3D6A43DB7D72445
```

```
message = 1 million times "a"
key = 00112233445566778899AABBCCDDEEFF
      0123456789ABCDEF0011223344556677
      8899AABBCCDDEEFF0123456789ABCDEF
      00112233445566778899AABBCCDDEEFF
MAC = BB459133294AF921181055854C16B063
      36E9B0514761BA2396B8A0A8028393BA
```

4.3.2 Set 2

```
message = "" (empty string)
key = 0123456789ABCDEF0123456789ABCDEF
      0123456789ABCDEF0123456789ABCDEF
      0123456789ABCDEF0123456789ABCDEF
      0123456789ABCDEF0123456789ABCDEF
MAC = 5FE3C3165038E39BB4EDD8F359DEE4EC
      498B42EAC720DCE4AE3C72A59004C864

message = "a"
key = 0123456789ABCDEF0123456789ABCDEF
      0123456789ABCDEF0123456789ABCDEF
      0123456789ABCDEF0123456789ABCDEF
      0123456789ABCDEF0123456789ABCDEF
MAC = 3E68B80599468985E22387166FC3CE56
      FB3A163C2FA74DA24E1EE60646620D31

message = "abc"
key = 0123456789ABCDEF0123456789ABCDEF
      0123456789ABCDEF0123456789ABCDEF
      0123456789ABCDEF0123456789ABCDEF
      0123456789ABCDEF0123456789ABCDEF
MAC = 3A2232B8A45BAC227251703663B6C127
      049A44301EC1CC94BBF137DD803BA5F3

message = "message digest"
key = 0123456789ABCDEF0123456789ABCDEF
      0123456789ABCDEF0123456789ABCDEF
      0123456789ABCDEF0123456789ABCDEF
      0123456789ABCDEF0123456789ABCDEF
MAC = 63D6AD7C0708C100C8E3762F38290D8D
      26B7685C108B838EEE1E4BB30EAA93DD

message = "abcdefghijklmnopqrstuvwxyz"
key = 0123456789ABCDEF0123456789ABCDEF
      0123456789ABCDEF0123456789ABCDEF
      0123456789ABCDEF0123456789ABCDEF
      0123456789ABCDEF0123456789ABCDEF
```

```
MAC = B2E53839554FD9DC715D72CAB7A1EB9A
      4375FFB195711AAD59A6B29BF91BF33E

message = "abcdbcdecdefdefgefghfghighij
           hijkijkljklmklmnlmnomnopnopq"
key = 0123456789ABCDEF0123456789ABCDEF
      0123456789ABCDEF0123456789ABCDEF
      0123456789ABCDEF0123456789ABCDEF
      0123456789ABCDEF0123456789ABCDEF
MAC = 52B1479DC3283500CEA71E3B1AD7BF96
      425A1669F22CC81F89CC0BE89CEF89B7

message = "A...Za...z0...9"
key = 0123456789ABCDEF0123456789ABCDEF
      0123456789ABCDEF0123456789ABCDEF
      0123456789ABCDEF0123456789ABCDEF
      0123456789ABCDEF0123456789ABCDEF
MAC = CD6A1D7D07CEAA8770F5617A39A3369B
      3993759C33B9B02483AF573AB4CAE1DF

message = 8 times "1234567890"
key = 0123456789ABCDEF0123456789ABCDEF
      0123456789ABCDEF0123456789ABCDEF
      0123456789ABCDEF0123456789ABCDEF
      0123456789ABCDEF0123456789ABCDEF
MAC = D2BC3DAC84555DD0C27BFF9FA17CB1B3
      7BA9CE77C1B7A2C1E7DFD192B12BF625

message = "Now is the time for all "
key = 0123456789ABCDEF0123456789ABCDEF
      0123456789ABCDEF0123456789ABCDEF
      0123456789ABCDEF0123456789ABCDEF
      0123456789ABCDEF0123456789ABCDEF
MAC = 42014276732OC2C9EC8E0BC6EB6C25CD
      A6342CD94F881200F90D7AF948F9AC53

message = "Now is the time for it"
key = 0123456789ABCDEF0123456789ABCDEF
      0123456789ABCDEF0123456789ABCDEF
      0123456789ABCDEF0123456789ABCDEF
      0123456789ABCDEF0123456789ABCDEF
MAC = C4BE6C24D373A293AF43CB500B1DBCCE
      61755F5B5E064FB7B08136F029FC90BF

message = 1 million times "a"
key = 0123456789ABCDEF0123456789ABCDEF
      0123456789ABCDEF0123456789ABCDEF
      0123456789ABCDEF0123456789ABCDEF
      0123456789ABCDEF0123456789ABCDEF
MAC = 50008B8DC7ED3926936347FDC1A01E9D
      5220C6CC4B038B482C0F28A4CD88CA37
```

Part E

Asymmetric encryption schemes

1. PSEC-KEM

1.1 Introduction

1.1.1 Overview

PSEC-KEM is a key encapsulation mechanism based on the security of the Diffie-Hellman key agreement protocol on elliptic curves. The Diffie-Hellman key agreement protocol [201] was developed by Whitfield Diffie and Martin Hellman in 1976, and was the first example of an asymmetric cryptosystem. Since then it has been the basis for many of the most popular and efficient cryptosystems used today. The use of elliptic curve groups for cryptography was first suggested simultaneously by Koblitz [373] and Miller [446].

PSEC-KEM was submitted to the NESSIE project by the Nippon Telegraph and Telephone (NTT) Corporation of Japan [238]. It is also included in the draft ISO/IEC standard on asymmetric encryption, ISO/IEC 18033-2 [312, 584].

1.1.2 Outline of the primitive

The action of PSEC-KEM takes place in a prime order cyclic subgroup of an elliptic curve group, generated by a point P. The public-private key pair for the scheme is the same as in a Diffie-Hellman scheme, i.e. the public key consists of the point P (sometimes thought of as a parameter of the system) and a point $W = sP$ for some randomly generated s, and the private key is s. PSEC-KEM produces symmetric keys of some pre-specified length $KeyLen$. The scheme makes use of a key derivation function $KDF(\cdot)$, which maps octets strings of an arbitrary length to octet strings of length $KeyLen$ (see Sect. 3).

The encapsulation algorithm takes a fixed length random seed r and the public key as input. It works in several stages. First it generates a random symmetric key K and a random integer α suitable for use with the Diffie-Hellman protocol from the random seed r, using a key derivation function. It then runs the Diffie-Hellman protocol to generate the first part of the encapsulation $C_1 = \alpha P$ and a mask seed $Q = \alpha W$. It uses both Q and C_1 to generate a mask M for the initial seed r via a mask generating function. Lastly it computes the second part of the encapsulation C_2 by XORing the seed r with the mask M.

The decapsulation algorithm takes as input both parts, C_1 and C_2, of an encapsulated key and the private key, and outputs the symmetric key K. It also works in several stages. First it recovers the mask seed Q using the Diffie-Hellman protocol, i.e. setting $Q = sC_1$. It then computes the mask M from Q and C_1 via the mask generating function. It can then compute the seed r by XORing the second part of the encapsulation C_2 with the mask M. Having found r, the decapsulation algorithm can apply the relevant key derivation function to find both the secret key K and the random integer α used by the encapsulation algorithm in the Diffie-Hellman protocol. The decapsulation algorithm releases the symmetric key K only if the first part of the encapsulation was correctly formed from α, i.e. only if $C_1 = \alpha P$.

1.1.3 Security and performance

There is a security proof for PSEC-KEM that uses a distinguishing attacker in the adaptive chosen ciphertext model (IND-CCA2 model) to construct a short list of solutions for a computational Diffie-Hellman problem [195, 584]. The proof uses the random oracle model and has a tight reduction. This proof is a strong security argument. It suggests that, given an encapsulation of a key, the best technique to distinguish between the real key and a randomly generated key is to invert the instance of the Diffie-Hellman protocol used in the encapsulation.

Side channel attacks are a problem for any asymmetric encryption scheme based on elliptic curve groups and PSEC-KEM is no exception. It is also vulnerable to a fault attack. However it does not appear to be any more vulnerable to these attacks then any other elliptic curve based scheme and countermeasures exist for all of these attacks. It is unclear to what extent the attacks of Lim and Lee [399], Biehl *et al.* [58] and Antipa *et al.* [23] are effective against PSEC-KEM; however it is recommended that all key and ciphertext elliptic curve points are validated before use.

The performance characteristics for PSEC-KEM are good. The encapsulation algorithm appears to be close to optimal for a Diffie-Hellman based primitive. The decapsulation algorithm is slower: requiring two scalar multiplications rather than one. This extra computational overhead is necessary to provide a stronger security proof.

The main advantage of PSEC-KEM is its flexibility. The scheme gives a good performance profile with a high level of security, and should be suitable for almost all applications.

1.2 Description

1.2.1 Parameter generation

First, a security parameter should be chosen. For a security parameter k, the order of the cyclic group should be an integer whose length (in octets) is greater than or equal to the value given in the following table.

k	72	80	112	128
Size of subgroup order	18	20	28	32

The scheme should be defined over a finite field $\mathbb{F}_q$ of characteristic 2 or over a prime field. An elliptic curve $E(\mathbb{F}_q)$ should be chosen on which the computational Diffie-Hellman problem is intractable. A point P on $E(\mathbb{F}_q)$ needs to be selected where P has prime order p and the length of p corresponds to the value defined in the table above for the security parameter k. Let $l(k)$ be the length of p, the order of the elliptic curve subgroup.

The length of the symmetric key produced by the KEM, $KeyLen$, is defined by the requirements of the DEM.

1.2.2 Key generation

Key generation for PSEC-KEM is the same as for any Diffie-Hellman based scheme.

Description:

Input: A security parameter 1^k
A fixed length random string r
An elliptic curve $E(\mathbb{F}_q)$
A point P on the elliptic curve $E(\mathbb{F}_q)$
The order of P, p

Output: A PSEC-KEM public key pk
A PSEC-KEM secret key sk

1. Verify p is a prime. If not, output "error" and abort.
2. Verify the fact that P is a point on the elliptic curve $E(\mathbb{F}_q)$ and that P has order p. If not, output "error" and abort.
3. Generate an integer s in the range $[2, p-1]$ using the random seed r in such a way that every possible value of s will be generated with approximately equal probability.
4. Set $W := sP$.
5. Set $pk := (W)$.
6. Set $sk := (s)$.
7. Output the public key pk and the secret key sk.

1.2.3 Encapsulation algorithm

To generate a symmetric key of length $KeyLen$ and an encapsulation of that key the following algorithm is executed.

Description:

Input: A PSEC-KEM public key $pk = (W)$
A fixed length random string r
The length of the random string r, $rLen$
An elliptic curve $E(\mathbb{F}_q)$
A point P on the elliptic curve $E(\mathbb{F}_q)$
The order of P, p
The size of required symmetric key $KeyLen$

Output: A symmetric key K of length $KeyLen$
An encapsulation C

1. Validate the public key and the system parameters.
2. Set $H := KDF(\mathsf{I2OSP}(0,4)||r, KeyLen + l(k) + 16)$.
3. Parse H as $t||K$ where t is an octet string of length $l(k) + 16$ and K is an octet string of length $KeyLen$.
4. Set $\alpha := \mathsf{OS2IP}(t) \bmod p$.
5. Set $Q := \alpha W$.
6. Set $C_1 := \mathsf{ECP2OSP}(\alpha P)$.
7. Set $C_2 := KDF(\mathsf{I2OSP}(1,4)||C_1||\mathsf{ECP2OSP}(Q), rLen) \oplus r$.
8. Set $C := (C_1, C_2)$.
9. Output the symmetric key K and the encapsulation C.

1.2.4 Decapsulation algorithm

To decapsulate an encapsulated key of length $KeyLen$ the following algorithm is executed.

Description:

Input: A PSEC-KEM private key $sk = (s)$
An encapsuled key C
An elliptic curve $E(\mathbb{F}_q)$
A point P on the elliptic curve $E(\mathbb{F}_q)$
The order of P, p
The size of required symmetric key $KeyLen$
Output: A symmetric key K of length $KeyLen$

1. Validate the private key and the system parameters.
2. Parse C as (C_1, C_2).
3. Set $X := \mathsf{OS2ECPP}(C_1)$.
4. Verify that X is a point on $E(\mathbb{F}_q)$ and that X lies in the subgroup generated by P. If not, output "error" and abort.
5. Set $rLen$ to be the length of C_2.
6. Set $Q := sX$.
7. Set $r := C_2 \oplus KDF(\mathsf{I2OSP}(1,4)||C_1||\mathsf{ECP2OSP}(Q), rLen)$.
8. Set $H := KDF(\mathsf{I2OSP}(0,4)||r, KeyLen + l(k) + 16)$.
9. Parse H as $t||K$ where t is an octet string of length $l(k) + 16$ and K is an octet string of length $KeyLen$.
10. Set $\alpha := \mathsf{OS2IP}(t) \bmod p$.
11. Check $C_1 = \mathsf{ECP2OSP}(\alpha P)$. If not, output "error" and abort.
12. Output K.

1.2.5 Guidelines for implementation

It is important to validate all inputs to the encapsulation and decapsulation algorithms before they are used. In the case of key material and system parameters, this prevents chosen modulus attacks and fault attacks.

In particular it is necessary to trust the properties of the elliptic curve points. So, for both the encapsulation and decapsulation algorithm it is necessary to trust that

– p is a prime number whose length is as defined by the security parameter and,
– P is a point on the elliptic curve $E(\mathbb{F}_q)$ and P has order p.

If these conditions do not hold then the algorithm should output "error" and abort. The two obvious methods to validate the system parameters are to check that the properties hold during the execution of the algorithm (which may be computationally expensive), or to have some kind of certified trust mechanism in place so the algorithm can trust that the system parameters have been correctly validated by some third party.

It is also necessary, in the encapsulation algorithm, to check the properties of the elliptic curve point W given in the public key. In particular it is necessary to check that W is a point that lies on the elliptic curve $E(\mathbb{F}_q)$ and that W lies in the subgroup generated by P.

1.3 Test vectors

The following test vectors were generated by PSEC-KEM with a security parameter $k = 80$. The algorithms use the key derivation function $Mech1$ (see Sect. 3) with the hash function SHA-1. The algorithm was run on an elliptic curve $E(\mathbb{F}_q)$ where q is the prime number

$$q = \texttt{8223961F0E209569238C61C0801A3C2B2634F651}$$

and the Weierstrass equation of E is $y^2 = x^3 + ax + b$ where

$$a = \texttt{43655667BBC2C6818E67576128B54D8910E81E38}$$
$$b = \texttt{585ED69A1A5E3AF3549DF5829428663C08677BC3}$$

We chose the point P to be the elliptic curve point with co-ordinates (x_P, y_P) where

$$x_P = \texttt{44A82D7665486DE731466714B05403C3C8852A1D}$$
$$y_P = \texttt{2CEFA73540C723F2B25E5E8EC1E98B0E17B6F0B8}$$

The point P has prime order p where $p = q$. Hence p has length 20 (octets); this corresponds to a security parameter $k = 80$. The algorithm was asked to produce keys of length $KeyLen = 16$ octets.

The test vectors were generated using a public key (W) where $W = (x_W, y_W)$ is an elliptic curve point and

$$x_W = \texttt{16741D8B8750ED51BCD8D30CF5A86B06BFA8F022}$$
$$y_W = \texttt{7A98E5DE25931089B03F8D9D4BDB471CDCBEC57A}$$

The corresponding private key was (s) where

$$s = \texttt{81117F9AED24B2E00A234D9919BE9E8094CFE905}$$

Elliptic curve points were stored in an uncompressed format.

1.3.1 Test vectors for key encapsulation

The following are the correct results of the key encapsulation algorithm being run with a fixed length random seed r, producing a symmetric key K and an encapsulated key $C = (C_1, C_2)$.

r = 30313233343536373839616263646566 676A68696A6B6C6D6E6F707172737475

K = 09AD82F3600D4D0CB420FF032A21E7A4

C_1 = 0472500F21F2F048D3A226EECA8F1635 A611E5F9131912366334 8E7E758DE3D9
53B9C547DD5ACB13B5

C_2 = 199D62F5C8403009AC30DC2B2B9CA63A D200E3D69CD51AA4C758E11B2F09AEC7

1.3.2 Test vectors for key decapsulation

The following key encapsulation $C = (C_1, C_2)$ correctly decapsulates to give the symmetric key K.

C_1 = 0426992349860E2AF901E34E517D3011 D454239C7E208EA94F29B4F82F09C3F3
0DAE21795A2F7DE48D

C_2 = 3C591D799AF57710E04A978B6C068900 14178290FAACDF6487180F94BC81FC10

K = 264E3A9B8501D9D7BF9AB5F14E9CDE09

The following key encapsulation $C = (C_1, C_2)$ fails to decapsulate owing to step 11 of the decapsulation algorithm (failure of the integrity check).

C_1 = 0426992349860E2AF901E34E517D3011 D454239C7E208EA94F29B4F82F09C3F3
0DAE21795A2F7DE48D

C_2 = 11111111111111111111111111111111 11111111111111111111111111111111

2. RSA-KEM

2.1 Introduction

2.1.1 Overview

RSA-KEM is a key encapsulation mechanism based on the famous RSA encryption algorithm. The RSA encryption algorithm was proposed in 1978 by Ronald L. Rivest, Adi Shamir and Leonard M. Adleman [543]. The KEM-DEM method for constructing a hybrid asymmetric encryption algorithm was proposed by Ronald Cramer and Victor Shoup [171] and has been widely adopted by developers and standards bodies. RSA-KEM was not formally submitted to NESSIE but has been included in the evaluation as a *de facto* standard for a KEM-DEM based hybrid encryption scheme and as an algorithm which is included in the draft ISO standard on asymmetric encryption, ISO/IEC 18033-2 [312, 584].

2.1.2 Outline of the primitive

The public key is an RSA modulus $n = pq$ with an intractable factorisation and a public exponent $e \geq 3$. The secret key is $1/e \bmod \mathrm{LCM}(p-1)(q-1)$ which can be easily pre-computed given p and q. The only parameter is $KeyLen$, the length of the symmetric key that the KEM is required to produce. The scheme also uses a key derivation function which maps input octet strings of arbitrary lengths to octet strings of length $KeyLen$ (see Sect 3).

The encapsulation algorithm takes as input a random seed r and the public key. It computes a symmetric key by applying the key derivation function to r. It computes the encapsulation by computing $C = r^e \bmod n$.

The decapsulation algorithm takes as input an encapsulation C and the secret key. It computes the key by applying the key derivation function to $C^{1/e} \bmod n$.

2.1.3 Security and performance

There is a security proof of RSA-KEM that uses a distinguishing attacker in the adaptive chosen ciphertext model (IND-CCA2 model) to solve the RSA problem (or the e-th root problem depending on the key generation mechanism) [193, 195, 584]. The proof uses the random oracle model and has a tight reduction; indeed the reduction appears to be close to optimal for KEMs derived from one-way trapdoor permutations. This "proof" is a strong security argument. It suggests that, given an encapsulation of a key, the best technique to distinguish between the real key and a randomly generated key is to compute the e-th root of the encapsulation.

The RSA encryption algorithm exhibits some homomorphic properties, most noticeably that if $C_1 = m_1^e \bmod n$ and $C_2 = m_2^e \bmod n$ then $C_1 C_2 \bmod n$ is the encryption of $m_1 m_2 \bmod n$. Whilst this is highly desirable in certain applications, it does allow unknown messages to be manipulated in quite specific ways. In RSA-KEM we rely on the nature of the key derivation function to destroy any relations between keys that might result from relations in the encapsulations. Hence the use of a good key derivation function is critical.

There is also an attack against the RSA cryptosystem that can be applied to RSA-KEM. It has been shown that if the secret exponent d is less than $n^{0.292}$ then it can be recovered from the modulus n and public exponent e [106]. It has been conjectured that it will be possible to recover d from n and e providing $d < n^{0.5}$.

Side channel attacks are a threat against RSA-KEM. In particular, the exponentiation operation is vulnerable to simple power analysis, the key derivation function might be vulnerable to a Hamming weight attack and the scheme is vulnerable to fault/chosen modulus attacks. Countermeasures exist for all known types of attack against RSA-KEM.

The performance characteristics of RSA-KEM are good. Both the encapsulation and decapsulation algorithms only require one modular exponentiation. Performance can be further improved by choosing an advantageous encryption exponent such as $e = 65537$ or, if fast encryption is a priority, $e = 3$. No such advantageous decryption exponent can be chosen. However for adequate security, it will be necessary to choose a large modulus, and so the size of both the public and secret key will be larger than in many elliptic curve schemes.

2.2 Description

2.2.1 Parameter generation

First, a security parameter should be chosen. For a security parameter k, the length of the modulus (in octets) should be greater than or equal to the value given in the following table.

k	72	80	112	128
Modulus length	128	192	512	750

The length of the symmetric key produced by the KEM, $KeyLen$, is defined by the requirements of the DEM.

2.2.2 Key generation

There are two methods to generate a valid key pair for RSA-KEM. Both take as input a security parameter 1^k and a random string of some fixed length r. Let

$l(k)$ be the modulus length corresponding to the security parameter k. The first method works as follows.

Description:

Input: A security parameter 1^k
A fixed length random string r
Output: An RSA-KEM public key pk
An RSA-KEM secret key sk

1. Generate a prime p of length $\lceil l(k)/2 \rceil$ using the $PrimeGen(r, l(k))$ routine.
2. Set $r := NextRand(r)$.
3. Generate a prime $q \neq p$ of length $\lceil l(k)/2 \rceil$ using the $PrimeGen(r, l(k))$ routine.
4. Set $r := NextRand(r)$.
5. Set $n := pq$.
6. Check that n has length $l(k)$. If not, goto step 1.
7. Generate an odd public exponent e in the range $[3, \mathrm{LCM}(p-1, q-1))$ using the random seed r in such a way that every possible value of e will be generated with approximately equal probability.
8. Set $r := NextRand(r)$.
9. Check that $\mathrm{GCD}(e, (p-1)(q-1)) = 1$. If not, goto step 7.
10. Set $d \equiv e^{-1} \bmod (p-1)(q-1)$.
11. Check that $d > n^{0.5}$. If not, goto step 7.
12. Set $pk := (n, e)$.
13. Set $sk := (n, d)$.
14. Output the public key pk and the secret key sk.

The second method works as follows.

Description:

Input: A security parameter 1^k
A fixed length random string r
Output: An RSA-KEM public key pk
An RSA-KEM secret key sk

1. Select a public exponent e, an odd integer greater than two.
2. Generate a prime p of length $\lceil l(k)/2 \rceil$ using the $PrimeGen(r, l(k))$ routine.
3. Set $r := NextRand(r)$.
4. Check that $\mathrm{GCD}(e, p-1) = 1$. If not, goto step 2.
5. Generate a prime $q \neq p$ of length $\lceil l(k)/2 \rceil$ using the $PrimeGen(r, l(k))$ routine.
6. Set $r := NextRand(r)$.
7. Check that $\mathrm{GCD}(e, q-1) = 1$. If not, goto step 2.
8. Set $n := pq$.
9. Check that n has length $l(k)$. If not, goto step 2.
10. Check that $e < \mathrm{LCM}(p-1, q-1)$. If not, goto step 2.

11. Set $d \equiv e^{-1} \bmod (p-1)(q-1)$.
12. Check that $d > n^{0.5}$. If not, goto step 2.
13. Set $pk := (n, e)$.
14. Set $sk := (n, d)$.
15. Output the public key pk and the secret key sk.

Care must be taken when implementing the second method that, for a given security parameter, e is small enough so that primes p and q can be found such that $e < \mathrm{LCM}(p-1, q-1)$.

2.2.3 Encapsulation algorithm

To generate a symmetric key of length $KeyLen$ and an encapsulation of that key the following algorithm is executed.

Description:

Input: An RSA-KEM public key $pk = (n, e)$
A fixed length random string r
The size of required symmetric key $KeyLen$
Output: A symmetric key K of length $KeyLen$
An encapsulation C

1. Validate the public key and the system parameters.
2. Set Len to be the length of n in octets.
3. Generate a random integer m in the range $[0, n)$ using the random seed r in such a way that every possible value for m will be generated with approximately equal probability.
4. Set $M := \mathsf{I2OSP}(m, Len)$.
5. Set $K := KDF(M, KeyLen)$.
6. Set $C_{raw} := m^e \bmod n$.
7. Set $C := \mathsf{I2OSP}(C_{raw}, Len)$
8. Output the symmetric key K and the encapsulation C.

2.2.4 Decapsulation algorithm

To decapsulate an encapsulated key of length $KeyLen$ the following algorithm is executed.

Description:

Input: An RSA-KEM private key $sk = (n, d)$
An encapsulated key C
The size of required symmetric key $KeyLen$
Output: A symmetric key K of length $KeyLen$

1. Validate the private key and the system parameters.
2. Set Len to be the length of n in octets.
3. Check that C is an octet string of length Len. If not, output "error" and abort.

4. Set $C_{raw} = \mathsf{OS2IP}(C)$.
5. Check that C_{raw} is an integer in the range $[0, n)$. If not, output "error" and abort.
6. Set $m := C^d \bmod n$.
7. Set $M := \mathsf{I2OSP}(m, Len)$.
8. Set $K = KDF(M, KeyLen)$.
9. Output K.

2.2.5 Guidelines for implementation

It is important to validate all inputs to the encapsulation and decapsulation algorithms before they are used. In the case of key material and system parameters, this prevents chosen modulus attacks and fault attacks. In the case of RSA-KEM it is usual to use some certified trust mechanism so the algorithm can trust that the system parameters and the keys are valid.

2.3 Test vectors

The following test vectors were generated by RSA-KEM with a security parameter $k = 80$. The algorithms use the key derivation function $Mech1$ (see Sect. 3) with the hash function SHA-1. The algorithm was asked to produce keys of length $KeyLen = 16$ octets. The test vectors were generated using a public key

```
n  =  9923916CF5589CE08EB945D635AE4534 2D443EC2E7D46980CCF48DC877ADA2C1
      0F92A33D77D8AFB83DEA73C48CB5D42F 5A9D34F10A004C6904EEFFAEFF1DB64B
      6DD770283F4F9B67370F570353DE0DFD 392CF6AA35FCE915D0F45B8087D90CBD
      CE2C5C1205680ED36E69D5FC1C46D5C2 7BFA5BF0AD8D2C94C454EF33F21254EE
      2704C031EEE0F19282D6F104A566B434 A5B562F24308FC2598BFBC9CEB7277FA
      6149CD20C217A8AF9CEC19791C3D5DDA 4721877675F3ACB8B80E4012EA3622B1
e  =  00000000000000000000000000000000 00000000000000000000000000000000
      00000000000000000000000000000000 00000000000000000000000000000000
      00000000000000000000000000000000 00000000000000000000000000000000
      00000000000000000000000000000000 00000000000000000000000000000000
      00000000000000000000000000000000 00000000000000000000000000000000
      00000000000000000000000000000000 00000000000000000000000000010001
```

and a private key

```
n  =  9923916CF5589CE08EB945D635AE4534 2D443EC2E7D46980CCF48DC877ADA2C1
      0F92A33D77D8AFB83DEA73C48CB5D42F 5A9D34F10A004C6904EEFFAEFF1DB64B
      6DD770283F4F9B67370F570353DE0DFD 392CF6AA35FCE915D0F45B8087D90CBD
      CE2C5C1205680ED36E69D5FC1C46D5C2 7BFA5BF0AD8D2C94C454EF33F21254EE
      2704C031EEE0F19282D6F104A566B434 A5B562F24308FC2598BFBC9CEB7277FA
      6149CD20C217A8AF9CEC19791C3D5DDA 4721877675F3ACB8B80E4012EA3622B1
d  =  870401F293B9C5CE82673CF868A9B660 134CE91CC472D575F6BDE2C78D24ACAB
      1484CFA1A1298D7B9E3338506152EAB9 B9658348C4ED9070C325C88DCC65B0D4
      7E0A84DB273E93A003BE65940C7C69CF 097AE81B17B05CFC9C16E519C42C0C7B
      6A01AF12FF709FD952489E38ADF57776 1CA0B49A7E0CD71330C641B5CD3F3E3E
      728E489506F545158535E1011B8F85F2 D4A0285594D6530A18FB0D1895662A22
      F7D60506962412FA2EC5E0F3CA0CA6DC 2B139053E0A15E53D95A337A7F3AF731
```

The random integer m in the encapsulation algorithm is derived from the fixed length octet string r by setting $m = \mathsf{OS2IP}(r)$.

2.3.1 Test vectors for key encapsulation

The following are the correct results of the key encapsulation algorithm being run with a seed r, producing a symmetric key K and a encapsulated key C.

```
r  =  00000000000000000000000000000000 00000000000000000000000000000000
      00000000000000000000000000000000 00000000000000000000000000000000
      00000000000000000000000000000000 00000000000000000000000000000000
      00000000000000000000000000000000 00000000000000000000000000000000
      00003031323334353637383961626364 65666768696A6B6C6D6E6F7071727374
      75767778797A4142434445464748494A 4B4C4D4E4F505152535455565758595A
K  =  A2926ABCAF5AE7AC2ECDA7FEF959EE1B
C  =  7F6A4A769F055D72E01B09CB6F8726CD 4D9EFEE126CC70194E7EC4B2896548C3
      AD3A3A26C2006A80E8B93027480C885B A49494E318DD200DA5CEA006A0B384DE
      F0F9EDFC98553BB7D917AA7B8EE355B5 29854E6808D37125D32F124A2D68DEF6
      693E6308C7ED8EFA5B491A7CDA9BDD20 6B155527789203DDF829C4A6F0BE50B6
      1436C4D63C785A068BECBC9008D4EB5D FCCF9DC3DB281F2B30A5C3DAD645F058
      18687B0BB115302D5E1E0B2BC4F6A416 D9CCA5E98C17D240E4EF08603F584A25
```

2.3.2 Test vectors for key decapsulation

The following key encapsulation C correctly decapsulates to give the symmetric key K.

```
C  =  07D4816C8406394DB9FB6884FC802A83 C20968C1502167FC19AFED75C15187F6
      80A2E2A3E1E18D04645044F4EDFFFB55 C7F252E400B491FDA5C06FBBE9FF5EE9
      AEC472208881A0D11EC4739FC0409008 D39DAD6DA9D35F9880D7F570459CDFCF
      779377988216C14309152201CB6A3620 C1FB7AAF806AF44D8C51449545F2E01A
      C5B8AE1063354629C212EC357B5EE2F2 777FD058B1F48E560E1643982B4B1BB7
      DD5131AE621BA102A6A84DBA7CE1108D 3F32C2D9DFDFED02F2126E9B82638B0C
K  =  D65B0E88B014E0C22F6F66CE453AE0E9
```

The following key encapsulation C fails to decapsulate owing to step 3 of the decapsulation algorithm (the key encapsulation is not a correct size).

```
C  =  11111111111111111111111111111111 11111111111111111111111111111111
```

The following key encapsulation C fails to decapsulate owing to step 5 of decapsulation algorithm (the key encapsulation represents an integer bigger than the modulus).

```
C  =  9923916CF5589CE08EB945D635AE4534 2D443EC2E7D46980CCF48DC877ADA2C1
      0F92A33D77D8AFB83DEA73C48CB5D42F 5A9D34F10A004C6904EEFFAEFF1DB64B
      6DD770283F4F9B67370F570353DE0DFD 392CF6AA35FCE915D0F45B8087D90CBD
      CE2C5C1205680ED36E69D5FC1C46D5C2 7BFA5BF0AD8D2C94C454EF33F21254EE
      2704C031EEE0F19282D6F104A566B434 A5B562F24308FC2598BFBC9CEB7277FA
      6149CD20C217A8AF9CEC19791C3D5DDA 4721877675F3ACB8B80E4012EA3622B2
```

3. ACE-KEM

3.1 Introduction

3.1.1 Overview

ACE-KEM is a key encapsulation mechanism based on the security of the Diffie-Hellman key agreement protocol in certain groups, in particular in the multiplicative group of integers modulo a prime and in a prime order subgroup of an elliptic curve group. The Diffie-Hellman key agreement protocol [201] was developed by Whitfield Diffie and Martin Hellman in 1976, and was the first example of an asymmetric cryptosystem. Since then it has been the basis for many popular and efficient cryptosystems used today. The use of elliptic curve groups in cryptography was first suggested simultaneously by Koblitz [373] and Miller [446].

ACE-KEM was submitted to the NESSIE project by IBM's Zurich Research Laboratory [563]. It is also included in the draft ISO/IEC standard on asymmetric encryption, ISO/IEC 18033-2 [312, 584].

3.1.2 Outline of the primitive

The action of ACE-KEM takes place in a prime order cyclic subgroup of some large group. This larger group could be either the multiplicative group of integers modulo a prime or an elliptic curve group. We will assume that the group action of this group is written additively and that the prime order cyclic subgroup is generated by an element P.

The public-private key pair of the scheme consists of four Diffie-Hellman keys, i.e. the public key is the set of group elements P, $W = wP$, $X = xP$, $Y = yP$ and $Z = zP$, and the corresponding private key is the set of integers w, x, y and z.

ACE-KEM produces symmetric keys of some pre-specified length $KeyLen$. The scheme makes use of a key derivation function $KDF(\cdot)$, which maps octet strings of an arbitrary length to octet strings of length $KeyLen$ (see Sect. 3). The scheme also uses a hash function $Hash(\cdot)$, which maps octet strings of an arbitrary length to octet strings of a fixed length. This fixed length should be smaller than the length of the order of the subgroup generated by P.

The encapsulation algorithm takes a fixed length random seed r and the public key as input. First it computes a symmetric key K and an encapsulation of that key C_1 using a technique similar to ECIES-KEM, in other words it sets $C_1 = rP$,

$Q = rZ$ and $K = KDF(C_1||Q)$. Next it computes a second encapsulation $C_2 = rW$, a hash value $\alpha = Hash(C_1||C_2)$ and a final encapsulation $C_3 = rX + \alpha rY$.

The decapsulation algorithm takes as input the three parts of the encapsulated key C_1, C_2 and C_3, and the private key. It computes the hash value $\alpha = hash(C_1||C_2)$ and the integer $t = x + y\alpha$. It now checks that the encapsulation is valid by checking that $C_2 = wC_1$ and $C_3 = tC_1$. If this check is successful then the algorithm computes $Q = zC_1$, $K = KDF(C_1||Q)$ and outputs the symmetric key K.

3.1.3 Security and performance

There are several security proofs for ACE-KEM and it is the only practical scheme that has a proof which does not rely on the random oracle model.

The first security proof uses a distinguishing attacker in the adaptive chosen ciphertext model (IND-CCA2 model) to solve the decisional Diffie-Hellman problem [171]. This proof does not use the random oracle model and has a tight reduction. Hence it is a very strong security argument. It proves that for an attack to be successful the scheme must either use a weak hash function, use a weak key derivation function or be built on a curve for which the decisional Diffie-Hellman problem is easily solved.

The second security proof uses a distinguishing attacker in the adaptive chosen ciphertext (IND-CCA2 model) to solve the gap Diffie-Hellman problem [171]. This proof uses the random oracle model and has a tight reduction. This proof is a strong security argument. It suggests that, given an encapsulation of a key, the best technique to distinguish between the real key and a randomly generated key is to solve the instance of the Diffie-Hellman protocol used in the encapsulation. However this proof assumes that the scheme is built on a group for which the decisional Diffie-Hellman problem is easily solved.

The precise nature of ACE-KEM's vulnerabilities to side channel attacks depends upon the group used to implement the scheme. However, whether the scheme is implemented on the multiplicative group of integers modulo a prime or on an elliptic curve group, the scheme is vulnerable to both simple power analysis and to fault attacks. However it does not appear to be any more vulnerable to these attacks than any other Diffie-Hellman based scheme implemented on that group, and countermeasures exist for all of these attacks. It is unclear to what extent the attacks of Lim and Lee [399], Biehl *et al.* [58] and Antipa *et al.* [23] are effective against ACE-KEM; however it is recommended that all key and ciphertext group elements are validated before use.

The performance characteristics for ACE-KEM are not good however. Hence ACE-KEM is only recommended for situations where security is more important than speed. For even greater security several asymmetric schemes could be used in sequence. Such a sequence should include both ACE-KEM and RSA-KEM (with suitably independent and randomly generated keys).

3.2 Description

3.2.1 Parameter generation

First, the group on which the scheme is going to be based should be chosen. This group should either be a prime order subgroup G of the group H of multiplicative integers modulo a prime, or a prime order subgroup G of an elliptic curve group H. The size of the subgroup G, and possibly the group H, may well depend upon the size of the security parameter k. The parameters that need to be chosen for each type of group will be dealt with separately.

It is also necessary to be able to encode elements of the group as octet strings. For this purpose we define two functions: GE2OSP which encodes a group element as an octet string, and OS2GEP which decodes an octet string as a group element. The particular form of these functions will be detailed separately for each type of group.

3.2.1.1 Parameter generation for modular arithmetic groups

In the modular arithmetic setting, ACE-KEM is a scheme that acts in a prime order, cyclic subgroup G of the multiplicative group of integers modulo a prime q. Hence H contains $q-1$ elements. Let G have order p. For a security parameter k, p and q should be chosen to have length (in octets) greater than or equal to the value given in the following table.

k	72	80	112	128
Length of q	128	192	512	750
Length of p	18	20	28	32

It is necessary for the hash function to produce an output whose length is less than the length of p. Also, in order to ensure that the group H has a subgroup G of the required size, it is necessary (and sufficient) for p to divide $q-1$.

A generator $P \in H$ for G also needs to be selected. The simplest way to do this is to select an element $P' \in H$ uniformly at random and set $P = \alpha P'$ where $\alpha = (q-1)/p$. If $P \neq 1$ then P is a generator for a subgroup of order p.

Let the length of q be $qLen$. The encoding function, GE2OSP(X), will be I2OSP$(X, qLen)$ and the decoding function, OS2GEP(X), will be OS2IP(X). The group action will be given by multiplication modulo q, i.e. to "add" two elements of the group we multiply the two integers together and take the result modulo q.

The length of the symmetric key produced by the KEM, $KeyLen$, is defined by the requirements of the KEM.

3.2.1.2 Parameter generation for elliptic curve groups

For a security parameter k, the order of the cyclic subgroup G should be a prime p whose length (in octets) is greater than or equal to the value given in the following table.

k	72	80	112	128
Length of p	18	20	28	32

The scheme should be defined over a finite field $\mathbb{F}_q$ of characteristic 2 or over a prime field. An elliptic curve $E(\mathbb{F}_q)$ should be chosen on which either

- the decisional Diffie-Hellman problem is intractable, or
- the computational Diffie-Hellman problem is intractable there exists an algorithm that correctly solves the decisional Diffie-Hellman problem.

A point P on $E(\mathbb{F}_q)$ needs to be selected where P has order p and the length of p corresponds to the value defined in the table above for the security parameter k.

The encoding function, $\mathsf{GE2OSP}(X)$, will be $\mathsf{ECP2OSP}(X)$ and the decoding function, $\mathsf{OS2GEP}(X)$, will be $\mathsf{OS2ECPP}(X)$. The action of the group will be given by addition of elliptic curve points, i.e. to 'add" two group elements one merely adds the two points on the elliptic curve.

The length of the symmetric key produced by the KEM, $KeyLen$, is defined by the requirements of the DEM.

3.2.2 Key generation

The key generation algorithm for ACE-KEM is as follows.

Description:

Input: A security parameter 1^k
A fixed length random string r
A group H
A cyclic, prime order subgroup G of H
A group element P that generates G
The order of P, p

Output: An ACE-KEM public key pk
An ACE-KEM secret key sk

1. Verify p is a prime. If not, output "error" and abort.
2. Verify the fact that P is an element of the group G and that P has order p. If not, output "error" and abort.
3. Generate an integer w in the range $[2, p-1]$ using the random seed r in such a way that every possible value of w will be generated with approximately equal probability.
4. Set $r := NextRand(r)$.
5. Generate an integer x in the range $[2, p-1]$ using the random seed r in such a way that every possible value of x will be generated with approximately equal probability.
6. Set $r := NextRand(r)$.
7. Generate an integer y in the range $[2, p-1]$ using the random seed r in such a way that every possible value of y will be generated with approximately equal probability.
8. Set $r := NextRand(r)$.

9. Generate an integer z in the range $[2, p-1]$ using the random seed r in such a way that every possible value of z will be generated with approximately equal probability.
10. Set $W := wP$.
11. Set $X := xP$.
12. Set $Y := yP$.
13. Set $Z := zP$.
14. Set $pk := (W, X, Y, Z)$.
15. Set $sk := (w, x, y, z)$.
16. Output the public key pk and the secret key sk.

3.2.3 Encapsulation algorithm

To generate a symmetric key of length $KeyLen$ and an encapsulation of that key the following algorithm is executed.

Description:

Input: An ACE-KEM public key $pk = (W, X, Y, Z)$
A fixed length random string r
A group H
A cyclic, prime order subgroup G of H
A group element P that generates G
The order of P, p
The size of required symmetric key $KeyLen$

Output: A symmetric key K of length $KeyLen$
An encapsulation C

1. Validate the public key and the system parameters.
2. Generate a random integer s in the range $[0, p)$ using the random seed r in such a way that every possible value for s will be generated with approximately equal probability.
3. Set $Q := \mathsf{GE2OSP}(sZ)$.
4. Set $C_1 := \mathsf{GE2OSP}(sP)$.
5. Set $K := KDF(C_1 || Q, KeyLen)$.
6. Set $C_2 := \mathsf{HGE2OSP}(sW)$.
7. Set $\alpha := \mathsf{OS2IP}(Hash(C_1 || C_2))$.
8. Set $C_3 := \mathsf{GE2OSP}(sX + \alpha sY)$.
9. Set $C := (C_1, C_2, C_3)$.
10. Output the symmetric key K and the encapsulation C.

3.2.4 Decapsulation algorithm

To decapsulate an encapsulated key of length $KeyLen$ the following algorithm is executed.

Description:

Input: An ACE-KEM private key $pk = (w, x, y, z)$
An encapsulated key C
A group H
A cyclic, prime order subgroup G of H
A group element P that generates G
The order of P, p
The size of required symmetric key $KeyLen$
Output: A symmetric key K of length $KeyLen$

1. Validate the private key and the system parameters.
2. Parse C as (C_1, C_2, C_3).
3. Set $P_1 := \mathsf{OS2GEP}(C_1)$.
4. Verify that P_1 is an element of the group H and that P_1 lies in the subgroup generated by P. If not, output "error" and abort.
5. Set $P_2 := \mathsf{OS2GEP}(C_2)$.
6. Verify that P_2 is an element of the group H and that P_2 lies in the subgroup generated by P. If not, output "error" and abort.
7. Set $P_3 := \mathsf{OS2GEP}(C_3)$.
8. Verify that P_3 is an element of the group H and that P_3 lies in the subgroup generated by P. If not, output "error" and abort.
9. Set $\alpha := \mathsf{OS2IP}(Hash(C_1 || C_2))$.
10. Set $t := x + \alpha y \bmod p$.
11. Check that $P_2 = wP_1$. If not, output "error" and abort.
12. Check that $P_3 = tP_1$. If not, output "error" and abort.
13. Set $Q := \mathsf{GE2OSP}(zP_1)$.
14. Set $K = KDF(C_1 || Q, KeyLen)$.
15. Output K.

3.2.5 Guidelines for implementation

It is important to validate all inputs to the encapsulation and decapsulation algorithms before they are used. In the case of key material and system parameters, this prevents chosen modulus attacks and fault attacks.

In particular it is necessary to trust the properties of the group elements. So, for both the encapsulation and decapsulation algorithms it is necessary to trust that

– p is a prime number whose length is as defined by the security parameter and,
– P is an element of the group H and P has order p.

If these conditions do not hold then the algorithm should output "error" and abort. The two obvious methods to validate the system parameters are to check the properties hold during the execution of the algorithm (which may be computationally expensive), or to have some kind of certified trust mechanism in place so the algorithm can trust that the system parameters have been correctly validated by some third party.

It is also necessary, in the encapsulation algorithm, to check the properties of the group elements W, X, Y and Z in the public key. In particular it is necessary to check that each of these elements is a member of the group H and that they lie in the subgroup generated by P.

In the modular arithmetic case both of these checks are easy. An integer Q is an element of H if and only if it lies in the range $[1, q-1]$ and it is in the subgroup generated by P if and only if $pQ = 1 \mod q$.

It is slightly harder to perform the check in the elliptic curve case. A point $Q \in \mathbb{F}_q \times \mathbb{F}_q$ is an element of H if and only if its co-ordinates satisfy the equation defining the elliptic curve. More information about the structure of the elliptic curve group H is needed in order to check that an element $Q \in H$ is in the subgroup generated by P.

In either case, care should also be taken that, should the decapsulation algorithm fail (i.e. output "error" and abort), no information in any form should be returned to the users as to the cause of the failure. This includes information that may be obtainable from side-channels (such as timing information or power consumption information). Most notably the user should not be able to tell if the decapsulation algorithm fails to decapsulate a key because of a failed check in step 11 or in step 12 of the algorithm.

3.3 Test vectors

The following test vectors were produced using the modular arithmetic version of ACE-KEM with a security parameter $k = 80$. This should in no way be construed as a recommendation of that version of ACE-KEM over the elliptic curve version. The algorithms use the hash function SHA-1 and the key derivation function $Mech2$ (see Sect. 3) with the hash function SHA-1. The algorithm was asked to produce keys of length $KeyLen = 16$ octets.

The test vectors were generated using modular arithmetic over a finite field $\mathbb{F}_q$ where q has length 192 (octets) and a subgroup of prime order p where p has length 21 (octets). The subgroup was generated by an element P. The parameters p, q and P have the following values.

```
q  =  F63D79988776665293AFD497B15FCC77 1F52D012676993E8BFDF57A99BD50564
      EAB8752416C4D56F335F91E5F6C6848C 6C03F1A96CDBD112A802BA6C2618B962
      01BAC254C2069FAF232ADE367FBF4B00 8A2CD76D8E0E7FA287DDE8A91A906ABF
      367F47BA07E0AB95A7044A63330A8282 C55FC1AA862AB8E38AB195A68FAC6CB1
      43DA3831680AD753622021B7582EDEA7 691C866F33FEDBB982AFF50993877E6F
      8FA8AC6F41234EE51C642567EFAF2D12 5D4A99C383B932E8355AC7B276796B99

p  =  91BA928B778960257687 2652D07BAE14D863CF8D5B

P  =  9A1C3BFAEE3570DB8FF99B324D965DA0 97990C7F1458A27729E71B7A2D6917E3
      B68AE4908BF3F588DECD34E8848DE578 17EA566D3AD130A4AF63B69F9A8E920F
      5E8182DE1B439615447D3CFB29BC4780 D93DFDA7558F4988B70BE22125528D8A
      14336CD94FE5D52DFC21B420B6A492A8 3F46285B4647059B0241A216A7D22196
      047966A99BB9C18216CE871970154CCA DB571B046F9195D8E3FC28C0259AE90F
      2140B3A1FB0FF90B88F63C8FA38A9421 B058399632F00E6640000F7ACB63198C
```

The test vectors were produced using a public key (W, X, Y, Z) where W, X, Y and Z are the following integers.

$$\begin{array}{rcl}
W & = & \texttt{507AB6F4D38DC8F36B60F975D1EAF191 046549D0D9336A61CA3DB2731279EF6A} \\
 & & \texttt{99DC4854B572B0BD095F6A3A74F4A3C8 8DD996792B99B7ECAD13F28B2C08B584} \\
 & & \texttt{2E6CA574F6B7A3FFE5E2BBF9541B2098 103AC75D82010FB79228CB71F2BB0E8C} \\
 & & \texttt{829BBECED204F1B9E9EE85A8AF55030E E9AA548A2A8E570A50FFC6AFED719295} \\
 & & \texttt{5A2D78A3A98EAFDC0FD65A1D71B4F2E8 9E98FF39F428DBBD866714D630CE2BCA} \\
 & & \texttt{1D1F93645824AB29DF1DA7D3D45AB29F B54C2F5BD8BEC366A5C74F5408B07094} \\
X & = & \texttt{9B1A7E9A5B8A14C4B6D2F3FA92A69CA1 BEED7F9F3F50DC37ED9FD38FB09C097F} \\
 & & \texttt{7F693523826B90B4951CC165F71EB884 43F9E9DDC7A840876A8BCE9D46615C0C} \\
 & & \texttt{42604F83B01980CDC8BDB8A749B6B223 A1F3E433D9D160BCF304A37094D8D935} \\
 & & \texttt{7FBE8B9384794E7D3206BBC2F915B186 B1BCED307CB398E4440AE8C301FB4C5F} \\
 & & \texttt{4F79EA9BDDCE3959F86FBC904DC229A2 DA0F93B0EF07C97AE383C19430C671D4} \\
 & & \texttt{4C131A70429FF917376459A49C0967AD 9A039B12C23E62B89E7D1762197F2EF2} \\
Y & = & \texttt{0FEA8BCBC351B50817BA75CAB58F6333 9D3C884F775C2D3359774AB87DEC6B93} \\
 & & \texttt{6B1EF29737790F7E08507C93EA3A18D3 C4E15F160E44C708D1F837720ED0A12A} \\
 & & \texttt{0716164E26744A5743B37DB885B3943B 71D1FCB1331905DCAF172FB81D0163B9} \\
 & & \texttt{28BEC52A1D7E8DE4995782B15A675521 6EE4D60925B8CE8BEAC8EE90C2AC6C9F} \\
 & & \texttt{018CFA8514B574CC5D671A7866811F71 E5DCB66ED6476B566395CB8C0AAAD9AD} \\
 & & \texttt{981DAB6701889D22E3CD69036796EEB6 04C5C966839FCBADDAE7C027B912E744} \\
Z & = & \texttt{EB5C4FDB37DAE7EA0B07024831F281AE B97771BCB8FEF8408FE0295DE843883E} \\
 & & \texttt{913ECEDDC63285D728999C3FDF6C8818 C26B82763168202DB917C66B310B8D15} \\
 & & \texttt{886F14D402352FAE87E2691FC308B5D2 0FA3C4C4BE0F70F227D9C4D183EA4F2D} \\
 & & \texttt{58A990FCE0E7340BCA261B46B3ADA468 F0212E06947029F0F04DF266C0E1FC44} \\
 & & \texttt{9E37401987284DC0B15471D606177522 CF342CD1B960F3CA72597B8F13F22917} \\
 & & \texttt{3C26062D51DF7552EB4680B7E63C0BEC 53B1C26C19BEC973983358B672A314CB}
\end{array}$$

The corresponding private key is (w, x, y, z) where w, x, y, z are the following integers.

$$\begin{array}{rcl}
w & = & \texttt{854B4B8962F278651290D25F0616AB83C3F885B6E3} \\
x & = & \texttt{72A5AE967910CDE05D287AF608514012D5E002101F} \\
y & = & \texttt{2715CE067E9608FDA9A0A1B6FC51085A2867C89448} \\
z & = & \texttt{41816B06DADE4B10D703F14853F039C1E83AF3FE7D}
\end{array}$$

The random integer s in the encapsulation algorithm is derived from the fixed length octet string r by setting $s = \mathsf{OS2IP}(r)$.

3.3.1 Test vectors for key encapsulation

The following are the correct results of the key encapsulation algorithm being run with a seed r, producing a symmetric key K and an encapsulated key $C = (C_1, C_2, C_3)$.

r	$=$	303132333435363738396162636465666768696A6B	
K	$=$	798B355BEF635E9B0E9055C5C3B99D2F	
C_1	$=$	3764D7B60FA7667E7E61EAFCBB03FB13	E3FC22BD0CBFBF22B31FA708CB4CBB89
		936D3F455EBBC79093522383CB606707	E0B7CA1F6826F3442C912C1044858611
		8FA6F885309F94CD3D75FE2136525785	F378B6D5E59E237B70BD38E15FC4F45F
		E7C0CAC4A1F7EB77BFB8555AC556291C	84E76FC8DAA0B8940ECFCA2B75AF82EF
		75F7B12529700F00A260B18F81D444DA	0BCB2263D50A13B4986068B4DAF43771
		D34829DC5B3A75B0A0DC717C2A2B2746	88DA41DA6D1DEE08580B4A82B0380F2E
C_2	$=$	F2CDC24EE70CFB13D269D13A6611BA87	65D167B49C950EFEBA1F65092BCFA4C1
		8672A0A06511D512FD8E99F6931D0BEA	00407393B5C963F14E5DD8C7356F8280
		861E56FBFDBBB73DD63974F6DE18F196	B1D9F820759E7858CC5F1EE7556C9B38
		E6359F8F1577916A75BBA9C0A992CBAC	1F0572C3414204C90F7115920131684E
		78AEE0A785FEF9BAC0D52A87F5C5E92A	8C7CEDB7AE6AB03CDF29FABA316955E1
		3A9BE8A9BBEB9AFE904F5DEF94CD42CB	EC0425638E536A6D882B30F8732CBED4
C_3	$=$	77602C9E02C0E9F0535374AD20E9135D	D4978C5C5D981F12DD64E7CC7AC9914C
		BA566BA94479A6A614B7EF060B5E2F4F	077D5A55534E5C496052016EE0441D41
		84F01D59D33D2633BCB2217961 1C250D	97842111D5D19B7B0117E3ABD9928FAD
		518959310913ADFBA3299F1F5C16E223	308758D2E78707A45794A15BA596A9F1
		A9BBEF3FCA87E0F4BD66CE15BFF09826	3900C7A1A133230F93B301ED29557025
		160CDC7178C31BA1051853F97051734B	C9F18278A0639B989DB40790D427BFEF

3.3.2 Test vectors for decapsulation

The following key encapsulation $c = (C_1, C_2, C_3)$ correctly decapsulates to give the symmetric key K.

C_1	$=$	40F09FE0D10B290183336B0E67FD2811	424E13A1E3624C52BB6A0C1AB8C60BE3
		C312F528392F10B0C2383C78D3923A38	E26B2A70905B42F4B676DED2C2F98DC2
		7BF94C60B1457F5780F32379C18A2741	FD1D32285BDA06F11F88FEB4D3A9459D
		97CDA8DE2404616FA8F7312C2E7B3B46	7D8D2F3506F9CD0C1BEC2B87E49352ED
		C7BE5114CBA10FC50294E309C352E7A3	72DA66D0EDCED06A059EFC506B04EA6B
		A03DDECCC560E1A7D007533A8940A223	41965EE7BCF46DE89CBD65E4D3BA1BFE
C_2	$=$	9E06428DF618F0595FDEDA33F4E36921	82A06BFED2CACDD5FFD8007098F6B722
		40839A3B50F9985A3EEB6FC655F2AACE	01B59783E04B096B994EB628C35EAE99
		DBF9D0089D47500384C31E68D2DF7DF6	1EE557E6B6AB53DD8F5901FA5FD05077
		F1B7EFB0389D949A501C597855BCC70D	93B1AF007E3B7607715ACA507C82B250
		F3DBD97A9FFD73615EFA6CC8258A483C	143234C5460AD8D1B52F74BD5F66D4BF
		D396B629BB2920CE7FB481F1BA3FECE6	0B7FEB05B2D37FACFFBAB261EC6A904A
C_3	$=$	1477B83628AAF3989E3284B3EDE68307	614C5654454551C7186F5FF556797589
		F8878D4C89C563EACFD8F979C2764D62	2FA2C1058E400D71A00F03646FE3659E
		9657EF71728816946CF0AED33A0DE953	87D78D1C86FA78D42C7DD61F4983C021
		B3DFC63BCDFA70BAB4B97228597E81F6	1DA8A2C9A22CD6034BBA7F8D5B4F7DDE
		FCFBF3EEBD9C4AC5893F61593A19FE78	15F67F82327C49185C45C0D5116B374B
		F17DEA58BA5D954C046230B035DBBBF7	1BE731A50440D5639875EC335A22AE5F
K	$=$	480C18969F0D38100178221C069A555A	

The following key encapsulation $C = (C_1, C_2, C_3)$ fails to decapsulate owing to step 11 of the decapsulation algorithm (the integrity check on P_1 and P_2).

C_1 = 40F09FE0D10B290183336B0E67FD2811 424E13A1E3624C52BB6A0C1AB8C60BE3
C312F528392F10B0C2383C78D3923A38 E26B2A70905B42F4B676DED2C2F98DC2
7BF94C60B1457F5780F32379C18A2741 FD1D32285BDA06F11F88FEB4D3A9459D
97CDA8DE2404616FA8F7312C2E7B3B46 7D8D2F3506F9CD0C1BEC2B87E49352ED
C7BE5114CBA10FC50294E309C352E7A3 72DA66D0EDCED06A059EFC506B04EA6B
A03DDECCC560E1A7D007533A8940A223 41965EE7BCF46DE89CBD65E4D3BA1BFE

C_2 = 11111111111111111111111111111111 11111111111111111111111111111111
11111111111111111111111111111111 11111111111111111111111111111111
11111111111111111111111111111111 11111111111111111111111111111111
11111111111111111111111111111111 11111111111111111111111111111111
11111111111111111111111111111111 11111111111111111111111111111111
11111111111111111111111111111111 11111111111111111111111111111111

C_3 = 227D5E0F060C3394B971174A304ED7F5 D6315A9C809EA8F5878AEE1F125566E2
903CBF17A4DE79AC33B7D95212EAB2DA 73C961705AAD3D61ED9906116B4464C0
7786E58A76F1C4610B49BB2C47665949 E7F6664772FF7F36E5BA049F6E6E4803
1537ABEF6DD3F3A2DC10C97541E9762D D9815D84BC986BDAFED82258931DDDBE
44D161B0F16B0ED004565C4DF2271191 D89332D60198E102B9F3DFB87A013CA0
E1504B5F949D7EEDAD3857FDB8A2784A 7B58718731F647054601D32DBDDE9AE9

The following key encapsulation $C = (C_1, C_2, C_3)$ fails to decapsulate owing to step 12 of the decapsulation algorithm (the integrity check on P_1 and P_3).

C_1 = 40F09FE0D10B290183336B0E67FD2811 424E13A1E3624C52BB6A0C1AB8C60BE3
C312F528392F10B0C2383C78D3923A38 E26B2A70905B42F4B676DED2C2F98DC2
7BF94C60B1457F5780F32379C18A2741 FD1D32285BDA06F11F88FEB4D3A9459D
97CDA8DE2404616FA8F7312C2E7B3B46 7D8D2F3506F9CD0C1BEC2B87E49352ED
C7BE5114CBA10FC50294E309C352E7A3 72DA66D0EDCED06A059EFC506B04EA6B
A03DDECCC560E1A7D007533A8940A223 41965EE7BCF46DE89CBD65E4D3BA1BFE

C_2 = 9E06428DF618F0595FDEDA33F4E36921 82A06BFED2CACDD5FFD8007098F6B722
40839A3B50F9985A3EEB6FC655F2AACE 01B59783E04B096B994EB628C35EAE99
DBF9D0089D47500384C31E68D2DF7DF6 1EE557E6B6AB53DD8F5901FA5FD05077
F1B7EFB0389D949A501C597855BCC70D 93B1AF007E3B7607715ACA507C82B250
F3DBD97A9FFD73615EFA6CC8258A483C 143234C5460AD8D1B52F74BD5F66D4BF
D396B629BB2920CE7FB481F1BA3FECE6 0B7FEB05B2D37FACFFBAB261EC6A904A

C_3 = 11111111111111111111111111111111 11111111111111111111111111111111
11111111111111111111111111111111 11111111111111111111111111111111
11111111111111111111111111111111 11111111111111111111111111111111
11111111111111111111111111111111 11111111111111111111111111111111
11111111111111111111111111111111 11111111111111111111111111111111
11111111111111111111111111111111 11111111111111111111111111111111

Part F

Digital signature schemes

1. RSA-PSS

1.1 Introduction

1.1.1 Overview

The RSA-PSS signature scheme was proposed in 1996 by Mihir Bellare and Phillip Rogaway [54] as an example of digital signature scheme with an efficient proof of security. It is a randomized digital signature scheme with appendix, based on the RSA trapdoor permutation discovered by Ronald L. Rivest, Adi Shamir and Leonard Adleman [543]. The RSA-PSS submitted to NESSIE by RSA Security [326] is slightly different from the original scheme. These modifications were made by Mihir Bellare, Phillip Rogaway and Burt Kaliski to facilitate implementation and integration into existing protocols.

1.1.2 Outline of the primitive

The public key is an RSA modulus $n = pq$ with intractable facorisation. The parameters are the octetlength l_n of the modulus, the octetlength l_{H} of the output of a hash function Hash and the public exponent $e \geq 3$.

The signing algorithm works as follows: first one computes a hash value h from the message m and a random seed r, then the bijective PSS encoding of h and r to an element $x \in \mathbb{Z}/n\mathbb{Z}$. The signature appendix is $s = x^{1/e} \bmod n$.

The verification algorithm works as follows: The element $x = s^e \bmod n$ is computed. Then h and r are deduced from x with the inverse of the PSS encoding. The value h is compared with the hash of the message and r.

The PSS encoding hides the random seed r before it is raised to the power $1/e$. Essentially, it computes $x = (r \oplus \mathsf{MGF}(h)) \| h$ where MGF is a mask generating function.

1.1.3 Security and performance

There is a security proof of RSA-PSS that uses an existential forger under adaptive chosen message attack to solve the RSA problem, in the random oracle model, with tight reduction [54, 151, 322]. This “proof” is a strong security argument. It means that it is very likely that the best technique to forge a new RSA-PSS signature if one had access to a black box that makes valid signatures is to compute e-th roots modulo the public key, or to find a collision in the hash function.

If a source of external randomness is not available then RSA-PSS becomes the older RSA-FDH scheme, which has similar proven security, but with a loose reduction. This means that the modulus length should probably be increased in that case.

Side-channel attacks against RSA-PSS are a threat, but the fact that the signature algorithm is randomised protects against some of those attacks.

The performances of RSA-PSS are very good for signature verification, in particular for very small public exponents like 3. The performances for signature generation are not so good, and the length of the appendix might be considered to large.

A variant of RSA-PSS allows partial message recovery. It has same security and performances, but the signed message is shorter.

A public modulus $n = pqr$ or $n = p^q$ can speedup the signature generation, but the security may be lower and the first variant is patented.

1.2 Description

1.2.1 Parameter generation

First a security parameter k should be chosen, and will give an indication on the computing power that an attacker should need to break the scheme. The minimal octetlength l_n of the modulus and the minimal octetlength l_{H} of the output of the hash function are deduced from this security parameter according to the following table.

k	72	80	112	128
l_n	128 1024/8	192 1536/8	512 4096/8	750 6000/8
l_{H}	18 144/8	20 160/8	28 224/8	32 256/8

Any odd exponent $e \geq 3$ can be chosen, but it is recommended to use $e = 65537$ or distinct random e for each public key, unless fast verification is a priority and $e = 3$ is chosen.

Another parameter is the hash function used in the scheme. For more flexibility, a hash function identifier may be embedded in the signature. Therefore a parameter of the scheme is the function named ChooseHash, which can be of two types. Either this function takes as input a one-octet string which is the value BC and outputs a l_{H}-bit hash function, or this function takes as input a two-octet string $HashID \| \mathsf{CC}$ (with $HashID$ being some hash function identifier) and outputs a l_{H}-bit hash function depending on $HashID$. Any secure hash function can be chosen. SHA-1 is the default value used in the test vectors.

1.2.2 Key generation

When given a modulus octetlength l_n, the public exponent e and a source of randomness, many different algorithms can be described for generating the keys.

The output distribution of the resulting public (and private) keys is dependent on the algorithm, and in particular is dependent on the technique used to generate random prime numbers. Such techniques are described e.g. in ISO-18032 [311] and the PrimeGen routine used below is supposed to generate impredictible random primes of a given size.

An example of key generation algorithm is given below. The important criterion for a key generation algorithm for RSA-PSS is that is does not generate weak keys and that the keys are impredictible (one necessary condition is that its output set is sufficiently large). The techniques for generating RSA keys have not been evaluated by NESSIE, therefore the following algorithm is given as an example.

Description:

Input: The modulus length l_n
The odd exponent $e \geq 3$
A source of randomness $\mathcal{R}$
Output: An RSA-PSS public key pk
An RSA-PSS secret key sk

1. Generate a prime p of length $\lceil l_n/2 \rceil$ using the PrimeGen($\mathcal{R}$, $\lceil l_n/2 \rceil$) routine.
2. Check that $\mathrm{GCD}(e, p-1) = 1$. If not, goto step 1.
3. Generate a prime q of length $\lceil l_n/2 \rceil$ using the PrimeGen($\mathcal{R}$, $\lceil l_n/2 \rceil$) routine.
4. Check that $q \neq p$. If not, goto step 3.
5. Check that and $\mathrm{GCD}(e, q-1) = 1$. If not, goto step 3.
6. Set $n := pq$.
7. Check that n has length l_n. If not, goto step 1.
8. Check that $e < \mathrm{LCM}(p-1, q-1)$. If not, goto step 1.
9. Set $d \equiv e^{-1} \bmod (p-1)(q-1)$.
10. Check that $d > n^{0.5}$. If not, goto step 1.
11. Set $pk := n$.
12. Set $sk := (n, d)$ or $sk := (p, q)$.
13. Output the public key pk and the secret key sk.

Note that steps 4, 8 and 10 can be omitted. Since the PrimeGen routine is supposed to generate impredictible random primes, the checks made in these steps will be OK with very high probability.

Parameter and public key certification. The parameters and the public key should be certified. Public Key Infrastructures were not in the scope of the evaluation made by the NESSIE project, we will make the hypothesis that a public key pk is distributed with some certification data *cert*, which commits the public key to some parameters and to some validity deadline.

1.2.3 Signature generation

We describe below the signature generation algorithm that is recommended by NESSIE, which is slightly different from the submitted signature generation algorithm.

Description:

Input: The parameters e, l_n, l_{H} and $\mathsf{ChooseHash}$
The private key sk and the certification $cert$
The message m
The random seed r of octetlength l_r
Output: The appendix s

1. Choose a trailer value TF and deduces $\mathsf{Hash} = \mathsf{ChooseHash}(TF)$.
2. Compute $h_0 = \mathsf{Hash}(cert \| m)$.
3. Compute $h = \mathsf{Hash}(0_{(8\text{ octets})} \| h_0 \| r)$.
4. Compute $c = 0...01 \| r$ of length $l_c = l_n - l_{\mathsf{H}} - t - 1$ where t is the octetlength of TF.
5. Compute $a = c \oplus \mathsf{MGF}(h, l_c)$ where MGF is defined by $Mech1$ (cf. sect. 3) and outputs the first octets of $\mathsf{Hash}(h \| 0_{(4\text{ octets})}) \| \mathsf{Hash}(h \| 1_{(4\text{ octets})}) \| ...$
6. Compute the integer $x = \mathsf{OS2IP}(a \| h \| TF)$.
7. Compute the integer $s = x^{1/e} \bmod n$.

The main differences between this description and the RSA-PSS signature scheme that was submitted to NESSIE are:

– The mapping $\mathsf{ChooseHash} : TF \to \mathsf{Hash}$ is required to be explicitely described as a parameter of the scheme when the public key is validated and disseminated. This is a restriction compared to the submitted scheme where $HashID$ could be any value defined in ISO/IEC 10118.
– If $\mathsf{ChooseHash}$ requires a two-octet trailer field, then all hash functions that are allowed have the same output length. This is a restriction compared to the submitted scheme.
– The certification data is prepended to the message before hashing. Compatibility with the submitted scheme is obtained if the certification data is empty.

1.2.4 Signature verification

We describe below the signature verification algorithm that is recommended by NESSIE, which is slightly different from the submitted signature verification algorithm.

Description:

Input: The parameters e, l_n, l_{H} and $\mathsf{ChooseHash}$
The public key n and the certification $cert$
The message m and the appendix s
Output: The boolean value valid / invalid

1. Compute the integer $x = s^e \bmod n$.
2. Compute the octet string $a \| b \| TF = \mathsf{I2OSP}(x, ...)$ where TF has one or two octets (depending on $\mathsf{ChooseHash}$) and b has l_H bits.
3. If TF is valid then deduce $\mathsf{Hash} = \mathsf{ChooseHash}(TF)$, else the output will be $\mathsf{invalid}$ and Hash takes an arbitrary value.
4. Compute $h_0 = \mathsf{Hash}(cert \| m)$.
5. Compute $c = a \oplus \mathsf{MGF}(b, l_c)$ where MGF is defined by $Mech1$ (cf. sect. 3) and outputs the first octets of $\mathsf{Hash}(b \| 0_{(4 \text{ octets})}) \| \mathsf{Hash}(b \| 1_{(4 \text{ octets})}) \| ...$
6. If the first octets of c are zeroes followed by the `01` octet, compute r such that $c = 0...01 \| r$, else give an arbitrary value to r and the output will be $\mathsf{invalid}$.
7. Compute $h = \mathsf{Hash}(0_{(8 \text{ octets})} \| h_0 \| r)$.
8. Output $\mathsf{invalid}$ if $b \neq h$ or the output was previously set to $\mathsf{invalid}$, else output valid.

1.2.5 Guidelines for implementation

1.3 Test vectors

The following test vectors are generated for RSA-PSS with the default parameters for a security parameter $k = 80$. The modulus length is $l_n = 192$ octets, which corresponds to 1536 bits, the public exponent is $e = 65537$ and the hash function is SHA-1, with a trailer field of `BC`. Here the certification data $cert$ is the empty string.

1.3.1 Test vector for signature generation

```
n  =  9923916CF5589CE08EB945D635AE4534 2D443EC2E7D46980CCF48DC877ADA2C1
      0F92A33D77D8AFB83DEA73C48CB5D42F 5A9D34F10A004C6904EEFFAEFF1DB64B
      6DD770283F4F9B67370F570353DE0DFD 392CF6AA35FCE915D0F45B8087D90CBD
      CE2C5C1205680ED36E69D5FC1C46D5C2 7BFA5BF0AD8D2C94C454EF33F21254EE
      2704C031EEE0F19282D6F104A566B434 A5B562F24308FC2598BFBC9CEB7277FA
      6149CD20C217A8AF9CEC19791C3D5DDA 4721877675F3ACB8B80E4012EA3622B1
d  =  870401F293B9C5CE82673CF868A9B660 134CE91CC472D575F6BDE2C78D24ACAB
      1484CFA1A1298D7B9E3338506152EAB9 B9658348C4ED9070C325C88DCC65B0D4
      7E0A84DB273E93A003BE65940C7C69CF 097AE81B17B05CFC9C16E519C42C0C7B
      6A01AF12FF709FD952489E38ADF57776 1CA0B49A7E0CD71330C641B5CD3F3E3E
      728E489506F545158535E1011B8F85F2 D4A0285594D6530A18FB0D1895662A22
      F7D60506962412FA2EC5E0F3CA0CA6DC 2B139053E0A15E53D95A337A7F3AF731
m  =  0000000000000000000000000000000000000000
r  =  00000000000000000000
s  =  4E436D1345A84E64413168077B2304AC 9DB2F5CCE403FEAC076398ECA093FD3A
      40D8AA3E2AFD5E49C3AA60BE57AB8622 884341E1D8A9047DFC95DE5A2010E056
      CF9F5BC39B4E54C878BCD8BB688A56F6 A12BE4BBF05BCC30C2A4F3A39252BC9F
      FC2289468755FD8B8F2CEE9ECBA90D22 8715EE8D48AA85ADF139AB31DA3BA3F4
      4DEDB6EF06EE47CF23EB178C0417AD13 F9148212A0C6CE2115AEBEB2A55F26EC
      8EDF92B1571E8DA133C5F1DCBE3200C4 D83B6D808732A015EC82750CFBF5112E
```

1.3.2 Test vectors for signature verification

The following is a valid signature.

```
 n  =  9923916CF5589CE08EB945D635AE4534 2D443EC2E7D46980CCF48DC877ADA2C1
       0F92A33D77D8AFB83DEA73C48CB5D42F 5A9D34F10A004C6904EEFFAEFF1DB64B
       6DD770283F4F9B67370F570353DE0DFD 392CF6AA35FCE915D0F45B8087D90CBD
       CE2C5C1205680ED36E69D5FC1C46D5C2 7BFA5BF0AD8D2C94C454EF33F21254EE
       2704C031EEE0F19282D6F104A566B434 A5B562F24308FC2598BFBC9CEB7277FA
       6149CD20C217A8AF9CEC19791C3D5DDA 4721877675F3ACB8B80E4012EA3622B1
 m  =  BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
 s  =  8ADA14B5C5DDF605C836960DE14E50ED 4A10378120D13928FBA4DDC3F8CB748C
       9CD66286E161D939FF30BB55E62C3FD0 8091D0BD6E3A2EF558EFDE0F2340247F
       7CC340123FB8795EFAE56742534BAE52 A1D445C15BFCCE70F46DF223EDB34671
       D3C721E72048CF54C77A9DEE8E9E4B5C 851374690A8D4EF7BE95ABA9819D8A1D
       DD2DC551EE748C534B28FB0D50421A42 05A9049E0F3DCF74792F42591E6812A9
       2F0624C71E7D3FA3598BA4150792F677 D0ECB5AE860761210A4DE31479FB5094
```

The following is an invalid signature where invalidity is detected at step 3.

```
 n  =  9923916CF5589CE08EB945D635AE4534 2D443EC2E7D46980CCF48DC877ADA2C1
       0F92A33D77D8AFB83DEA73C48CB5D42F 5A9D34F10A004C6904EEFFAEFF1DB64B
       6DD770283F4F9B67370F570353DE0DFD 392CF6AA35FCE915D0F45B8087D90CBD
       CE2C5C1205680ED36E69D5FC1C46D5C2 7BFA5BF0AD8D2C94C454EF33F21254EE
       2704C031EEE0F19282D6F104A566B434 A5B562F24308FC2598BFBC9CEB7277FA
       6149CD20C217A8AF9CEC19791C3D5DDA 4721877675F3ACB8B80E4012EA3622B1
 m  =  BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
 s  =  33333333333333333333333333333333 33333333333333333333333333333333
       33333333333333333333333333333333 33333333333333333333333333333333
       33333333333333333333333333333333 33333333333333333333333333333333
       33333333333333333333333333333333 33333333333333333333333333333333
       33333333333333333333333333333333 33333333333333333333333333333333
       33333333333333333333333333333333 33333333333333333333333333333333
```

The following is an invalid signature where invalidity is detected at step 6.

```
 n  =  9923916CF5589CE08EB945D635AE4534 2D443EC2E7D46980CCF48DC877ADA2C1
       0F92A33D77D8AFB83DEA73C48CB5D42F 5A9D34F10A004C6904EEFFAEFF1DB64B
       6DD770283F4F9B67370F570353DE0DFD 392CF6AA35FCE915D0F45B8087D90CBD
       CE2C5C1205680ED36E69D5FC1C46D5C2 7BFA5BF0AD8D2C94C454EF33F21254EE
       2704C031EEE0F19282D6F104A566B434 A5B562F24308FC2598BFBC9CEB7277FA
       6149CD20C217A8AF9CEC19791C3D5DDA 4721877675F3ACB8B80E4012EA3622B1
 m  =  BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
 s  =  0631045A205ECCB2ABBE872614B90D8A F0890C96D6D96112CCFCDC522A961539
       7D675446D5ADFFB92C68BF57DD70DB04 490C27D0EF439C781B6727616F9A133A
       33CAD7A7327F0DBD7D0EF9C0204004A4 1D4F51852F8B60A4870F2EF0A0FB5C08
       19E41FA31D44CB9A578E00405C268E12 D563922C5A79AEE0C6D062A349950135
       675DACE973138F78AEFE6534D71C398A 581E1D8482523941174E88EEB5E07686
       37BE874C8C94C50F112DAF01B37F4831 837C0C189123CE9246AAA87923317742
```

The following is an invalid signature where invalidity is detected at step 8.

```
n  =  9923916CF5589CE08EB945D635AE4534 2D443EC2E7D46980CCF48DC877ADA2C1
      0F92A33D77D8AFB83DEA73C48CB5D42F 5A9D34F10A004C6904EEFFAEFF1DB64B
      6DD770283F4F9B67370F570353DE0DFD 392CF6AA35FCE915D0F45B8087D90CBD
      CE2C5C1205680ED36E69D5FC1C46D5C2 7BFA5BF0AD8D2C94C454EF33F21254EE
      2704C031EEE0F19282D6F104A566B434 A5B562F24308FC2598BFBC9CEB7277FA
      6149CD20C217A8AF9CEC19791C3D5DDA 4721877675F3ACB8B80E4012EA3622B1
m  =  3333333333333333333333333333333333333333
s  =  8ADA14B5C5DDF605C836960DE14E50ED 4A10378120D13928FBA4DDC3F8CB748C
      9CD66286E161D939FF30BB55E62C3FD0 8091D0BD6E3A2EF558EFDE0F2340247F
      7CC340123FB8795EFAE56742534BAE52 A1D445C15BFCCE70F46DF223EDB34671
      D3C721E72048CF54C77A9DEE8E9E4B5C 851374690A8D4EF7BE95ABA9819D8A1D
      DD2DC551EE748C534B28FB0D50421A42 05A9049E0F3DCF74792F42591E6812A9
      2F0624C71E7D3FA3598BA4150792F677 D0ECB5AE860761210A4DE31479FB5094
```

2. ECDSA

2.1 Introduction

2.1.1 Overview

The Elliptic Curve Digital Signature Scheme (ECDSA), submitted to NESSIE by Certicom Corp., is the elliptic curve analog of the Digital Signature Scheme (DSA). It is a randomized signature scheme with appendix, based on the discrete logarithm problem on elliptic curves. Ellipitic curve cryptosystems first appeared in 1985, and ECDSA was proposed in 1992 by Scott Vanstone [610]. It was accepted as an ISO [309] standard in 1998, accepted as an ANSI [21] standard in 1999 and accepetd in 2000 as an IEEE [299] standard.

2.1.2 Outline of the primitive

The parameters of the scheme are a well chosen elliptic curve $E(\mathbb{F})$ over a suitable finite field $\mathbb{F}$ and a point G on this curve such that the subgroup $\langle G \rangle$ generated by G has prime order q. The scheme makes use of a function H that takes an input of arbitrary length, and outputs an element of $\{1, \ldots, q-1\}$. The private key is an integer d in $\{1, \ldots, q-1\}$. The public key is the point Q of $E(\mathbb{F})$ defined by $Q = dG$.

The signing algorithm for a message m works as follows: the signer chooses at random an integer k in $\{1, \ldots, q-1\}$ and computes the point $R = kG$, which first coordinate is denoted x_R. He then computes the integers $r = x_R \bmod q$, $e = \mathsf{H}(m)$ and $s = k^{-1}(e + dr) \bmod q$. If $s = 0$, the signer chooses another k, else, the appendix is (r, s).

The verification algorithm works as follows: the receiver first checks that s is in $\{1, \ldots, q-1\}$. He computes $e = \mathsf{H}(m)$, $u_1 = es^{-1} \bmod q$, $u_2 = rs^{-1} \bmod q$ and the point $T = u_1 G + u_2 Q$. If $T = \mathcal{O}$, the signature is rejected. Else, the receiver compares r to $x_T \bmod q$, where x_T is the first coordinate of T.

2.1.3 Security and performance

There is no proof of security for ECDSA, neither in the standard model nor in the random oracle model. Recently, Brown has published a proof of security in the generic group model that applies to ECDSA but some specific properties of the scheme invalidate this generic proof. However, there exist proofs in the random oracle model that apply to variants of ECDSA, suggesting that ECDSA is secure.

ECDSA is subject to some side-channel attacks, in particular the scalar multiplication operation. However, efficient countermeasures can be implemented. An overview can be found in [508].

From the perfomance point of view, the main advantage of ECDSA is the keys and signature lengths, which are really short. For instance, with a field of size 160 bits, the public key is 40 bytes long, the private one is 20 bytes long and the signature length is 40 bytes. Signature generation and verification are also really fast.

The choice of the field (a prime or a characteristic 2 finite field) can influence the performances.

2.2 Description

2.2.1 Parameter generation

The set of parameters for ECDSA is $\mathsf{Param} = (l, FI, a, b, G, q, h, Hid)$, where l is the security parameter. FI is the $FieldIdentifier$ that specifies the finite field $\mathbb{F}$. If $\mathbb{F} = \mathbb{F}_p$, with p an odd prime, $FI = (p)$. If $\mathbb{F} = \mathbb{F}_{2^m}$, $FI = (m, f(X))$, where $f(X)$ is the irreducible polynomial specifying the polynomial basis representation used for the elements of the field. The integers a and b are the coefficients that define an elliptic curve $E(\mathbb{F})$. G is a point on $E(\mathbb{F})$ of order a large prime q, and h is the cofactor $h = \#E(\mathbb{F})/q$. A parameter of the scheme is the function H that takes an input of arbitrary length and outputs a element of $\{1, \ldots, q-1\}$. Hid is a value that indicates the function H that has been chosen. The parameters are subject to the following requirements:

- taking logarithms on $E(\mathbb{F})$ should be intractable;
- $\#E(\mathbb{F}) \neq \#\mathbb{F}$;
- $(\#\mathbb{F})^B \not\equiv 1 \bmod q$ for any $1 \leq B < 20$;
- $h \leq 4$.

The user can either generate himself these parameters, or choose to use recommended ones. Methods to generate the parameters can be found in [300] and [136], and recommended parameters are available in [137].

2.2.1.1 Key generation

The key generation algorithm inputs a valid set of parameters for the ECDSA, and outputs a valid key pair $(\mathsf{pk}, \mathsf{sk})$.

Description:

Input: The set of parameters Param
Output: An ECDSA public key pk
An ECDSA secret key sk

1. Generate a random integer d such that $d \in \{1, \ldots, q-1\}$.

2. Compute the point $Q := dG$.
3. Set $\mathsf{pk} := Q$ and $\mathsf{sk} := d$.
4. Output the public key pk and the secret key sk.

Parameter and public key certification. It is necessary to ensure that the domain parameters and the public key have the requise properties. This is done to avoid attacks enabled by the use of invalid parameters. Methods for validating domain parameters and public keys can be found in [300] and [136].

2.2.1.2 Signature generation

The signature generation is described below. It uses the `sGen` routine, which takes as additional input an integer randomly generated.

Description of the signature generation algorithm:

Input: The parameters Param
The secret key $\mathsf{sk} = d$
The message m
Output: The appendix S

1. Run the `sGen` routine.
2. If `sGen` returns "`fail`", go back to step 1.
3. If `sGen` returns the pair (r, s), output the appendix:

$$S := \mathsf{I2OSP}(r)\|\mathsf{I2OSP}(s).$$

Description of the `sGen` routine:

Input: The parameters Param
The secret key $\mathsf{sk} = d$
The message m
A random integer k in $\{1, \ldots, q-1\}$
Output: The integer pair (r, s) or "`fail`"

1. Compute the point $R := kG$. Let $R = (x_R, y_R)$.
2. Convert the field element x_R to an integer $i := \mathsf{FE2IP}(x_R)$.
3. Compute $r := i \bmod q$.
4. Compute the integer $e := \mathsf{H}(m)$.
5. Compute the integer $s := k^{-1}(e + dr) \bmod q$.
   If $s = 0$, return "`fail`".
6. Output the integer pair (r, s).

2.2.1.3 Signature verification

Below is escribed the signature verification algorithm recommended by NESSIE, which is slightly different from the submitted one.

Description:

Input: The parameters Param
The public key pk
The message m and the appendix S
Ouput: The boolean value valid / invalid

1. Parse the octet string S as $S_1 \| S_2$, where S_1 and S_2 are $\lceil \log_2(q)/8 \rceil$ octet long.
2. Compute the integers $r := \mathsf{OS2IP}(S_1)$ and $s := \mathsf{OS2IP}(S_2)$.
3. Verify that $s \in \{1, \ldots, q-1\}$.
4. Compute the integer $e := \mathsf{H}(m)$.
5. Compute the integer $w := s^{-1} \bmod q$.
6. Compute the integer $u_1 := ew \bmod q$.
7. Compute the integer $u_2 := rw \bmod q$.
8. Compute the point $T := u_1 G + u_2 Q$. Let $T = (x_T, y_T)$.
   If $T = \mathcal{O}$, output invalid and stop.
9. Convert the field element x_T to an integer $j := \mathsf{FE2IP}(x_R)$.
10. Compute the integer $t := j \bmod q$.
11. If $t = r$, output valid.
    Else output invalid.

2.3 Test vectors

TODO

3. SFLASH

3.1 Introduction

3.1.1 Overview

The SFLASH signature scheme described here is a modified version of the original submission SFLASH. The SFLASH signature scheme has been submitted to NESSIE by Patarin *et al.* [514]. It is based on multivariate polynomials. The first cryptosystem of this kind has been proposed in 1988 by Matsumoto and Imai [428]: the C^* cryptosystem. It has been broken by Patarin in 1995 [512]. A variation of the scheme, secure against this attack and called the C^{*--} cryptosystem, has been proposed by Patarin, Courtois and Goubin in 1998 [515]. SFLASH is a C^{*--} scheme with a special choice of the parameters. It has been broken by Gilbert and Minier [259], so the scheme has been modified to avoid this attack. The only difference between SFLASH (new version) and the original submission SFLASH is the choice of the underlying field.

At the time when NESSIE made its selection, for the size of the parameters of SFLASH were such that all attacks against the underlying MQ hard problem required more computational power than 2^{80} Triple-DES operations. However, some improvements have been published since then [156]. The size of the parameters of SFLASH have been increased to resist this new attack and a new variant of SFLASH is recommended by its submitters [162, 163].

3.1.2 Outline of the primitive

The underlying field is $K = \mathbb{F}_{128}$. $\mathcal{L}$ denotes an extension of degree 37 of K. $\mathbf{F}$ is the function from $\mathcal{L}$ to $\mathcal{L}$ defined by $\mathbf{F}(x) = x^{128^{11}+1}$.

The secret key consists in two affine bijections $\mathbf{s}$ and $\mathbf{t}$ of K^{37} and a 80-bit string Δ. The public key is the function $\mathbf{G}$ from K^{37} to K^{26} defined as the 26 polynomials $(P_1, \ldots, P_{26})$, where $(P_1, \ldots, P_{37})$ are the quadratic polynomials describing the function $\mathbf{t} \circ \mathbf{F} \circ \mathbf{s}$ from K^{37} to K^{37}.

The signing algorithm works as follows: the message is hashed in a 182-bit string y, which is then hashed together with the secret string Δ to give a 77-bit string r. The signature appendix is $s = \mathbf{s}^{-1} \circ \mathbf{F}^{-1} \circ \mathbf{t}^{-1}(y \| r)$.

The verification algorithm works as follows: the message is hashed as above into y. The verifier then computes the value $y' = \mathbf{G}(s)$ and compares it with y.

3.1.3 Security and performances

The security of SFLASH is based on the one-wayness of the function $\mathbf{G}$, which is not well defined. Indeed, there is no proof of security that reduces the security of SFLASH to a trusted mathematical problem. However, the results concerning the two kind of possible attacks are sufficient for short term security.

The first attack is to solve a random set of quadratic equations. This is the MQ problem and it is NP-hard. However, some relatively efficient algorithms exist to solve this problem [128, 164, 224], but for the parameters of SFLASH, they require more computational effort than 2^{80} Triple-DES.

The second one is to attack the C^{*--} scheme [515], but this would also require more than 2^{80} Triple-DES.

SFLASH has been designed to be a very fast signature scheme, both for signature generation and verification. Furthermore, it is designed to be implemented on low-cost smart-cards, without any co-processor. A highly optimized such implementation is described in [12], and SFLASH seems to be the fastest signature scheme on low-cost smart-cards. The drawback is the size of the public key, which might be found too large.

3.2 Description

3.2.1 Parameters of the scheme

The SFLASH signature scheme uses two fields and a function which are fixed.

The first field is $K = \mathbb{F}_{128} = \mathbb{F}_2[X]/(X^7 + X + 1)$, It is represented as a $\mathbb{F}_2$-vector space by the bijection π between ${\mathbb{F}_2}^7$ and K defined by:

$$\forall b = (b_0, \ldots, b_6) \in \{0,1\}^7, \pi(b) = b_6 X^6 + \ldots + b_1 X + b_0$$

The second one is the extension of K of degree 37 defined by $\mathcal{L} = K[X]/(X^{37} + X^{12} + X^{10} + X^2 + 1)$. The field $\mathcal{L}$ is represented as a K-vector space by the bijection ϕ between K^{37} and $\mathcal{L}$ defined by:

$$\forall a = (a_0, \ldots, a_{36}) \in K^{37}, \phi(a) = a_{36} X^{36} + \ldots + a_1 X + a_0$$

The function that will be used is the function $\mathbf{F}$ from $\mathcal{L}$ to $\mathcal{L}$ defined by $\mathbf{F}(x) = x^{128^{11}+1}$.

3.2.2 Key generation

The secret key consists in two affine bijections of K^{37}, $\mathbf{s}$ and $\mathbf{t}$, and an 80-bit string Δ. The affine bijection $\mathbf{s}$ (resp. $\mathbf{t}$) will be described by the 37×37 square matrix S_L (resp. T_L) and the 37×1 column matrix S_C (resp. T_C) over K with respect to the canonical basis of K^{37}.

So, in order to generate the complete key of SFLASH, the following elements have to be generated:

- The secret invertible 37×37 matrix S_L and the secret 37×1 column matrix S_C, all the coefficients being in K, that describes the affine bijection $\mathbf{s}$.
- The secret invertible 37×37 matrix T_L and the secret 37×1 column matrix T_C, all the coefficients being in K, that describes the affine bijection $\mathbf{t}$.
- The 80-bit secret string Δ.

Note that generating an element of K is equivalent to generating a 7-bit string.

The key generation uses the following functions, described below:

- the function `Aff-Bij-Gen` generates a 37×37 invertible matrix and a 37×1 column matrix.
- the function `next-8bit-random-string` outputs a random 8-bit string.

The key generation is then as follows, representing Δ as a 10-byte string:

```
1. $(S_L, S_C)$ := Aff-Bij-Gen()
2. $(T_L, T_C)$ := Aff-Bij-Gen()
3. for $i$ from 0 to 9
     $\Delta[i]$ := next-8bit-random-string()
```

The public key is deduced from the secret one. By construction, the function $\mathbf{t} \circ \phi^{-1} \circ \mathbf{F} \circ \phi \circ \mathbf{s}$ from K^{37} to K^{37} is a quadratic transformation over K, *i.e.* for $x = (x_0, \ldots, x_{36}), y = (y_0, \ldots, y_{36})$ in K^{37}, the equation $y = \mathbf{t} \circ \phi^{-1} \circ \mathbf{F} \circ \phi \circ \mathbf{s}\,(x)$ can be written as follows:

$$\left\{ \begin{array}{ccc} y_0 & = & P_0(x_0, \ldots, x_{36}) \\ & \vdots & \\ y_{36} & = & P_{36}(x_0, \ldots, x_{36}) \end{array} \right.$$

where each P_i is a quadratic polynomial of the form:

$$P_i(x_0, \ldots, x_{36}) = \sum_{0 \le j < k < 37} \zeta_{i,j,k}\, x_j x_k + \sum_{0 \le j < 37} \nu_{i,j}\, x_j + \rho_i$$

The public key is the function $\mathbf{G}$ from K^{37} to K^{26} defined by the 26 first quadratic polynomials, *i.e.* $\mathbf{G}(x_0, \ldots, x_{36}) = (y_0, \ldots, y_{25})$, and is computed from the secret key.

3.2.2.1 The function `Aff-Bij-Gen`

This function generates a random 37×37 invertible matrix with entries in K. Two different methods can be used.

The first method is to generate a random 37×37 matrix with entries in K until it is invertible.

The second one is to generate an invertible 37×37 matrix by using the LU decomposition. Suitable random matrix L and U are generated, and the product LU is an invertible matrix.

Description: (using the LU decomposition)
Input:
Output: a 37×37 invertible matrix and a 37×1 column matrix

```
1. for $i$ from 1 to 37 do
   a) $C_i$ := (next-8bit-random-string() & 127)
   b) for $j$ from 1 to 37 do
      i. if $i < j$ then
         $U_{i,j}$ := (next-8bit-random-string() & 127 )
         $L_{i,j} := 1$
      ii. else if $i > j$ then
         $L_{i,j}$ := (next-8bit-random-string() & 127 )
         $U_{i,j} := 1$
      iii. else if $i = j$ then
         do $z$ := (next-8bit-random-string() & 127) until $z \neq 0$
         $U_{i,j} := z$
         $U_{i,j} := 1$
2. Compute $M = LU$.
3. Return $M, C$.
```

3.2.2.2 The function next-8bit-random-string

This function outputs a random 8-bit string each time it is asked to. It is initialised with a seed, and uses the hash function SHA-1.

Initialization: tab is a string of 55 bytes.

$$\texttt{tab} = \texttt{seed} \| \texttt{count}_3 \| \texttt{count}_2 \| \texttt{count}_1 \| \texttt{count}_0$$

The first 51 bytes are the seed seed: it is a byte string that should have a minimum of 80 bits of entropy, and its size is 51 bytes (if it is shorter, it is padded with 0s). The last 4 bytes of tab are used as a counter, initialized to 0.

$$\texttt{count} = \texttt{count}_3\, 256^3 + \texttt{count}_2\, 256^2 + \texttt{count}_1\, 256 + \texttt{count}_0$$

rando is a string of 20 bytes, position is a byte initialized to 19.

Description:

Input: tab, rando, position.
Output: an 8-bit string.

```
1. if position = 19 then
   a) count := count + 1
   b) rando := sha1(tab)
   c) position := $-1$
2. position := position+1
3. return rando[position]
```

3.2.3 Signature generation

Let the bit-string m be the message to sign. The signing algorithm works as follows:

1. Compute the 160-bit strings h_1 and h_2 as:

$$h_1 = \text{SHA-1}(m), \quad h_2 = \text{SHA-1}(h_1).$$

2. Let v be the following 182-bit string:

$$v = [h_1 \| h_2]_{0 \to 181}$$

3. Then compute the 77-bit string:

$$w = [\text{SHA-1}(v \| \varDelta)]_{0 \to 76}$$

4. Let y be the string of 26 elements of K defined by:

$$y = \pi([v]_{0 \to 6}), \pi([v]_{7 \to 13}), \dots, \pi([v]_{175 \to 181})$$

5. Let r be the string of 11 elements of K defined by:

$$r = \pi([w]_{0 \to 6}), \pi([w]_{7 \to 13}), \dots, \pi([w]_{70 \to 76})$$

6. Let x be the string of 37 elements of K defined by:

$$x = (x_0, \dots, x_{36}) = \mathbf{s}^{-1} \circ \phi^{-1} \circ \mathbf{F}^{-1} \circ \phi \circ \mathbf{t}^{-1}(y \| r)$$

7. The signature appendix is the 259-bit string:

$$s = \pi^{-1}(x_0) \| \dots \| \pi^{-1}(x_{36})$$

3.2.4 Signature verification

To verify a signed message (m, s), the following algorithm is used:

1. Compute the 160-bit string h_1 and h_2 as:

$$h_1 = \text{SHA-1}(m), \quad h_2 = \text{SHA-1}(h_1).$$

2. Let v be the following 182-bit string:

$$v = [h_1 \| h_2]_{0 \to 181}$$

3. Let y be the string of 26 elements of K defined by:

$$y = \pi([v]_{0 \to 6}), \pi([v]_{7 \to 13}), \dots, \pi([v]_{175 \to 181})$$

4. Let y' be the string of 26 elements of K defined by:

$$y' = \mathbf{G}\left(\pi([s]_{0 \to 6}), \pi([s]_{7 \to 13}), \dots, \pi([s]_{252 \to 258})\right)$$

5. If $y' = y$, accept the signature. Else, reject it.

3.2.5 Guidelines for implementation

As SFLASH has been specifically designed to be used on smart-cards, the following are guidelines for the implementation of the scheme on these plate-forms, where one should try to use as less memory as possible. More details can be found in [12].

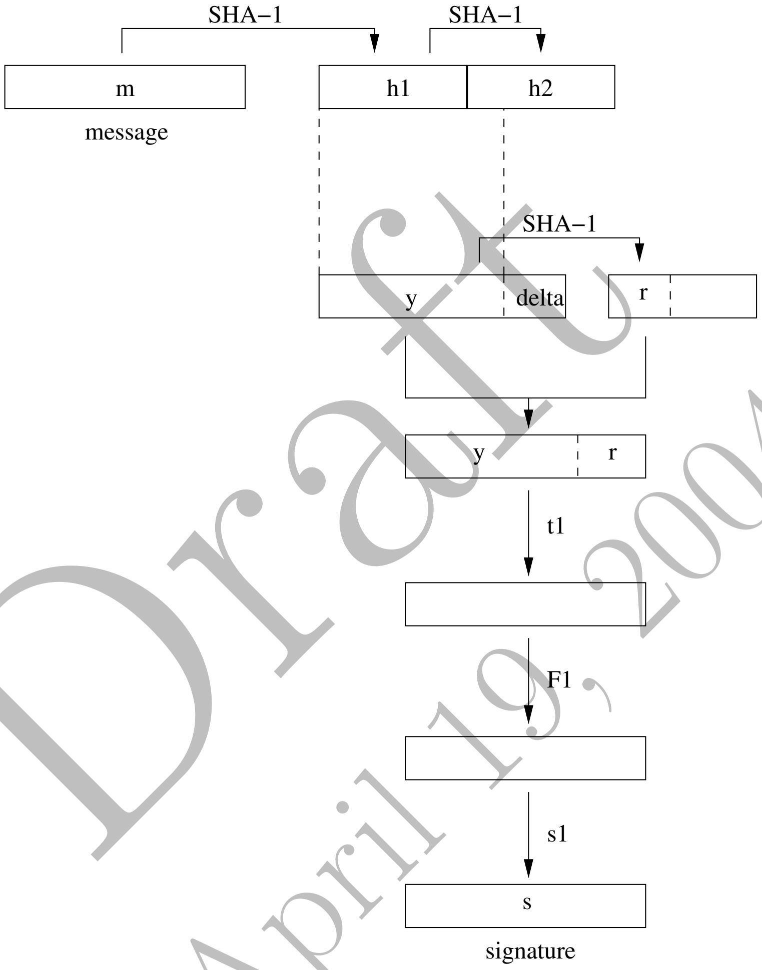


Fig. 60. The SFLASH signature generation algorithm.

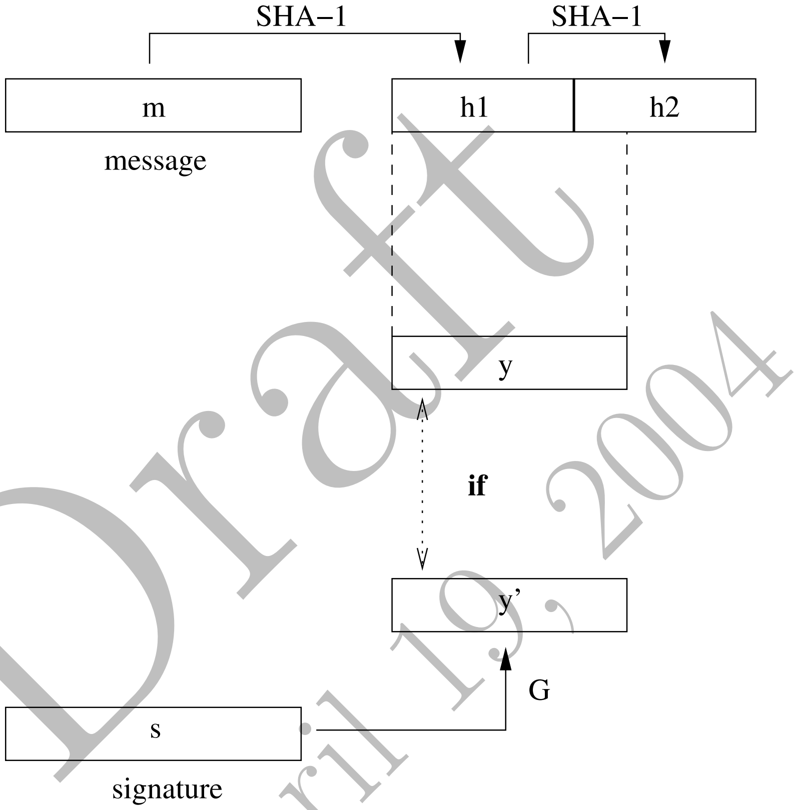


Fig. 61. The SFLASH signature verification algorithm.

3.2.5.1 Computation in K and $\mathcal{L}$

Addition is easy both in K and in $\mathcal{L}$. The field K is of characteristic 2 so two elements are added by XORing their representations. In $\mathcal{L}$, addition of two elements is done by adding the corresponding coefficients in their polynomial representation.

The simplest way to implement multiplication in K is to use two tables. Indeed, the multiplicative group K^* is cyclic, generated by an element α, so each element of K can be seen as a power of α. One table Exp implements the exponentiation:

$$\mathrm{Exp}[e] = \alpha^e,$$

and the second one Log gives the exponent of a non-zero element:

$$\mathrm{Log}[x] = e_x, \text{ where } x = \alpha^{e_x}.$$

The product of two elements x, y in K is then:

$$xy = \mathrm{Exp}\left[Log[x] + Log[y]\right].$$

Mulitplication in $\mathcal{L}$ is more costly: we have to compute the product of two polynomials, which will be of degree 72, and to reduce it modulo the irreducible ponynomial $X^{37} + X^{12} + X^{10} + X^2 + 1$. It is worth implementing seperatly the squaring of an element. Indeed, the square of a polynomial can be computed faster than multiplying it with itself since only the squares of the coefficients have to be computed, as mentionned in [12].

3.2.5.2 Computing the inverse of F

The function $\mathbf{F}$ from $\mathcal{L}$ to $\mathcal{L}$ is defined by:

$$\forall x \in \mathcal{L}, \quad \mathbf{F}(x) = x^{128^{11}+1}.$$

The signature generation algorithm requires the computation of its inverse. We have:

$$z = \mathbf{F}^{-1}(y) \Longleftrightarrow z = y^h, \text{ where } h = \left(128^{11} + 1\right)^{-1} \text{mod } (128^{37} - 1).$$

Instead of computing this with the classical “square and multiply” algorithm, Akkar *et al.* suggest in [12] to use a special addition chain, favouring the powers 128 and 128^7. Indeed, these powers are easy to handle, as the application $x \longmapsto x^{128}$ is K-linear on $\mathcal{L}$. This method drastically reduces the number of multiplications. So, to compute $z = \mathbf{F}^{-1}(y)$, we perform the following steps:

1. $z_0 := \left(y^2\right)^2$
2. $z_1 := y \times z_0$
3. $z_2 := \left({z_0}^2\right)^2$
4. $z_3 := z_1 \times z_2$
5. $z_4 := {z_3}^2 \times z_3$
6. $z_5 := \left({z_2}^2\right)^2$

7. $z_6 := \left(\left({z_5}^{128}\right)^{128}\right)^{128} \times z_4$
8. $z_7 := \left(\left({z_6}^{128} \times z_6\right)^{128} \times z_6\right)^{128}$
9. $z_8 := \left(\left(z_7 \times z_4\right)^{128}\right)^{128^7}$
10. $z_9 := {z_5}^{128^7} \times z_7 \times z_5 \times z_8$
11. $z := \left(\left(\left({z_9}^{128^7}\right)^{128^7}\right)^{128} \times z_8\right)^{128^7} \times z_9$

This method requires only 12 multiplications in $\mathcal{L}$, instead of 145 with the classical "square and multiply" algorithm.

Being linear over K, the operation $x \longmapsto x^{128^7}$ is fulfilled by a fixed matrix multiplication. This computation can be accelerated by using a technique related to the Gray code. The idea is to find a road in the matrix that minimizes the number of XORs. This technique, as explained in [12], accelerates a lot the operation, but also increases the code size.

The function $x \longmapsto x^{128} = x^{2^7}$ can be implemented by performing 7 squares, which is faster than a matrix multiplication.

3.2.5.3 Side-channel attacks

Steinwandt *et al.* [595] report DPA-based attacks against SFLASH. Akkar *et al.* [12] propose as a countermeasure to this kind of attack to mask all the intermediate values. This is done by using the homomorphic properties of the functions $\mathbf{t}, \mathbf{F}, \mathbf{s}$. The implementation of this countermeasure doubles the running time of SFLASH.

Part G

Asymmetric Identification Schemes

1. GPS

1.1 Introduction

Current asymmetric identification techniques follow the paradigm of interactive proofs of knowledge of some secret. They allow to build practical schemes that possess the zero-knowledge property, intuitively that the only information conveyed by the prover to the verifier is the knowledge of its own secret. This is an optimal property which sometimes is not malleable enough to achieve good performance. This has led to the emergence of the witness indistinguishability condition, whereby it is very difficult for an active attacker to trace back the holder of some secret. This is explained in detail in Book II, Chap. 8.

The prototypes of many identification schemes are the Fiat-Shamir scheme (see Book I, Sect. 8.4.1) and the Schnorr identification scheme (see Book I, Sect. 8.4.2), based respectively on the RSA problem and the discrete logarithm problem.

1.1.1 Overview

GPS has been submitted by ENS, France Télécom and La Poste. GPS is essentially a modified version of the Schnorr identification scheme. Unlike the Schnorr scheme, GPS uses a generator g with unknown order and the exponent is calculated in $\mathbb{Z}$ rather than modulo the order of g.

1.1.2 Outline of the primitive

The GPS identification scheme consists of ℓ iterations of an identification round, where ℓ is part of the security parameters of the scheme. We describe one round of the GPS identification scheme.

Let A, B, S be parameters with $|A| \geq |S| + |B| + 80$, $|B| = 32$ and $|S|$ greater than 140 bits. It would be better if $|S| = 180$ meaning A is approximately a 300-bit number. Let n be a RSA modulus ($n = pq$ where p, q are 512-bit primes). The factorisation of n should be unknown to all except maybe a trusted authority, which enables the scheme to be rendered identity-based. Also, let g be a random element of $\mathbb{Z}/n\mathbb{Z}$, coprime to n and $s \in [1, S]$. Define $I = g^{-s} \pmod{n}$.

Public parameters. The public parameters are A, B, S, g, I and n.

Private parameters. The private parameter is s.

Let us examine a protocol round.

1. *Commitment*: the prover picks $r \in \{0, \dots, A-1\}$ at random and sends to the verifier $x = g^r \pmod{n}$.
2. *Challenge*: the verifier chooses a random $c \in \{0, \dots, B-1\}$ and sends it to the prover.
3. *Response*: the prover checks that $c \in \{0, \dots, B-1\}$, computes $y = r + sc$ and sends it to the verifier.
4. *Verification*: the verifier checks that $y \in \{0, \dots, A + (B-1)(S-1) - 1\}$, computes $z = g^y I^c \pmod{n}$ and accepts the prover if and only if $z = x$.

Below is a schematic description of the 3-fold interaction between the prover and the verifier in one round of the protocol.

Prover		**Verifier**
choose $r \in \{0, \dots, A-1\}$ compute $x = g^r \pmod{n}$	$\xrightarrow{x}$	
	$\xleftarrow{c}$	choose $c \in \{0, \dots, B-1\}$
check $c \in \{0, \dots, B-1\}$ compute $y = r + cs$	$\xrightarrow{y}$	check $y \in \{0, \dots, A + (B-1)(S-1) - 1\}$ accept if $g^y I^c = x \pmod{n}$

1.1.3 Security and performance

The GPS protocol is computationally zero-knowledge if ℓ and B are polynomial in $|n|$ and $\ell SB/A$ is negligible. It is witness-indistinguishable if n is a product of two primes p and q such that $(p-1)/2$ and $(q-1)/2$ have no small prime factor, and if g is a quadratic residue modulo one of the prime factors and a quadratic non-residue modulo the other, provided $|B| > 2\,\mathrm{ord}(g)$.

Performance is very fast for the response step of the protocol, namely a few hundred cycles only. Other steps compare favourably against ISO/IEC 9798-5 standards (see Book II, Sect. 8.3.2) at a few million cycles. The only slow part is typically the parameter generation (done only once), since one has to choose an RSA-type modulus n.

1.2 Description

We introduce some notation for a description of an implementation of GPS. Let $\texttt{Rand}(k)$ be a function which outputs a random integer between 1 and k, possibly using an auxiliary input (salt). We denote by $(r_m \dots r_0)_2$ the representation of an integer r to the base 2.

In the Figs. 62 and 63, we suppose the relevant parameters input to the algorithms have been uploaded into some registers. We do not delve into the ways to store those quantities, contenting ourselves in calling a multiplication of X and Y as a single operation, regardless of the size of the operands.

Input: The public and private parameters A, B, S, g, I, n, s and the challenge c.
Output: Prover's transcript x, y in the execution of one round of the GPS identification scheme.

```
/* generation of a random seed $r = (r_m \dots r_0)$, with $m = |A|$ */
  Compute Rand$(A-1) = r = (r_m \dots r_0)$
/* computation of $x = g^r \pmod{n}$ by the square and multiply algorithm */
  $x \leftarrow 1$
  For $i$ from $m$ to $0$
    $x \leftarrow x^2 \pmod{n}$
    If $(r_i == 1)$ then $x \leftarrow gx \pmod{n}$
  Output $x$
  Compute $y = r + cs$
  Output $y$.
```

Fig. 62. On the prover's side

Input: The public parameters A, B, S, g, I, n and the response y
Output: 1 if the execution of one round of the GPS identification scheme is successful, 0 otherwise.

```
/* generation of a random challenge $c = (c_t \dots c_0)$, with $t = |B|$ */
  Compute $c = (c_t \dots c_0) =$ Rand$(B-1)$
/* computation of $v = g^y I^c \pmod{n}$ using the Straus-Shamir trick. We write $y = (y_u \dots y_0)$. Since $u > t$, we can also write $c = (c_u \dots c_0)$ by padding an initial string of zeroes. */
  $v \leftarrow 1$
  $a \leftarrow gI$ /* trick to speed up computation if space available */
  For $i$ from $u$ to $0$
    $v \leftarrow v^2 \pmod{n}$
    If $\big((y_i, c_i) \mathrel{!=} (0,0)\big)$ then $v \leftarrow v g^{y_i} I^{c_i} \pmod{n}$ /* using register $a$ this counts as one multiplication */
  Output $1$ if $(x == v)$
```

Fig. 63. On the verifier's side

1.3 Test vectors

We reproduce below the Maple code used to generate test vectors. The modulus n is the product of two primes p and q of 768 bits each, so that the length of n is 1536 bits.

```
> p:=79727899923569440438389630137288496738592515774469850369151286
132742793871190580855604956705926457618732713878584208818796893862O
818649562225703831516783578097686179783733631029059826238257O8833673
622381692681536310162855139678398 63;

  p := 797278999235694404383896301372884967385925157744698503691 51\
       28613274279387119058085560495670592645761873271387858420 88\
        18796893862081864956225703831516783578097686179783733631O2\
        90598262382570883367362238169268153631016285513967839863

> convert(p,hex);

  83774F2A38137979F6E131F153757FEFFC3212E558371BC067695B62E89C2D32\
        B4AC1B64DA2C34451851FAD91DB6F3A032CB6171BCF7A01C06245CA3B0\
        9B4505C4C705DCB27E3AB5A18A3A42BFC5F565FE5C40B0935188FB6FA0\
        F3FF8C25AA77

> q:=1219301903136313933826730221545775090895421676586362270369497
9157575042272331690042290290249081308752607069359200456345872961744
9755593042922772245804941200110686970610807355422705798515915654950
8944893686347331038904494739939720039;

  q := 12193019031363139338267302215457750908954216765863622703694\
        97915757504227233169004229029024908130875260706935920045 63\
        45872961744975559304292277224580494120011068697061080735 54\
        2270579851591565495089448936863473310389044947399397200 39

> convert(q,hex);

  C90E0A709F78176A6C22EC2380A2377ED6FDA45056BDFFFF3BA3C4508EC06A84\
        05D7CDAFD67EFB4E91E7C91D501B8C5A806DB9C852630CED17804CC1FB\
        C42F2BC65B5550F026DAF4F236CCFB8F99CF7149DCD3C9AA5E1F00AEC0\
        B33E5808FB67

> n:=p*q;

  n := 97212380109869796953499244473820098433967663075756597844214\
        96723505149852228551541363697963597630116116472742610067O3\
        23273393329347661140064611469528043398074101868493775807 49\
        76005849350096246163153170207154803327764403891235495339 55\
        13648983858679308486984128814034238436292458209871066374 16\
        37879435340626899282625842893425936879209207992728130466 47\
        01788498353030447929676766933686440526121344020185610194 72\
        5545667291311399759128553738536005398098974277804114657

> convert(n,hex);

  673FE30AF9856A3966707223394C4FDC761EFFFC304414EF3C13295873AE93A3\
        66499D8606CDB642D57247B38AD6DD270B86D9CFEB2B763001770F1253\
        86D8913B62C9298CF933CC48B77C991578D76FDB78FBDE4682D08331BC\
        FADFDDEF96FD9340BD3E83D6420719C37BE88E9566CBB8A2A62099D86D\
        FD67D74AC877F13B0E9E4BB19603005A85C1655791D2CF0A8C3BF28E2A\
        922F4B188850C85A0DC1CE08D832FF5CBB7FDEA3DC367CBB4BB19AE320\
        773B0F5F2F5A8333B78BD28D0242E1

> g:=2;

                                g := 2

> convert(g,hex);

                                  2

> A:=2^239;

  A := 8834235323891921647916487503714592579137419484378094790608\
        3100646309888

> convert(A,hex);

     800000000000000000000000000000000000000000000000000000000000

> S:=2^159;

        S := 730750818665451459101842416358141509827966271488

> convert(S,hex);

               8000000000000000000000000000000000000000

> B:=2^15;

                              B := 32768

> convert(B,hex);

                                 8000

> s:=rand(S)();

         s := 356202518350650557724073146581318210568977468 59

> convert(s,hex);

               63D447004789E2CC3BA125E0AFE82092C9687AB

> Ipub:=g &^ (-s) mod n;

  Ipub := 6359487736562223488056833032821216647730832078591663486 9\
        45895463453330220981505151783672188003788766934288367430 52\
        56317555457200816313321041317664204253317650692026922712 09\
```

```
          4589546345333022098150515178367218800378876693428836743052\
          5631755545720081631332104131766420425331765069202692271209\
          1036978247237094052676825101926413551477691184110173410536\
          6104180792447620721540049264316377911821103187391173814010\
          4478106658708533974941146120293012181033568185097927780386\
          0497721001235800617281724152658353403410696352110168175858\
          397892534879436628368364964239768630223658977481765566 2677

> convert(Ipub,hex);

   438B57CCA73AD273F2B05AC45471FB3FC1C83FC9DD3273F3A61493F4F38D5A1A\
          F2873E2760D0F2226A6A716CE7F947DA138A036E8ED625A2EC4BF4C666\
          45190B8EF201C261DA050FE8C2791EC7C79CF7C9AB0108390165971250\
          F729E11740B6ADA131F3BD28510845EEB2784583869E05941556B12054\
          0D4D067546EC9D950A06529E81C47402B9EA3CBCB80E5D4E34D713BBE6\
          54498C87BC9204A66EC0BD199A88F7231378DB503204F82F8CF423A894\
          ACF9C376D5329512036 6550EC62855

> r:=rand(A)();

   r := 1957985440535954301632598070686917520699614846508473931674 3\
          7407082494718

> convert(r,hex);

      1C5E9256EED9663A380722DF413D36E1A14E79C2520F44945DF47927BAFE

> g&^r mod n;

   5673702143156668336795415632017583560347247288573482040456624861\
          4285371028392797549664244051800806931654569932317265599423\
          8439516056824077106032212263556926418687999212562001987222\
          2637981799727936238764349748489352954494816999568207744658\
          0421463226012018942345355034785155002847867478176490723409\
          6279723352510011758067995366608498562622664875735337279307\
          5524820406467401013919418072358772105840786702686774171325\
          34449288290178178186954765726306749409146360427342

> convert(%,hex);

   3C42B4032CE04AC011721CA2FB1502CCB7C5B13B7C753475DB06F91044698955\
          167FF533841B6AC7F8552E7EB8BFDAC19A547E10EB0381D8235FD257DD\
          C050233056E7CAC3C9477A4EAD7093D91F2EAF08DBB844B7C4186FEAF9\
          0B850DAE30E826770C55ED4E33ACAA5E4FBEED2BD9A2414A3556662BB8\
          F037CFF924297F4373F45AA631AAA37878CF9EB46E73D882752F3746AE\
          6E9FED153553C449A753584317ECB0D042EE82B7A18258231BDC872401\
          FE73CC04DFF88D7C03C5E28485F34E

  > c:=rand(B)();

                                c := 10072

  > convert(c,hex);

                                   2758

  > y:=r+c*s;

    y := 19579854405359543016361857424517452731170117112408409686912\
          2481188858566

  > convert(y,hex);

       1C5E9256EED9663A38FC9D5BD86D2070D2831A84F60FD1E076E6BB916AC6

  > (g&^y mod n)*(Ipub&^c mod n) mod n;

    56737021431566683367954156320175835603472472885734820404566624861\
          42853710283927975496642440518008069316545699323172655994423\
          84395160568240771060322122635569264186879992125620019872222\
          26379817997279362387643497484893529544948169995682077446658\
          04214632260120189423453550347851550028478674781764907234409\
          62797233525100117580679953666084985626226648757353372793307\
          55248204064674010139194180723587721058407867026867741713325\
          344492882901781781869547657263067494091463604273422
```

```
> convert(%,hex);

  3C42B4032CE04AC011721CA2FB1502CCB7C5B13B7C753475DB06F91044698955\
       167FF533841B6AC7F8552E7EB8BFDAC19A547E10EB0381D8235FD257DD\
       C050233056E7CAC3C9477A4EAD7093D91F2EAF08DBB844B7C4186FEAF9\
       0B850DAE30E826770C55ED4E33ACAA5E4FBEED2BD9A2414A3556662BB8\
       F037CFF924297F4373F45AA631AAA37878CF9EB46E73D882752F3746AE\
       6E9FED153553C449A753584317ECB0D042EE82B7A18258231BDC872401\
       FE73CC04DFF88D7C03C5E28485F34E
```

Book IV

Selected research papers

About the NESSIE submission "Using the general next bit predictor like an evaluation criteria"

by Markus Dichtl and Pascale Serf [*]

Abstract. In this paper, we are dealing with a testing methodology based on machine learning, submitted to NESSIE by J.C. Hernandez, J.M. Sierra, C. Mex-Perera, D. Borrajo, A. Ribagorda, and P. Isasi. We show that their evaluation method suggested for measuring the unpredictability of pseudorandomly generated bit streams only works for linear feedback shift registers with a very small number of tabs. Contrary to what they claim, it neither works for LFSRs with a cryptologically reasonable number of tabs, nor for the truncated linear congruential generator.
Hence, this method is considered not suitable for the evaluation of cryptographic algorithms.

Keywords. cryptography, cryptology, machine learning, learning algorithm C4.5, artificial intelligence, pseudorandomly generated bit stream, linear feedback shift register, truncated linear congruential generator, predictability, evaluation criteria

1 Summary of the Suggested Evaluation Method

J.C. Hernandez, J.M. Sierra, C. Mex-Perera, D. Borrajo, A. Ribagorda, and P. Isasi submitted to NESSIE a testing methodology for measuring the unpredictability of a pseudorandomly generated key stream. They propose to use machine learning techniques: frames of a fixed length from the bit stream together with the bit immediately following the frame are used as learning data for a classification algorithm. The idea is to assign the frames to two classes, namely those followed by 0 bits and those followed by 1 bits. A learning algorithm is supposed to learn this classification from training data. The submitters suggest to use the learning algorithm C4.5 by J.R. Quinlan, which seems to be used quite often for machine learning. C4.5 builds a decision tree from the learning data and applies some heuristic simplifications to the decision tree.

[*] Siemens AG, Corporate Technology, München, Germany
{Markus.Dichtl,Pascale.Serf}@mchp.siemens.de
August 22, 2001

2 Examples of the Application of the Evaluation Method and their Reproducibility

2.1 Linear Feedback Shift Register

As an example, the submitters examine the linear feedback shift register with primitive polynomial $x^{15} + x + 1$. The results are very surprising. Therefore, we tried to reproduce them by applying the learning tool, exactly as the submitters describe in their submission. We can confirm that the learning algorithm indeed finds out that the output bit is the XOR of 2 earlier bits. The submitters found 100% accuracy only for a framelength of 20 bits, and an accuracy of 97% for a framelength of 15. We obtained 100% accuracy for both framelengths.

The result of the submitters for a framelength of 10 is very surprising. It is theoretically impossible to make relevant predictions in this case, because the bits whose XOR determines the output bit are not contained in the frame, and from the theory of linear feedback shift registers, it is well known that all 10-tuples from a LFSR of length 15 occur about the same number of times. Nevertheless, the submitters claim to be able to make predictions with an accuracy of 95%. In our experiments, the success rate achieved was very close to 50%.

Since the problem to find out that the result bit is the XOR of 2 input bits is rather easy, we also tested cryptographically better LFSRs with more tabs. The length of the LFSRs was 15, and we used 5000 frames of length 20 for learning. The success rate was very close to 50% with 8 tabs. With 6 and 4 tabs, the success probabilities were better than 50%.

Why does the learning algorithm not succeed with many tabs? When we solve linear equations, the problem is easy, no matter what the number of tabs is. The problem can be solved with just 15 frames. However, the learning algorithm does not know about the linear structure of the problem, it has to find that out for itself. Unfortunately, decision trees are not very well suited to represent XORs of many variables. The size of the decision tree grows exponentially with the number of tabs. When few tabs are used, the heuristic simplification algorithms used by C4.5 prefer the simpler decision tree representing the linear relationship to the many other, more complicated decision trees also in concordance with the data. When the number of tabs grows, the exponential growth of the correct decision tree makes it less and less attractive for the heuristic simplification.

2.2 Truncated Linear Congruential Generator

The submitters also claim to obtain results with a truncated linear congruential generator. We also tried to verify these results. We used a framelength of 80. The paper gave an accuracy of 100% for the next bit prediction, but our result was only 50.1%. Now one might think that we just did not apply the method correctly, but finally we found out how to reproduce the results from the paper. We just had to implement the truncated linear congruential generator wrongly, and then we almost got the results from the paper. Our error rates and the error rates of the submitters only differed by 0.1%. This small difference might be explained by some imprecision in the description of the experiment of the submitters. They claim to have generated 100000 bits, and to have used 50000 frames for learning and 50000 frames for testing. However, it takes 100080 bits to obtain 100000 frames. In our experiment, we used exactly 100000 frames.

The generator is defined by the iteration $x := (a * x) \bmod m$. The problem is that the multiplication $a * x$ of 32-bit-integers must be done with a precision of 64 bits (or maybe some bits less, but 32 bits is definitely not enough). When this is implemented using standard C arithmetic, a reduction modulo 2^{32} is done implicitly before the reduction modulo m. And apparently this happened to the submitters. The consequence of this mistake is that the generator enters a cycle of length 1651. There is a pre-period of length 18064. Of course, the machine learning algorithm just learns the cycle, and is then able to predict the next bits. The submitters give a strange explanation why their algorithm gives better predictions for the test data than for the learning data. We think to have a better explanation: the data used in the learning phase contained the pre-period, which had not been learned perfectly; the test data contained only data from the cycle, which had been learned perfectly.

3 Conclusion

The results of our experiments indicate that the suggested evaluation method for pseudorandom bit sequences does not work as the submitters claim. The only case for which the claim of the submitters could be verified are linear feedback shift registers with a very small numbers of tabs. These LFSRs are much too simple to be cryptologically relevant. For the truncated linear congruential generator, the approach of the submitters did not work at all.

Because of these shortcomings the submission is considered not suitable for the evaluation of cryptographic algorithms.

Constructive Methods for the Generation of Prime Numbers

by Marc Joye and Pascal Paillier [*]

Abstract. The generation of prime numbers underlies the use of most public-key cryptosystems, essentially as a primitive needed for the creation of RSA key pairs. Surprisingly enough, despite decades of intense mathematical studies on primality testing and an observed progressive intensification of cryptography, prime number generation algorithms remain scarcely investigated and most real-life implementations are of dramatically poor performance.

This paper shows simple techniques that substantially improve all algorithms previously suggested or extend their capabilities. We derive fast implementations on appropriately equipped portable devices like smart-cards embedding any kind of cryptographic coprocessor. This allows on-board generation of RSA keys featuring a very attractive (average) processing time.

Our motivation here is to help transferring this task from terminals where this operation usually took place until now, to portable devices themselves in near future for more confidence, security, and compliance with network-scaled distributed protocols such as electronic cash or mobile commerce.

1 Introduction

Undoubtedly, the lack of *efficient* prime number generators severely restricts the development of public-key cryptography in embedded environments. Several algorithms that generate prime numbers do exist, some of them being well-known and popular [121,122,168,430], but most of them are hardly adapted to the computational context of portable devices like smart cards or PDA's, where memory capabilities and processing power are somewhat limited. A noticeable exception is found in a recent heuristic algorithm by Joye, Paillier and Vaudenay [332].

This paper improves their algorithm in multiple directions. First, we give a more general description with extended parameter choices that fit any given (crypto-)processor architecture. Second, we present new techniques that speed up the entire process and reduce the standard statistical deviation, especially in the generation of so-called units. Third, we consider the issue of length extendibil-

[*] Security Technology Department, Gemplus, France
`http://www.gemplus.com/`
`{marc.joye,pascal.paillier}@gemplus.com`

This is an extended version of our results presented at the 2nd Open NESSIE Workshop (Egham, UK, September 12–13, 2001).

ity, that is, algorithmic solutions for obtaining primes of arbitrary, dynamically chosen bit-size.

The prime number generation algorithms we consider here find their main application in the generation of RSA keys on embedded platforms. This context of use implies the additional condition on a prime q being generated, that $q-1$ be coprime to the public RSA exponent e. We show how our algorithms may automatically fulfil this latter condition at negligible cost, at least for small or smooth values of e. Finally, as an illustrative application of our techniques, we show how to efficiently generate a random safe (resp. quasi-safe) RSA modulus as the product of two safe (resp. quasi-safe) primes. This answers a problem left open in [332].

The rest of the paper is organised as follows. In Section 2, we review the RSA primitive, both in standard and CRT modes, thereby fixing notations. The following sections constitute the core of the paper as we show how to generate an RSA modulus of prescribed length, for a given public exponent. In Section 7, we explain how to generate a safe RSA modulus in a more efficient way. Finally, we conclude in Section 8.

2 The RSA Primitive

RSA is certainly the most widely used cryptosystem today. We give hereafter a short description which allows us to introduce notations, referring the reader to the original paper [543] or any textbook in cryptography (*e.g.* [441]) for further detail.

Let $N = pq$ be the product of two large primes. We let e and d denote a pair of matching public and private exponents according to

$$e\,d \equiv 1 \pmod{\lambda(N)}\ ,$$

with $\gcd(e, \lambda(N)) = 1$ and λ being Carmichæl's function. As $N = pq$, we have $\lambda(N) = \operatorname{lcm}(p-1, q-1)$. Given $x < N$, the public operation (*e.g.* message encryption or signature verification) consists in raising x to the e-th power modulo N, *i.e.* in computing $y = x^e \bmod N$. Then, given y, the corresponding private operation (*e.g.* decryption of a ciphertext or signature generation) computes $y^d \bmod N$. From the definition of e and d, we obviously have that $y^d \equiv x \pmod{N}$. The private operation can be carried out at higher speed through Chinese remaindering (CRT mode [167, 202]). Computations are independently performed modulo p and q and then recombined. In this case, private parameters are (p, q, d_p, d_q, i_q) with

$$\begin{cases} d_p = d \bmod (p-1)\ , \\ d_q = d \bmod (q-1)\ , \text{ and} \\ i_q = q^{-1} \bmod p\ . \end{cases}$$

We then obtain $y^d \bmod n$ as

$$\operatorname{CRT}(x_p, x_q) = x_q + q\,[i_q(x_p - x_q) \bmod p]\ ,$$

where $x_p = y^{d_p} \bmod p$ and $x_q = y^{d_q} \bmod q$. We expect a theoretical speed-up factor close to 4 (see [167]), compared to the standard, non-CRT mode.

Thus, an RSA modulus $N = pq$ is the product of two large prime numbers p and q. If n denotes the bit-size of N then p must lie in the range $\left[\left\lceil 2^{n_0-1/2}\right\rceil, 2^{n_0}\right]$ and q must lie in the range $\left[\left\lceil 2^{n-n_0-1/2}\right\rceil, 2^{n-n_0}\right]$ for some $1 < n_0 < n$, so that $2^{n-1} < N = pq < 2^n$. For security reasons, so-called balanced moduli are generally preferred, which means $n_0 = \lceil n/2 \rceil$.

The next section describes an efficient (trial-division free, as opposed to [109, 122, 168, 430]) algorithm for producing a prime q uniformly distributed in some given interval $[q_{\min}, q_{\max}]$, or a sub-interval thereof, $q_{\min}$ and $q_{\max}$ being two arbitrarily chosen integers and $q_{\min} < q_{\max}$. Our proposal actually consists in a pair of algorithms: the prime generation algorithm itself and an algorithm for generating invertible elements, also called *units* [332]. We assume that a random number generator is available, and that some fast (pseudo) primality (resp. compositeness [26, 115, 372, 450, 521, 540, 590]) testing function T is provided as well. As this paper focuses on prime number generation and not on primality testing, we refer the reader to the excellent survey in [441, Chapter 4].

3 Generic Prime Number Generation

Let $0 < \varepsilon \leq 1$ denote a quality parameter (a typical value for ε is 10^{-3}). Our setup phase requires to choose a product of (distinct) primes $\mathit{\Pi} = \prod_i p_i$ such that there exist integers t, v, w satisfying

(P1) $1 - \varepsilon < \dfrac{w\mathit{\Pi} - 1}{q_{\max} - q_{\min}} \leq 1$;
(P2) $v\mathit{\Pi} + t \geq q_{\min}$;
(P3) $(v + w)\mathit{\Pi} + t - 1 \leq q_{\max}$;
(P4) the ratio $\phi(\mathit{\Pi})/\mathit{\Pi}$ is as small as possible.

The primes output by our algorithm lie, in fact, in the sub-interval $[v\mathit{\Pi} + t, (v + w)\mathit{\Pi} + t - 1] \subseteq [q_{\min}, q_{\max}]$ as illustrated on Fig. 3.

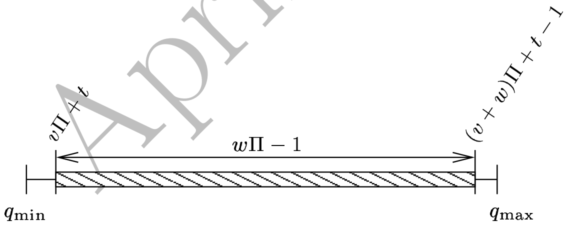


Fig. 64. ε-approximated output domain

The error in the approximation is captured by the value of ε meaning that a smaller value for ε gives better results (cf. Property (P1)). The minimality of the ratio $\phi(\Pi)/\Pi$ in Property (P4) ensures that Π contains a maximum number of distinct primes and that these primes are as small as possible. Given any triplet $(q_{\min}, q_{\max}, \varepsilon)$, computing the tuple (Π, v, w, t) that best matches Properties (P1)-(P4) is easy. We do not describe that pre-computation stage explicitly here due to lack of space.

```
Parameters: $l = v\Pi$, $m = w\Pi$, $t$, and $a \in (\mathbb{Z}/m\mathbb{Z})^* \setminus \{1\}$
Output:     a random prime $q \in [q_{\min}, q_{\max}]$

1. Randomly choose $k \in (\mathbb{Z}/m\mathbb{Z})^*$
2. Set $q \leftarrow [(k - t) \bmod m] + t + l$
3. If $(\mathsf{T}(q) = \texttt{false})$ then
     a) Set $k \leftarrow ak \pmod{m}$
     b) Go to Step 2
4. Output $q$
```

Fig. 65. Generic prime generation algorithm for $q \in [q_{\min}, q_{\max}]$.

We now proceed to describe our prime number generation algorithm in its most generic version, as depicted on Fig. 65. The first step requires the random selection of an integer $k \in (\mathbb{Z}/m\mathbb{Z})^*$, and we show how to do that efficiently later in the paper. At this stage, it is worthwhile noticing that if a and k both belong to $(\mathbb{Z}/m\mathbb{Z})^*$ so does their product $ak \bmod m$, since $(\mathbb{Z}/m\mathbb{Z})^*$ is a (multiplicative) group. Therefore, throughout the above algorithm, k remains coprime to m and also to Π — remember that Π contains a large number of prime factors by Property (P4). This, in turn, implies that q is coprime to Π as $q \equiv [(k - t) \bmod m] + t + l \equiv k \pmod{\Pi}$ and $k \in (\mathbb{Z}/\Pi\mathbb{Z})^*$. Hence, this technique ensures *built-in* coprimality of our prime candidate q with a large set of small prime numbers. Consequently, the probability under which q is prime at Step 3 is in fact quite high. When q is found to be composite, a new candidate is derived by "recycling" q in a way that preserves its coprimality to Π. An interesting feature is that each and every prime lying in the prescribed range (and therefore each and every prime in $[q_{\min}, q_{\max}]$ except an ε fraction of it, *cf.* Fig. 3) can be selected by our technique. Finally, note that the order of a in $(\mathbb{Z}/m\mathbb{Z})^*$ should be made as large as possible in order to prevent the search sequence, *i.e.* the list of successive candidates, from falling into a periodic, prime-free infinite loop.[1]

The previous algorithm is actually very general and can be adapted in numerous ways, depending on hardware capabilities of the targeted processor architecture. Public-key crypto-processors generally allow super-fast (modular) additions, subtractions and multiplications over large integers, and this renders other types of computations comparatively prohibitive, unless specific hardware is integrated to support these. In the sequel, we give a possible implementation to illustrate

[1] This event may occur with very small probability, though, unless the order of a in $(\mathbb{Z}/m\mathbb{Z})^*$ happens to be trivially small.

this, in which we attempt to increase our algorithm's performance to its uppermost level while running on a general-purpose crypto-processor. Other choices of parameters may lead to better results on specific platforms.

3.1 An Implementation Example

The optimal value for t is certainly $t = 0$. Moreover, it is advantageous to choose the constant a so that performing a multiplication by a modulo m turns out to be a somewhat trivial operation. In the end, the best possible choice is $a = 2$, because multiplying by 2 then reduces to a single bit shift or addition, possibly followed by a subtraction. Unfortunately, 2 must belong to $(\mathbb{Z}/m\mathbb{Z})^*$ and owing to Property (P4), 2 is a factor of Π and so of m, a contradiction. A simple trick here consists in choosing m odd (so that $2 \in (\mathbb{Z}/m\mathbb{Z})^*$) and in slightly modifying the above framework in order to ensure that a prime candidate q be always odd. We require $\Pi = \prod_i p_i$ (with $p_i \neq 2$) and integers v and w (w odd) satisfying:

(P2') $v\Pi + 1 \geq q_{\min}$;
(P3') $(v + w)\Pi - 1 \leq q_{\max}$.

Another improvement consists in letting the value of l vary as a random multiple of Π instead of fixing it. This allows to compute modulo Π instead of modulo m, resulting in faster arithmetic. Putting it all together, we obtain the algorithm shown on Fig. 66.

```
Parameters: $\Pi$ odd, $v$, $w$, $l$
Output:     a random prime $q \in [q_{\min}, q_{\max}]$

 1. Randomly choose $k \in (\mathbb{Z}/\Pi\mathbb{Z})^*$
 2. Randomly choose $j \in \{v, \ldots, v + w - 1\}$
 3. Define $l \leftarrow j\,\Pi$
 4. Set $q \leftarrow k + l$
 5. If ($q$ even) then $q \leftarrow \Pi - k + l$
 6. If ($\mathsf{T}(q) = $ false) then
      a) Set $k \leftarrow 2k \pmod{\Pi}$
      b) Go to Step 4
 7. Output $q$
```

Fig. 66. Faster prime generation algorithm

Note that if $k+l$ is even then $\Pi-k+l$ is odd since $\Pi-k+l \equiv \Pi+(k+l) \equiv \Pi \equiv 1 \pmod 2$. Hence, as before, any candidate q belonging to our search sequence is coprime to 2Π: we get $\gcd(q, 2) = 1$ as q is odd, and $\gcd(q, \Pi) = 1$ as $q \equiv \pm k \pmod{\Pi}$ and $\pm k \in (\mathbb{Z}/\Pi\mathbb{Z})^*$.

A complexity analysis is easily driven from the work of [332]. The expected number of calls to T, *i.e.* the number of primality or compositeness tests required in average, heuristically amounts to $\ln 2 \cdot |q_{\max}| \cdot \frac{\phi(\Pi)}{\Pi}$. Naturally the exact, concrete efficiency of our implementation also depends on hardware-related features. In any case, in practice, a spectacular execution speed-up[2] is generally observed in comparison with usual, incremental and trial-division-based prime number generators.

4 Generation of Units

All prime generation algorithms presented in this paper (as well as those in [332]) require the random selection of some element $k \in (\mathbb{Z}/m\mathbb{Z})^*$. This section provides an algorithm that efficiently produces such an element with uniform output distribution. We base our design on the next two propositions.

Proposition 4.1. *For all* $k \in \mathbb{Z}/m\mathbb{Z}$, $k \in (\mathbb{Z}/m\mathbb{Z})^*$ *if and only if* $k^{\lambda(m)} \equiv 1 \pmod{m}$.

Proof. We have $k \in (\mathbb{Z}/m\mathbb{Z})^*$ if and only if, for all primes $p_i \mid m$, $\gcd(k, p_i) = 1$ meaning that $k^{p_i - 1} \equiv 1 \pmod{p_i}$ so that $k^{\lambda(m)} \equiv 1 \pmod{m}$ by Chinese remaindering. □

Proposition 4.2. *For all* k *and* $r \in \mathbb{Z}/m\mathbb{Z}$ *s.t.* $\gcd(r, k, m) = 1$, *we have*

$$[k + r(1 - k^{\lambda(m)})] \in (\mathbb{Z}/m\mathbb{Z})^* \quad .$$

Proof. Let $\prod_i {p_i}^{\delta_i}$ denote the prime factorisation of m. Define $\omega(k, r) := [k + r(1 - k^{\lambda(m)})] \in \mathbb{Z}/m\mathbb{Z}$. Let p_i be a prime factor of m. Suppose that $p_i \mid k$ then $\omega(k, r) \equiv r \not\equiv 0 \pmod{p_i}$ since $\gcd(r, p_i)$ divides $\gcd(r, \gcd(k, m)) = \gcd(r, k, m) = 1$. Suppose now that $p_i \nmid k$ then $k^{\lambda(m)} \equiv 1 \pmod{p_i}$ and so $\omega(k, r) \equiv k \not\equiv 0 \pmod{p_i}$. Therefore for all primes $p_i \mid m$, we have $\omega(k, r) \not\equiv 0 \pmod{p_i}$ and thus $\omega(k, r) \not\equiv 0 \pmod{{p_i}^{\delta_i}}$, which, invoking Chinese remaindering, concludes the proof. □

We benefit from these facts by devising the unit generation algorithm shown on Fig. 67. This algorithm is self-correcting in the following sense: as soon as k is relatively prime to some factor of m, it remains coprime to this factor after the updating step $k \leftarrow k + rU$. This is due to Proposition 4.2. What happens in simple words is that, viewing k as the vector of its residues $k \bmod p_i^{\delta_i}$ for all $p_i^{\delta_i} \mid m$ (*i.e.* the RNS representation of k based on m, see [202]), non-invertible coordinates of k are continuously re-randomised until invertibility is reached for all of them. In this respect, it is not difficult to prove that the output distribution is uniform provided that the random number generator has a uniform output distribution. Besides, the cost of this unit generator can be formally evaluated and roughly amounts to a couple of exponentiations[3] modulo m with exponent

[2] which usually amounts to one order of magnitude.

[3] The running time happens to be rather insensitive to the choice of Π's prime factors, provided that the first small primes are included.

```
Parameters: $m$ and $\lambda(m)$
Output:     a random unit $k \in (\mathbb{Z}/m\mathbb{Z})^*$

1. Randomly choose $k \in [1, m)$
2. Set $U \leftarrow (1 - k^{\lambda(m)}) \bmod m$
3. If $(U \neq 0)$ then
    a) Choose a random $r \in [1, m)$
    b) Set $k \leftarrow k + rU \pmod{m}$
    c) Go to Step 2
4. Output $k$
```

Fig. 67. Our unit generation algorithm

$\lambda(m)$, in average (again, we do not include these results here due to the lack of space). Note also that all computations fall into the range of operations easily and efficiently performed by any crypto-processor.

5 Length Extendibility

So far, our implementation parameters are Π, a, the triplet (v, w, t) and $\lambda(m)$ with $m = v\Pi$. These values are chosen once and for all and heavily depend on $q_{\min} = \lceil 2^{n_0 - 1/2} \rceil$ and $q_{\max} = 2^{n_0}$, if n_0 denotes the bit-size of prime numbers being generated. Now, the feature we desire here (and this is motivated by code size limitations embedded platforms usually have to work with), consists in the ability to use the parameters sized for n_0 to generate primes numbers of bit-size $n \neq n_0$. A performance loss is acceptable compared to the situation when parameters are generated for both lengths.

We propose an implementation solving that problem for any $n \geq n_0$, provided that a was chosen odd and that arithmetic computations can still be carried out over n-bit numbers on the processor taken into consideration. It is an extended version of the algorithm depicted on Fig. 65. We exploit the somewhat obvious, following facts:

1. letting $q_{\max}(x) = 2^x$ and $q_{\min}(x) = \lceil 2^{x-1/2} \rceil$, we have of course $q_{\max}(n) = q_{\max}(n_0) 2^{n-n_0}$ and $q_{\min}(n) \approx q_{\min}(n_0) 2^{n-n_0}$,
2. given $\Pi(n_0)$ chosen as in Section 3, we take
$$\begin{cases} \Pi(n) = \Pi(n_0)\ , \\ v(n) = v(n_0) 2^{n-n_0}\ , \\ w(n) = w(n_0) 2^{n-n_0}, \text{ and} \\ t(n) = t(n_0) 2^{n-n_0}\ , \end{cases}$$
hence $l(n) = l(n_0) 2^{n-n_0}$ and $m(n) = m(n_0) 2^{n-n_0}$,
3. $a(n) = a(n_0)$, hence $a(n) \in (\mathbb{Z}/m(n)\mathbb{Z})^*$ since $a(n_0)$ is taken odd,

4. given $\boldsymbol{\lambda}(n_0) = \lambda(m(n_0))$, it is easy to see that denoting $\boldsymbol{\lambda}(n) = \boldsymbol{\lambda}(n_0)2^{n-n_0}$, we have again $\boldsymbol{\lambda}(n) = \lambda(m(n))$, or at least $\boldsymbol{\lambda}(n) \propto \lambda(m(n))$ which is a sufficient condition for the unit generation algorithm to be effective.

These transformations happen to preserve Properties (P1), (P2) and (P3) we required earlier, with $\varepsilon(n) = \varepsilon(n_0)$. The reader easily sees that all parameters for some bit-size n may, as a direct consequence, be replaced by the respective parameters computed for n_0 multiplied by 2^{n-n_0}, except for $\Pi(n) = \Pi(n_0)$. By performing this replacement, we just accept to live with under-optimised performances because the ratio $\phi(\Pi(n))/\Pi(n)$ will not be chosen minimal. Still, our algorithm will output n-bit primes in a correct manner, for any dynamic choice of $n \geq n_0$, with a granularity of one single bit. Our extended algorithm is depicted on Fig. 68.

```
Parameters: $l(n_0) = v(n_0)\Pi(n_0)$, $m(n_0) = w(n_0)\Pi(n_0)$,
            $t(n_0)$, $a(n_0) \in (\mathbb{Z}/m(n_0)\mathbb{Z})^* \setminus \{1\}$, $n_0$
Input:      bit-size $n \geq n_0$
Output:     a random prime $q \in [q_{\min}(n), q_{\max}(n)]$

1. Set $m \leftarrow m(n_0)2^{n-n_0}$, $t \leftarrow t(n_0)2^{n-n_0}$ and $l \leftarrow l(n_0)2^{n-n_0}$
2. Randomly choose $k \in (\mathbb{Z}/m\mathbb{Z})^*$
3. Set $q \leftarrow [(k-t) \bmod m] + t + l$
4. If ($\mathsf{T}(q) =$ false) then
   a) Set $k \leftarrow a(n_0) \cdot k \pmod{m}$
   b) Go to Step 3
5. Output $q$
```

Fig. 68. Our scalable prime generation algorithm

In Step 2, the random unit generation is executed with parameters $m(n_0)2^{n-n_0}$ and $\boldsymbol{\lambda}(n_0)2^{n-n_0}$ instead of $m(n_0)$ and $\boldsymbol{\lambda}(n_0)$. This does not affect the algorithm whatsoever. Another observation is that the order of $a(n)$ modulo $m(n)$ is necessarily larger than (or equal to) the order of $a(n_0)$ modulo $m(n_0)$. It is therefore large enough for all our choices of n provided that $a(n_0)$ was correctly chosen in the first place.

6 Public Exponent

This section deals with the generation of a prime q such that the condition $\gcd(e, q-1) = 1$ is automatically satisfied. Let $e = \prod_i {e_i}^{\nu_i}$ denote the prime factorisation of our public exponent e. Because the RSA primitive induces a permutation (*i.e.* $\gcd(e, \lambda(N)) = 1$), it turns out that RSA primes p and q must be such that $\gcd(e_i, p-1) = 1$ and $\gcd(e_i, q-1) = 1$ for each prime e_i dividing e.

First, let us assume that $e_i \mid \Pi$ for all i. This happens in the most popular scenario where e is some small prime (like 3 or 17) or when e is chosen smooth.

Let α be an integer coprime to Π and of maximal order modulo e_i for all e_i dividing e, *i.e.*

$$\gcd(\alpha, \Pi) = 1 \text{ and } \operatorname{order}(\alpha \bmod e_i, e_i) = e_i - 1 \text{ for each } e_i \mid e \ .$$

In practice, the choice of a value for α may be done easily using Chinese remaindering. Note that for such an α, we get that $\operatorname{order}(\alpha, e_i)$ is simultaneously even for all prime factors $\{e_i\}_i$. We define $e^+ = \gcd(e, \Pi) = \prod_i e_i$ and denote by k_0 the initial value for k that the unit generation algorithm of Fig. 67 gets by invoking the random number generator in Step 1. It is easily seen that if we force

$$k_0 \equiv \alpha \pmod{e^+} \ , \tag{1}$$

then the unit k eventually output by the algorithm will also verify that $k \equiv \alpha \pmod{e^+}$. This is due to the algorithm's self-correctness. We then adapt the generic prime generation algorithm by choosing $a = \alpha^2$. By doing this, every candidate q generated by the sequence will satisfy

$$q \equiv \alpha^{2j+1} \pmod{e^+} \ ,$$

for some integer j, because $e^+ \mid \Pi$. So we can never have $q \equiv 1 \pmod{e_i}$ since α is of even order modulo e_i and q is an odd power of α. Consequently, $q \not\equiv 1 \pmod{e_i}$ for all i, which implies $\gcd(q-1, e) = 1$.

So our technique works when $e_i \mid \Pi$ for all i, that is, when e has only small prime factors. To deal with cases when $e_i \nmid \Pi$ for some $e_i \mid e$, we face the following options:

- either e is a prime number itself (like Fermat's fourth prime $2^{16} + 1$) and we add the verification step
$$q - 1 \overset{?}{\equiv} 0 \pmod{e}$$
before or after the primality test T is applied;
- or e is not prime but its factorization is known. We already know that $q \not\equiv 1 \pmod{e_i}$ when $e_i \mid \Pi$, so we have to ensure that the same holds when $e_i \nmid \Pi$. To do this, we simply check if $q - 1 \equiv 0 \pmod{e_i}$ for all prime factors $e_i \nmid \Pi$, or equivalently (but preferably) invoke Proposition 4.1 and make sure that
$$(q-1)^{\lambda(e^-)} \not\equiv 1 \pmod{e^-} \ ,$$
where $e^- = \prod_i e_i$ for all $e_i \nmid \Pi$.

In both cases, unfortunately, adding at least one complementary test to the implementation cannot be avoided. Finally, forcing $k_0 \equiv \alpha \pmod{e^+}$ in Eq. (1) is easily done by picking a random number r and setting $k_0 = \alpha + er \pmod{\Pi}$.

7 Generating Safe and Quasi-Safe Primes

We now show how to apply our generic techniques to the specific case of generating safe primes. A similar algorithm for quasi-safe primes is obtained in the same spirit with minor variations.

7.1 Safe Primes

We start by listing the properties fulfilled by a number q generated by our algorithm:

1. q is an n-bit number for a given bit-size parameter n,
2. q belongs to $[q_{\min}, q_{\max}]$ where $q_{\min} = \lceil 2^{n-1/2} \rceil$ and $q_{\max} = 2^n$,
3. q is a prime number,
4. $(q-1)/2$ is a prime number.

Using earlier notations, parameters $(\Pi, l = v\Pi, t, m = w\Pi, \lambda(m))$ are precomputed as in Section 3 so that $m-1$ lower-approximates the window width $q_{\max} - q_{\min}$ up to a fixed precision, say $\varepsilon = 10^{-3}$. We assume that we dispose of a random number generator over $[1, \Pi)$, a fast primality testing function $\mathsf{T}()$ mapping integers to $\{\mathtt{true}, \mathtt{false}\}$, and an algorithm for generating invertible elements modulo m as described in Section 4.

All the point here resides in the way the search sequence is carried out. It should ideally verify that each and every candidate $q^{(i)}$ be such that both $q^{(i)}$ and $(q^{(i)}-1)/2$ are coprime to Π. It is somewhat easy to guarantee that for $q^{(i)}$ by ensuring (like in previous sections) that

$$q^{(i)} \equiv a^i\, k \pmod{\Pi}$$

for some initially (uniformly) generated $k \in (\mathbb{Z}/m\mathbb{Z})^*$ and $a \in (\mathbb{Z}/m\mathbb{Z})^*$. However, the later constraint on $(q^{(i)}-1)/2$ is a bit more delicate. Our need here is to ensure that for each prime divisor $p \neq 2$ of Π,

$$q^{(i)} \not\equiv 1 \pmod{p}\ , \tag{2}$$

and $q^{(i)} \equiv 3 \pmod{4}$ if 2 divides Π.

To fulfil the condition given by Eq. (2), we will make sure that $q^{(i)} \mod p$ just cannot be an element of $\mathrm{QR}(p)$, the subgroup of quadratic residues mod p. Doing so, we ensure that $q^{(i)} \not\equiv 1 \pmod{p}$ whenever $p \mid \Pi$. We proceed in the following manner. First, the constant a is chosen in $\mathrm{QR}(m)$. Then, we choose once for all a parameter $u \in (\mathbb{Z}/m\mathbb{Z})^*$ such that

$$\text{for all (odd) primes } p \text{ dividing } \Pi, \text{ we have } u \notin \mathrm{QR}(p)\ .$$

From there on, the initial unit k is chosen as $k = u \cdot \chi^2 \mod m$ for some random $\chi \in (\mathbb{Z}/m\mathbb{Z})^*$ and we set as before

$$q^{(i)} = [(a^i k - t) \mod m] + t + l\ .$$

Then, for each and every odd $p \mid \Pi$, $q^{(i)} \equiv a^i u \chi^2 \mod p$ has a Legendre symbol different from 1, and consequently $q^{(i)} - 1$ cannot be 0 modulo p, *i.e.* $q^{(i)} - 1$ is coprime to (the odd part of) Π. When Π is even, we have to make sure, in addition to the above, that $q^{(i)} \equiv 3 \mod 4$ meaning that the last two bits in the binary representation of $q^{(i)}$ are forced to $\ldots 11_2$, thereby guaranteeing that $(q^{(i)}-1)/2$ is an odd number and consequently that $(q^{(i)}-1)/2 \in (\mathbb{Z}/\Pi\mathbb{Z})^*$. This is done by choosing Π a multiple of 4 and by forcing $k \equiv 3 \pmod{4}$ and $a \equiv 1 \pmod{4}$. Putting all this together and letting $\Pi = 4\,\Pi_{\mathrm{odd}}$ (with $\Pi_{\mathrm{odd}} = \prod_i p_i$ for primes $p_i > 2$) drives us to the algorithm described on Fig. 69.

```
Parameters: $\Pi = 4\,\Pi_{\mathrm{odd}}$, $l = v\Pi$, $m = w\Pi$, $t$
            $a \in \mathrm{QR}(m) \setminus \{1\}$ s.t. $a \equiv 1 \pmod 4$, and
            $u \in (\mathbb{Z}/m\mathbb{Z})^*$ s.t. $u \notin \mathrm{QR}(p), \forall p \mid \Pi_{\mathrm{odd}}$
Output:     a random prime $q \in [q_{\min}, q_{\max}]$ with $(q-1)/2$ prime

 1. Randomly choose $\chi \in (\mathbb{Z}/m\mathbb{Z})^*$
 2. Set $k = 4u\chi^2 + \big(4 - (\Pi_{\mathrm{odd}} \bmod 4)\big)\Pi_{\mathrm{odd}} \mod m$
 3. Set $q \leftarrow [(k-t) \bmod m] + t + l$
 4. If ($\mathsf{T}(q) =$ false or $\mathsf{T}((q-1)/2) =$ false) then
    a) Set $k \leftarrow ak \pmod m$
    b) Go to Step 2
 5. Output $q$
```

Fig. 69. Safe prime generation algorithm for $q \in [q_{\min}, q_{\max}]$.

7.2 Quasi-Safe Primes

A d-quasi-safe prime is a prime number q such that $(q-1)/2^d$ is a prime number. It is straightforward to extend our algorithm to the case of d-quasi-safe prime numbers. In this case, the constraint $q^{(i)} \equiv 3 \pmod 4$ has to be extended to $q^{(i)} \equiv 2^d + 1 \pmod{2^{d+1}}$.

7.3 Efficiency

Heuristically, about

$$\rho = \left(\frac{\phi(\Pi)}{\Pi} \ln q_{\max} + 1\right)\left(\frac{\phi(\Pi)}{\Pi} \ln q_{\max}\right)$$

primality tests are required. This is ≈ 25 times faster than incremental search algorithms (where we iterate $q = q + 2$ until q and $(q-1)/2$ are simultaneously prime) for 512-bit numbers.

7.4 Security

Since the groups of quadratic (resp. non) residues modulo $p \mid \Pi$ are all cyclic and contain $(p-1)/2$ elements, the set of possible outputs of our algorithms is a fraction of the set of safe primes, the ratio being $2^{-\nu}$ where ν is the number of distinct primes dividing Π. So there exists a loss of roughly ν bits of entropy compared to the uniform distribution over safe primes (heuristically). This may not be a lot, because typically $\nu = 74$ for 512-bit numbers and safe primes, although being quite scarce, remain numerous (about $2^{512-2\cdot\log 512} = 2^{494}$). Besides, as $n \to \infty$, ν becomes negligible before $n - 2\log n$ (see prior literature [332]), making the bias vanishing asymptotically.

8 Conclusion

We devised simple computation techniques that improve the work of [332] in multiple ways. It is argued that our algorithms present much better performances than previous, classical methods.

We also would like to stress that our prime generation algorithm may support additional modifications *mutatis mutandis* in order to simultaneously reach other properties on q — for instance forcing the last bits of q to fit the Rabin-Williams cryptosystem with even public exponents. Independently, some applications require that the pair of primes satisfy specific properties such as being strong or compliant with ANSI X9.31 recommendations [20]. We refer the reader to [332] for a collection of mechanisms allowing to produce such primes. We point out that our improvements may coexist perfectly with these.

We also proposed a specific implementation for generating safe prime numbers which really boosts real-life execution performances. We stress that, implementing our techniques, a complete RSA key generation process can be executed on any given crypto-enhanced embedded processor in nearly all circumstances and with extremely attractive running times.

Cryptanalysis of LILI-128

by Steve Babbage *

Abstract. LILI-128 is a stream cipher that was submitted to NESSIE. Strangely, the designers do not really seem to have tried to ensure that cryptanalysis is no easier than by exhaustive key search. We show that there are indeed attacks faster than exhaustive key search. We also demonstrate a related key attack which has very low complexity, and which could be of practical significance if the cipher were used (misused?) in a certain rather natural way.

Keywords. LILI-128, stream cipher, NESSIE, time-memory tradeoff, rekeying, related key attack.

1 Introduction

LILI-128 is a synchronous stream cipher designed by Dawson, Clark, Golić, Millan, Penna and Simpson [187], and submitted to NESSIE. It uses a 128-bit key.

No very serious effort seems to have been made by the designers to ensure that cryptanalysis of this cipher is as hard as exhaustive search on a 128-bit key. For instance they write that:

> *. . . we conjecture that the complexity of divide and conquer attacks on LILI-128 is at least* 2^{112} *operations. . . This is a conservative estimate, and the true level of security may be much higher.*

But it seems reasonable to insist that any cipher recommended by NESSIE should not be subject to any attack faster than exhaustive key search. In this note we show that there are indeed attacks faster than exhaustive key search. We also demonstrate a related key attack which has very low complexity, and which could be of practical significance if the cipher were used in a certain rather natural way.

2 Overview of LILI-128

There are two LFSRs: $LFSR_c$, which is 39 bits long, and $LFSR_d$, which is 89 bits long (so a total of 128 bits of internal state). Both have primitive feedback polynomials. For each keystream bit:

* Vodafone Group R&D, Newbury, UK
`steve.babbage@vodafone.com`

- The keystream bit is produced by applying a nonlinear function f_d to 10 of the bits in $LFSR_d$. f_d is balanced, of course; it has nonlinear order 6 and correlation immunity of degree 3. The stages from which the inputs are taken form a full positive difference set.
- $LFSR_c$ is clocked once. Two bits from $LFSR_c$ determine an integer c in the range $\{1, 2, 3, 4\}$.
- $LFSR_d$ is clocked c times.

The keystream generator is initialised simply by loading the 128 bits of key into the registers. Keys that cause either register to be initialised with all zeroes are considered invalid.

3 Time-Memory Tradeoff Attack

The simplest observation to be made is that the size of the internal state is only 128 bits, and so there are clearly time-memory tradeoff attacks faster than exhaustive search if any significant quantity of observed keystream is available (see [1, 2]).

The usual time-memory tradeoff involves:

- a preprocessing stage in which a large dictionary is built containing many (state, 128-bit keystream sequence) pairs, sorted by keystream sequence;
- an actual attack stage, in which observed (overlapping) 128-bit keystream sequences are looked up in the dictionary; if a match is found, then with high probability the associated state was the internal state of the generator when that observed keystream sequence was produced.

The basic attack introduced by Babbage [27] has complexity[1] $T = D = N/M$ and $P = M = N/D$, where T is the time for the actual attack stage, D is the quantity of observed keystream, N is the size of the internal state space (so 2^{128} in this case), M is the amount of memory required, and P is the time for the preprocessing stage. Even if a generous 2^{40} observed keystream bits were available, the dictionary would require memory for 2^{88} records, which is clearly impractical.

Biryukov, Shamir and Wagner [87,89] introduced techniques for saving memory, allowing a more flexible tradeoff $TM^2D^2 = N^2$ (and still $P = N/D$) for any $D^2 \leq T \leq N$. In this case, with 2^{40} observed keystream bits and memory for 2^{36} records, the time for an actual attack is 2^{104}. The memory and observed keystream requirements are just about feasible, and the time for both stages is faster than exhaustive search (although logarithmic terms have been omitted, which in practice would push the time closer to 2^{128}). With only 2^{28} observed keystream bits and memory for 2^{36} records, T becomes 2^{128}, so there is no improvement over exhaustive search.

But it must be remembered that this tradeoff is just about finding a common item in two lists: list A of keystream sequences generated from known states, and

[1] We ignore logarithmic terms.

list B of observed keystream sequences. One list is sorted into a dictionary, and then items on the other list are looked up in the dictionary. As observed in [27], it can be either of the lists that is sorted into a dictionary. So even with only 2^{28} observed keystream bits, an attack is possible with time complexity 2^{100} and memory 2^{28}: sort the observed overlapping 128-bit keystream sequences into a dictionary, then repeatedly (around 2^{100} times) try a random state, generate 128 bits of keystream from it, and look for the result in the dictionary.

It is clear that any significant quantity of consecutive keystream bits (or, more generally, regularly spaced linear combinations of keystream bits) can be used in this way for an attack that is faster than exhaustive key search. The more observed bits, the faster the attack.

4 Solving Simultaneous Linear Equations

Guess the 39 key bits used to initialise the clock control register $LFSR_c$. For the correct guess, you then know exactly how many times $LFSR_d$ has been clocked when each keystream bit is generated. Each keystream bit is thus a 6^{th} order function of ten bits, each of which is a known linear combination of the remaining 89 key bits. So each keystream bit is a known linear combination of all the possible products of up to 6 of those 89 bits.

There are $\sum_{i=1}^{6} \binom{89}{i} = 625173825 \approx 1.16 \times 2^{29}$ products of up to 6 from 89 bits. So with roughly that many observed keystream bits, the problem reduces to solving simultaneous linear equations in that many variables. (We also have to reject incorrect guesses for $LFSR_c$, but that is simple — either the linear equations will be inconsistent, or else their solutions will be inconsistent when interpreted as products of secret key bits.)

Without trying to implement it, it is difficult to know exactly how long solving that many simultaneous equations would take in practice. Coppersmith and Winograd [148] have an asymptotic time complexity for matrix inversion of $O(n^{2.376})$, but with a large constant factor. Strassen's algorithm [599] has complexity $7n^{2.807} - 6n^2$, so $2^{84.8}$ in this case; the overall attack would therefore have time complexity $2^{39+84.8} = 2^{123.8}$, which is marginally less than exhaustive key search. However, just storing the coefficients of the equations would require 2^{58} bits, which is impractical. So, with today's computers and algorithms, this seems to be an academic rather than a practical attack.

A rather similar attack is considered by the designers in [187], but they suggest that roughly $2^{39}\binom{89}{6}$ keystream bits would be required. As we see above, by guessing the contents of $LFSR_c$ and then performing a linearity attack just on $LFSR_d$, we eliminate the factor 2^{39}.

5 Rekeying / Related Key Attacks

It is very common for stream ciphers to be used repeatedly with the same secret key, loaded in combination with some varying non-secret initialisation vector.

There is therefore good reason to consider the effect of this rekeying — which in effect amounts to a related key attack, but with rather more justification than related key attacks tend to have against block ciphers.

The simplest way to combine a secret key and an IV is to XOR them together. (The use of IVs for encryption of multiple frames in WEP, in which a secret component is concatenated with a varying IV, can be viewed as a special case of this.) If LILI-128 is rekeyed in this way, then the system can become extremely weak, as we now explain.

Note: when this paper was originally written, the designers of LILI-128 had not specified a rekeying mechanism. Since then, they have [186]; the mechanism they propose avoids the attack described here. So this attack now serves as:

- *a warning that, if LILI-128 is to be used for a rekeyed application, the mechanism proposed in [186] should be used, rather than the more naïve one described above;*
- *a general related key attack, with as much chance of becoming relevant as a related key attack on a block cipher (see [345] for a discussion on how these attacks can become relevant in practical situations).*

Suppose that:

- the 128-bit secret key is k;
- a number of successive 128-bit IVs are $v_1 ... v_r$;
- LILI-128 is loaded (i.e. the registers initialised) with $k \oplus v_i$;
- the corresponding keystream sequences are available to the cryptanalyst.

The attack proceeds in two phases, which we will first outline and then describe in slightly more detail:

Phase 1: Guess the 39 secret key bits used to initialise the clock control register $LFSR_c$, and quickly reject incorrect guesses, so that the correct value is known. For each IV v_i, we now know exactly how many times $LFSR_d$ has been clocked when each keystream bit is generated.

Phase 2: Compare several keystream bits produced using different IVs but when $LFSR_d$ has been clocked exactly the same number of times. Deduce the secret key components of the 10 input bits to the nonlinear function f_d at that point. Repeat several times, to obtain plenty of linear equations in the 89 secret key bits used to initialise $LFSR_d$. Solve those linear equations to obtain the secret key bits.

For the detail, we will introduce some notation. When $LFSR_d$ is initialised with just the secret key k (i.e. with IV all 0s), and then clocked t times, let the 10-bit vector representing the inputs to the nonlinear output function f_d be k_t. When $LFSR_d$ is initialised with just v_i and then clocked t times, let the 10-bit vector representing the inputs to f_d be v_{it}.

Clearly, when $LFSR_d$ is initialised with $k \oplus v_i$ and then clocked t times, the 10-bit vector representing the inputs to f_d is $k_t \oplus v_{it}$.

Detail of phase 1. We can reject incorrect guesses for $LFSR_c$ as follows. Find a value t, and different IVs v_i and v_j, such that v_{it} and v_{jt} are equal, and for which keystream bits are in fact generated in each case when, according to our guess, $LFSR_d$ has been clocked exactly t times. (For any fixed t, the probability that a keystream bit will be generated when $LFSR_d$ has been clocked exactly t times is approximately 0.4.) If our guess is correct, the two keystream bits must be equal. If it is incorrect then they will be equal with probability roughly $\frac{1}{2}$.

Roughly 39 comparisons will suffice to reject all incorrect guesses, and identify the correct one.

For this method to work, there are tradeoffs between the number r of different IVs available and the length l of each keystream sequence; the nature of the tradeoffs depends to some extent on the nature of the IVs. If different IVs are independently random, then there are roughly $(0.4)(2^{-10})\binom{r}{2}l$ pairs of keystream bits with the same values of v_{it} and v_{jt}; this formula reaches the required value of ≈ 39 for instance when $r = 32$ and $l = 202$, or when $r = 16$ and $l = 832$, or when $r = 64$ and $l = 50$. We need $l \geq 20$ to ensure that the whole of $LFSR_c$ is covered. The analysis is slightly more complex if successive IVs are related, e.g. if they are successive values of some counter.

If we guess all 39 bits of $LFSR_c$ together, and then look to reject incorrect guesses, then the complexity is slightly greater than 2^{39}. But in fact we can break the work down and guess just a couple of bits at a time. (Guess the two secret key bits contributing to the first integer $c \in \{1, 2, 3, 4\}$; confirm or reject this guess as described above; go on to the two secret key bits contributing to the second integer c; etc etc.) So the complexity of Phase 1 is very low indeed.

Detail of phase 2. We now know exactly how many times $LFSR_d$ has been clocked when each keystream bit is generated. We proceed to determine the contents of $LFSR_d$.

For some value t, find several different IVs v_i such that in each case keystream bits are generated when $LFSR_d$ has been clocked exactly t times. Then consider all possible values for k_t. For the correct value of k_t, the observed keystream bit for IV v_i will always equal $f_d(k_t \oplus v_{it})$; for incorrect values of k_t, equality will hold with probability roughly $\frac{1}{2}$. Roughly ten different IVs will suffice to reject all incorrect guesses, and determine the correct one.

Determining k_t gives us ten linear equations in the secret key bits used to initialise $LFSR_d$. Repeating 10 or 11 times will give us 100–110 equations, which should be enough to determine those 89 secret key bits (there will be some overlap between the equations since the same register bit will appear repeatedly in different positions).

The complexity of Phase 2 is again extremely low. If even very short (not much more than 10-bit) keystream sequences are available for 25 different IVs, then for enough values of t the expected number of times a keystream bit is generated when $LFSR_d$ has been clocked exactly t times is approximately 10, which is sufficient. Slightly fewer keystream sequences will suffice if they are longer (the values of t with ten or more "hits" will be more scattered).

Other comments and summary. There are variations on the above process. k_0 can be determined without knowing anything at all about $LFSR_c$. Phases 1 and 2 could be combined: guess the first few bits used from $LFSR_c$, and one of k_1, k_2, k_3 and k_4, rejecting inconsistent guesses for both together; determine the rest of k_1, k_2, k_3 and k_4; go on to the next few bits used from $LFSR_c$, and so on. And we don't necessarily have to reject all but one possibility at every stage — we can keep a few possibilities "live" at once, as long as the number doesn't keep growing. With this combined approach it suffices to have roughly 30 sequences of a little over 20 bits each. Anyway, it is clear that an attack of this kind can be performed if a rather small number of rather short keystream sequences are available. And the complexity of the attack is very low (real-time, even, as far as that makes sense for an attack on multiple uses of the same cipher).

We have restricted ourselves to keys related by XORing different known IVs. If the IVs are *chosen* by the cryptanalyst — which is an optimistic but not completely fanciful assumption — then variations become possible with even smaller data requirements. Keys with more general chosen relationships would open up a host of other possibilities, but of less practical significance.

6 Design Criteria for the Nonlinear Function

The keystream bit is computed using a non-linear function on 10 of the bits in $LFSR_d$. Amongst other criteria, the function was chosen to have fairly high order correlation immunity (order 3). This choice was made to give resistance against correlation attacks.

This seems to be a misguided application of correlation immunity. Having no correlation to subsets of up to three of the input bits is rather pointless, because there is correlation to sums of four or more bits — and any sum of the bits from four or more stages of an LFSR is itself a linear sequence from the same LFSR. When all input bits come from one LFSR, sums of small numbers of input bits are no more in need of protection from correlation attacks than sums of large numbers of input bits.

As noted in section 4.3 of [187], there is merit in having at least first order correlation immunity, to prevent attacks that track a bit from one position in $LFSR_d$ to another. But correlation immunity of order greater than one seems an inappropriate criterion (the input stages to f_d form a full positive difference set, so no two bits appear together twice as inputs). A more appropriate criterion might have been to choose a balanced, first order correlation immune function with minimum correlation to any linear function of more than one bit. (It may be that any such function achieves only marginally better nonlinearity than the LILI-128 function — this observation is about the criteria for selecting the function rather than the function itself.)

7 Conclusions

General. If a general-purpose cipher has a 128-bit key, it is expected that there should be no attack faster than 128-bit exhaustive search. But it does not appear that the designers of LILI-128 have really tried to ensure that there are no attacks faster than exhaustive key search; there are various faster attacks, including at least one very straightforward one.

Related key attacks. For better or for worse, related key attacks against block ciphers are taken seriously. A related key attack faster than exhaustive key search against one of the AES candidates would have been enough to remove it from contention. We have demonstrated a related key attack against LILI-128 which requires only a few tens of related keys, and has very low complexity; these related keys could be available in practical use if the system is rekeyed in a certain naïve way, rather than according to the designers' recent recommendation, or in any other context in which a related key attack could become practical.

Differential, Linear, Boomerang and Rectangle Cryptanalysis of Reduced-Round Camellia

by Taizo Shirai [*]

Abstract. We propose a new search method to obtain differential characteristics and linear approximations with high probability from certain truncated paths of non-trivial rounds of Camellia. Using this technique, we show the differential characteristic with probability $p = 2^{-102}$ for 8-round Camellia without FL/FL^{-1} and whitenings (denoted as Camellia*), the linear approximation with probability $p = 2^{-118.62}$ for 9-round Camellia* and a distinguisher for boomerang and rectangle attacks with probability $\hat{p} = 2^{-50.97}, \hat{q} = 2^{-9.48}$ for 8-round Camellia. Moreover, using these results, we describe the following key recovery attacks which are faster than exhaustive key search of the 256-bit key cipher, the differential attack on 11-round Camellia*, the linear attack on 12-round Camellia*, the boomerang attack on 9-round Camellia and and the rectangle attack 10-round Camellia.

Keywords. Camellia, the differential attack, the linear attack, the boomerang attack, the rectangle attack, differential characteristic, linear approximation

1 Introduction

Camellia is a 128-bit block cipher designed by NTT and Mitsubishi Electric Corporation, which was suggested as a candidate for the NESSIE project and selected for the 2nd phase of the project, and also suggested as a candidate for the CRYPTREC project in Japan [24, 631, 633]. The number of attackable rounds or security measurement of Camellia have been considered in many published papers [25, 287, 292, 338, 339, 355, 392, 578, 602, 627]. However, in relation to two basic cryptanalysis, the differential cryptanalysis (DC) and the linear cryptanalysis (LC), the immunity of Camellia has been discussed only by the upper bounds of maximum differential characteristic probability (MDCP) and maximum linear characteristic probability (MLCP) [25,77,421,578]. No method has been reported to find concrete differential characteristics and linear approximations effective for the attacks on non-trivial rounds of Camellia. Even though the maximal probability cannot be found, finding differential characteristics or linear approximations with high probability near to the upper bounds should be considered as useful

[*] Sony Corporation
7-35 Kitashinagawa 6-chome, Shinagawa-ku, Tokyo, 141-0001 Japan
Taizo.Shirai@jp.sony.com

measurements for the cipher. Particularly revealing the probability is necessary to mount key recovery attacks.

At FSE2002, Shirai *et al.* proposed an improved algorithm for evaluation of the upper bounds of MDCP and MLCP [578]. The algorithm is based on discarding truncated paths which contain algebraic contradictions, yielding successful reevaluation of the lower bound of the least number of active S-boxes for Camellia without FL/FL^{-1} and whitenings(denoted as Camellia*).

We found that the underlying relation between output values of active S-boxes can be obtained by modifying Shirai *et al.*'s algorithm. The relation is very effective to reduce the search space of the differential characteristics and the linear approximations. Finally, we construct a feasible algorithm to search differential characteristics and the linear approximations with high probability near the upper bounds.

We apply the proposed method for experimental analysis of Camellia. As a result, we found an 8-round differential characteristic of Camellia* with probability $p = 2^{-102}$, a 9-round linear approximation with probability $p = 2^{-118.62}$ and an 8-round distinguisher for boomerang and rectangle attacks with probability $\hat{p} = 2^{-50.97}, \hat{q} = 2^{-9.48}$.

Moreover, using these results, we describe the following key recovery attacks which are faster than exhaustive key search of 256-bit key: the differential attack on 11-round Camellia*, the linear attack on 12-round Camellia*, the boomerang attack on 9-round Camellia and and the rectangle attack on 10-round Camellia.

This paper is organized as follows: In section 2, we give a short description of Camellia. In Section 3, we show a search methodology using the relation information obtained from Shirai *et al.*'s algorithm. In Section 4 we present results of the proposed method. In Section 5 we show attacks on Camellia* and Camellia by using the result s shown in the previous section. Section 6 contains a comparison of our result versus other reported results. Section 7 summarizes our conclusions.

2 Definition

In this paper, we use the following definitions.

Definition 0.1. (Differential Probability of f) *The differential Probability of function f is defined as follows:*

$$DP_f(\Delta x, \Delta y) = \frac{\sharp\{x \in \{0,1\}^n | f(x) \oplus f(x \oplus \Delta x) = \Delta y\}}{2^n}$$

Definition 0.2. (Linear Probability of f) *The linear Probability of function f is defined as follows:*

$$LP_f(\Gamma x, \Gamma y) = \left(2\frac{\sharp\{x \in \{0,1\}^n | x \cdot \Gamma x = f(x) \cdot \Gamma y\}}{2^n} - 1\right)^2$$

Definition 0.3. (reduced row-echelon matrix) *A matrix is a reduced row-echelon matrix if*

- *All rows of zero (if exists) are at the bottom of the matrix.*
- *The first nonzero number in a row is a 1 (leading 1).*
- *Each leading 1 is to the right of the leading 1's in the rows above it.*
- *Each column that contains a leading 1 has zeros everywhere else.*

Any matrix can be transformed into an unique reduced row-echelon matrix by performing a finite sequence of elementally row operations called sweep out method.

3 A Description of Camellia

Camellia is a Feistel block cipher with a block size of 128 bits and accepts 128, 192 and 256 key bits [24]. The number of rounds are determined by a key length, 18 rounds for 128 bits key, 24 rounds for 192 and 256 bits keys. The F-function of Camellia is composed of a so-called SPN type structure.

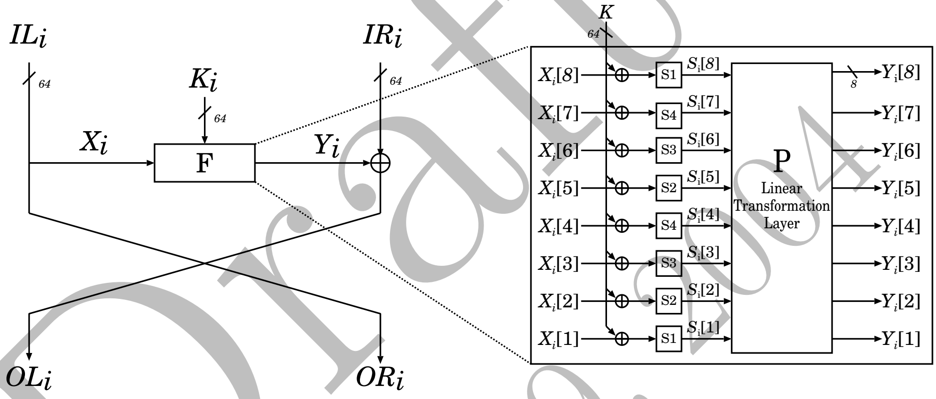


Fig. 70. the i-th round of a Feistel Network with SPN round function

In the i-th round, 64 bits of key K_i and 64 bits of data X_i are the input to the F-function. The input of F-function is split up into 8 bytes, represented by $X_i = (X_i[1], X_i[2], \ldots, X_i[8])$. After the key addition operation, each byte enters one of four type of S-boxes S1, S2, S3 and S4, which is determined by the position of the byte. Then the output of S-boxes $S_i = (S_i[1], S_i[2], \ldots, S_i[8])$ enters the linear transformation layer $P : \{0,1\}^{64} \rightarrow \{0,1\}^{64}$. The output of the linear transformation layer $Y_i = (Y_i[1], Y_i[2], \ldots, Y_i[8])$ is calculated as follows:

$$\begin{pmatrix} Y_i[1] \\ Y_i[2] \\ Y_i[3] \\ Y_i[4] \\ Y_i[5] \\ Y_i[6] \\ Y_i[7] \\ Y_i[8] \end{pmatrix} = \begin{pmatrix} 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \end{pmatrix} \begin{pmatrix} S_i[1] \\ S_i[2] \\ S_i[3] \\ S_i[4] \\ S_i[5] \\ S_i[6] \\ S_i[7] \\ S_i[8] \end{pmatrix}$$

where addition in the above computation is exclusive-or operation. Also let $a_{ij}, (1 \leq i, j \leq 8)$ be an element of the above matrix in the i-th row and the j-th column. Fig. 70 shows the i-th round of Camellia.

Camellia also includes a key scheduling algorithm, key whitening layers and key-dependent linear functions FL and FL^{-1} which are inserted every 6 rounds. Details of these components are described in [24]. We assume that round keys are independent and uniformly random.

Property 3.1. All S-boxes used in Camellia are differentially 4-uniform bijective functions generated by affine and inverse functions on $GF(2^8)$. The maximum differential probability (MDP) of the S-boxes are 2^{-6}. For a fixed input difference ΔX, $DP_S(\Delta X, \Delta Y) = 2^{-6}$ occurs 1 time for a output difference ΔY, and $DP_S(\Delta X, \Delta Y) = 2^{-7}$ occurs 126 times for other ΔYs. And this property also applicable to a fixed output difference ΔY. The values of linear probability can take one of nine probability in $\{2^{-6}, 2^{-6.39}, 2^{-6.83}, 2^{-7.36}, 2^{-8}, 2^{-8.83}, 2^{-10}, 2^{-12}, 0\}$. Therefore, the MLP are also 2^{-6}.

4 Search Methodology

In this section we show a method to get differential characteristics and linear approximations from a given truncated path.

A truncated path is a differential characteristic or linear approximation which is represented in truncated way. In the case of Camellia, each byte value in differences or linear masks is mapped to '0' or '1' according to whether the value is 0 or not, respectively [352,578]. Shirai *et al.* showed an evaluation algorithm to determine whether a given truncated path contain an algebraic contradiction [578].

We found the new underlying relation in truncated paths which hold no contradiction as follows. The upper side of Fig. 71 shows example of a truncated differential path with 9 active S-boxes for 5-round Camellia*. We note that 9 is the lower bound of the least number of active S-boxes for 5-round Camellia* [578]. Let the *variables* be the bytes of plaintext difference and output difference of S-boxes which is corresponding to 1's in the truncated path. The direct search algorithm to generate differential characteristics tries to evaluate all possible values of the variables. However, the number of variables is 14 (9 active S-boxes and 5 nonzero bytes of the plaintext), implying that the complexity $(2^8)^{14} = 2^{112}$ is too large for limited resources.

We solve the computation problem by labeling each variable as free or not. The lower side of Fig. 71 represents all variables by using the free variables in the truncated path. In this case there are only 4 free variables w, x, y and z among 14 variables. It means that the total number of candidates of differential characteristics is only $(2^8)^4 = 2^{32}$. With such sufficiently reduced variables we can search all paths exhaustively for finding the paths holding the best differential characteristic probability (DCP).

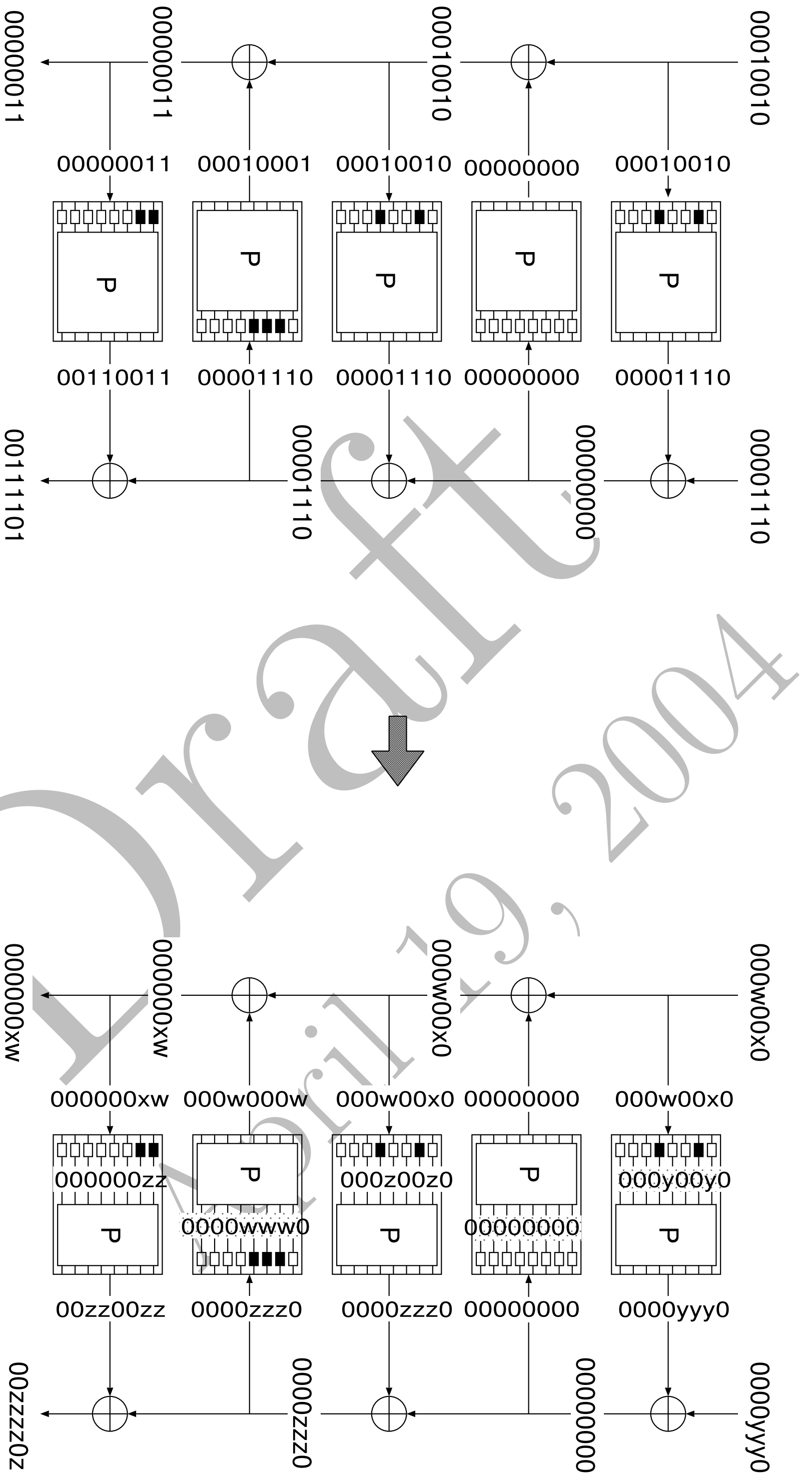


Fig. 71. Truncated Differential Path and Free Variable Representation

4.1 Algorithm to Find Free Variables of Truncated Paths

Now we show an algorithm to get the information of the free variables for a given truncated differential path.

A Feistel network can be divided into two series, one is called "the right chain" which starts from the right half of plain text and output values of the F-functions in the odd rounds are added to the chain, and the other part is called "the left chain" which start from left half of plain text. We continue to explain our method with the case of the right chain.

Given the properties of SPN type F-functions, the differential data

$$\Delta Y_{2i-1}[j], \Delta X_{2i}[j], (1 \leq i \leq r/2)$$

in the right chain can be expressed as the following linear forms, where r is the round number. (Without loss of generality we assume r is even.):

$$\Delta Y_{2i-1}[j] = \sum_{k=1}^{8} a_{jk} \Delta S_{2i-1}[k] \tag{3}$$

$$\Delta X_{2i}[j] = \Delta PR[j] + \sum_{l=1}^{i} \sum_{k=1}^{8} a_{jk} \Delta S_{2l-1}[k] \tag{4}$$

From the above representation we get $8r$ linear equations in the right chain.

Each variable in the equations has a truncated differential value 0 or 1 determined by a given truncated differential path. Removing variables corresponding to 0 from these equations, we can obtain a system of homogeneous linear equations which also contains $8r$ equations.

Let $M \cdot v$ be a matrix and vector form of the system of linear equations. After transforming M into reduced row echelon matrix M_{re} (see definition 3), label each variable as free or bound as follows.

- Label variable as 'bound' if it corresponds to a leading 1 in M_{re}
- Label variable as 'free' if it is not labeled as 'bound'

Fig. 72 shows an example of a reduced row-echelon matrix. In this example, variables v_1, v_2, v_3, v_4, v_6 correspond to leading 1's, thus these variables are labeled as bound. Thus the number of free variables is 4. More generally, let n be the column number and r be the rank of M_{re}, the number of free variables is $n - r$, which is known as dimension of the kernel space.

The same method can also be applied to retrieve linear approximations by replacing the linear transformation matrix with the matrix which transforms the output linear mask into the input linear mask [337, 578].

Let I be the sum of number of free variables in both the chains. If I is small enough, we can search all candidates of differential paths and linear approximations for a given truncated path. Tables 57, 58 show the distribution of the numbers of free variables for truncated differential paths and truncated linear paths with the least number of active S-boxes obtained from Shirai *et al.*'s result [578]. In the next section we show our experimental search of DCP, DLP, $\hat{p}, \hat{q}$ from truncated paths with sufficiently small I.

$$\begin{array}{ccccccccc}
v_1 & v_2 & v_3 & v_4 & v_5 & v_6 & v_7 & v_8 & v_9 \\
1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\
0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\
0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}$$

Fig. 72. An Example of Reduced Row-Echelon Matrix with 9 variables

Table 57. Distribution of Truncated Differential Paths

Round	Active	Total	$I = 1$	$I = 2$	$I = 3$	$I = 4$	$I = 5$	$I = 6$	$I = 7$	$I = 8$	$I = 9$
1	0	255	8	28	56	70	56	28	8	1	0
2	1	16	0	16	0	0	0	0	0	0	0
3	2	8	0	0	8	0	0	0	0	0	0
4	7	1328	0	0	16	136	424	496	224	32	0
5	9	80	0	0	0	8	32	32	8	0	0
6	12	1008	0	8	48	148	224	284	216	72	8
7	14	196	0	0	16	44	56	52	24	4	0
8	16	36	0	0	0	16	16	4	0	0	0
9	20	812	0	0	0	56	88	228	296	128	16
10	22	156	0	0	0	0	16	72	56	12	0

Active - the lower bound of the least number of active S-boxes
Total - the total number of truncated paths

Table 58. Distribution of Truncated Linear Paths

Round	Active	Total	$I = 1$	$I = 2$	$I = 3$	$I = 4$	$I = 5$	$I = 6$	$I = 7$	$I = 8$	$I = 9$
1	0	255	8	28	56	70	56	28	8	1	0
2	1	16	0	16	0	0	0	0	0	0	0
3	2	8	0	0	8	0	0	0	0	0	0
4	6	16	0	4	8	4	0	0	0	0	0
5	9	240	0	8	32	64	80	48	8	0	0
6	12	1948	0	24	160	320	448	544	352	92	8
7	14	192	0	0	0	20	56	80	32	4	0
8	17	1560	0	0	8	120	424	568	344	88	8
9	19	144	0	0	0	16	72	48	8	0	0
10	22	792	0	0	0	48	280	320	128	16	0

5 Experimental Search

5.1 Searching for the Best DCP and DLP

In this section we show a method to calculate the best DCP and LCP by combining pre-computed differential and linear probabilities of S-boxes and path candidates obtained from truncated paths with well reduced number of free variables I.

From the definition of I, there are 2^{8I} possible solutions in a truncated path with I free variables. Let C be the set of all solutions for a given truncated path. Then the DCP determined by $c_i \in C$, denoted $DCP(c_i)$, can be obtained by using the following lemma.

Lemma 0.1. *For $c_i \in C$, let $\alpha, \alpha_1, \alpha_r$ be the total number of active S-box, the number of active S-boxes in the first round, and the number of active S-boxes in the last round. For the j-th active S-box of c_i $(1 \leq j \leq \alpha)$, let $\Delta I_{c_i,j}, \Delta O_{c_i,j}$ be the input and output differences, and let $S_{c_i,j}$ be a kind of the S-box (S_1, S_2, S_3 and S_4). Then $DCP(c_i)$ is,*

$$DCP(c_i) = 2^{-6(\alpha_1+\alpha_r)} \prod_{i=\alpha_1+1}^{\alpha-\alpha_r} DP_{S_{c_i,j}}(\Delta I_{c_i,j}, \Delta O_{c_i,j}) \qquad (5)$$

Proof. Basically, we can construct the characteristic probability $DCP(c)$ for c_i as follows:

$$DCP'(c_i) = \prod_{i=1}^{\alpha} DP_{S_{c_i,j}}(\Delta I_{c_i,j}, \Delta O_{c_i,j}) \qquad (6)$$

Furthermore, the differential probability $DCP_{S_{c_i,j}}$ can be replaced by the probability 2^{-6}, which is the best one, in the first round and the last round. Because we can arbitrarily change the input differences in the first round and output differences in the last round. Thus we can obtain the equation (5). □

In the same manner, the values $LCP(d_i)$ for $d_i \in D$ can be obtained by replacing $DP_{S_{c_i,j}}(\Delta I_{c_i,j}, \Delta O_{c_i,j})$ with $LP_{S_{d_i,j}}(\Gamma I_{d_i,j}, \Gamma O_{d_i,j})$ where D is a set of linear approximations and d_i is a candidate in D.

5.2 Searching for the Best $\hat{p}$ and $\hat{q}$

The boomerang attack and the rectangle attack employ the differential attack as a building block and use two short round probabilities $\hat{p}, \hat{q}$ to construct distinguisher [69, 71, 211, 344, 616]. The $\hat{p}, \hat{q}$ are defined as follows:

$$\hat{p} = \sqrt{\sum_{\alpha \overset{\beta}{\to} \beta} Pr^2[\alpha \to \beta]} \qquad \text{and} \qquad \hat{q} = \sqrt{\sum_{\gamma \overset{\gamma}{\to} \delta} Pr^2[\gamma \to \delta]}$$

Calculating accurate $\hat{p}$ and $\hat{q}$ is a computationally difficult problem, so we calculated them approximately by characteristic probabilities. Using $DCP(c_i)$,

α_1, α_r defined before, we approximately calculated all possible combinations of the output differences of active S-boxes in the last round and the input differences of active S-boxes in the first round, respectively.

$$\hat{p}(c_i) = DCP(c_i)\sqrt{1 + \sum_{i=1}^{\alpha_r} \frac{{}_{\alpha_r}C_i \cdot 126^i}{(2i)^2}}$$

$$\hat{q}(c_i) = DCP(c_i)\sqrt{1 + \sum_{i=1}^{\alpha_1} \frac{{}_{\alpha_1}C_i \cdot 126^i}{(2i)^2}}$$

Using these formulas, we searched the best $\hat{p}(c_i)$ and $\hat{q}(c_i)$ from all path candidates.

5.3 Search Results

We found the best DCP, DLP, $\hat{p}$ and $\hat{q}$ by experiments. In this experiment, we searched for all truncated paths whose number of free variables $I \leq 4$ due to the limitation of the computational power. For example the search for a path with $I = 5$ takes a few days by our algorithm. Results are shown in Table 59.

Table 59. Search Results

Round	Active	Upper Bound [578]	Best DCP	Best $\hat{p}, \hat{q}$	Active	Upper Bound [578]	Best LCP
1	0	1	1	1	0	1	1
2	1	2^{-6}	2^{-6}	$2^{-3.49}$	1	2^{-6}	2^{-6}
3	2	2^{-12}	2^{-12}	$2^{-9.49}$	2	2^{-12}	2^{-12}
4	7	2^{-42}	2^{-42}	$2^{-27.83}$	6	2^{-36}	$2^{-36.77}$
5	9	2^{-54}	2^{-56}	$2^{-50.98}$	9	2^{-54}	$2^{-54.39}$
6	12	2^{-72}	2^{-74}	$2^{-66.08}$	12	2^{-72}	$2^{-72.39}$
7	14	2^{-84}	2^{-88}	$2^{-82.98}$	14	2^{-84}	$2^{-84.77}$
8	16	2^{-96}	2^{-102}	$2^{-96.98}$	17	2^{-102}	$2^{-102.77}$
9	20	2^{-120}	2^{-130}	$2^{-127.49}$	19	2^{-114}	$2^{-118.62}$
10	22	2^{-132}	–	–	22	2^{-132}	–

From the above results, we can conclude that 9-round Camellia* is distinguishable from random permutation using a linear approximation with probability $2^{-118.62}$. Taking into account that $\hat{p}\hat{q} \geq 2^{-64}$ is a necessary condition for boomerang and rectangle distinguisher, we can construct 8-round distinguisher by 4-round and 4-round blocks, 3-round and 5-round blocks or 5-round and 3-round blocks.

We show a an example of a 9-round linear approximation with probability $2^{-119.02}$ in Fig. 73. We note that this is not the best one in $LCP(c_i)$ but the best one in $LCP'(c)$ which has not replaced the probability of active S-boxes in the first and last round. This approximation is used in efficient key recovery linear attacks in the next section. The distribution of linear probabilities of 19 active S-boxes is shown in Table. 60 where $LP_{Sn}(\Gamma X, \Gamma Y)$ denotes the linear probability of the S-box Sn with input mask ΓX and output mask ΓY.

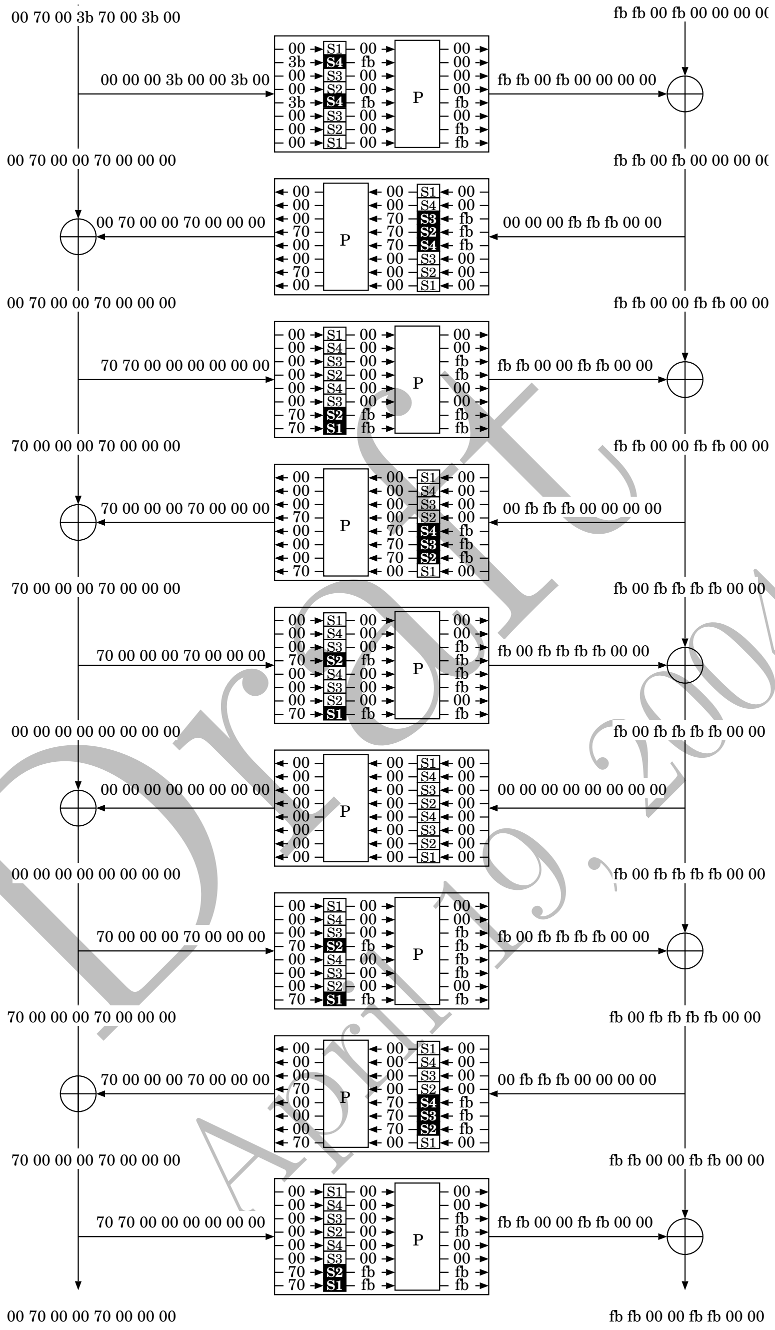


Fig. 73. An Example of Linear Approximation for 9-round Camellia*

Table 60. Distribution of Linear Probability of 19 Active S-boxes

$LP_{S4}(3b_x, fb_x) = 2^{-6}$	2 times
$LP_{S2}(fb_x, 70_x) = 2^{-6.39}$	3 times
$LP_{S3}(fb_x, 70_x) = 2^{-6.39}$	3 times
$LP_{S4}(fb_x, 70_x) = 2^{-6.39}$	3 times
$LP_{S1}(70_x, fb_x) = 2^{-6.39}$	4 times
$LP_{S2}(70_x, fb_x) = 2^{-6}$	4 times

6 Attacking Camellia* and Camellia

6.1 Differential Attack on 11-round 256-bit key Camellia*

First we show the differential attack on 11-round 256-bit key Camellia*. We use the following differential characteristic obtained by the search experiment.

$$\begin{aligned}
(\Delta IL_1, \Delta IR_1) &= (\texttt{0x6400000000000064}, \texttt{0x0000640064323232}) \rightarrow \\
(\Delta IL_2, \Delta IR_2) &= (\texttt{0x0000640064000000}, \texttt{0x6400000000000064}) \rightarrow \\
(\Delta IL_3, \Delta IR_3) &= (\texttt{0x0064000000000000}, \texttt{0x0000640064000000}) \rightarrow \\
(\Delta IL_4, \Delta IR_4) &= (\texttt{0x0064006400640000}, \texttt{0x0064000000000000}) \rightarrow \\
(\Delta IL_5, \Delta IR_5) &= (\texttt{0x0064006400640000}, \texttt{0x0064006400640000}) \rightarrow \\
(\Delta IL_6, \Delta IR_6) &= (\texttt{0x0064000000000000}, \texttt{0x0064006400640000}) \rightarrow \\
(\Delta IL_7, \Delta IR_7) &= (\texttt{0x0000640064000000}, \texttt{0x0064000000000000}) \rightarrow \\
(\Delta IL_8, \Delta IR_8) &= (\texttt{0x6400000000000064}, \texttt{0x0000640064000000}) \rightarrow \\
(\Delta OL_8, \Delta OR_8) &= (\texttt{0x0000640064323232}, \texttt{0x6400000000000064}) \qquad p = 2^{-102}
\end{aligned}$$

Let K_9' be the subkey bit of K_9 corresponding to 5 active S-boxes in the 9th round determined by ΔOL_8. The following key recovery algorithm finds the 40-bit K_9' and 64-bit K_{10} and K_{11} in the 10th and 11th round.

1. Initialize array of 2^{168} counters
2. Encrypt 2^{104} plaintext pairs with difference ΔPL.
3. For each ciphertext pairs do the following:
    - a) For each key value of K_{10} and K_{11} do the following:
        - i. Decrypt the last two rounds using the key
        - ii. Discard pairs with $\Delta OR_9 \neq (\texttt{0x0000640064323232})$
        - iii. For each pair, calculate $\Delta S_9 = P^{-1}(\Delta OL_9 \oplus \texttt{0x6400000000000064})$.
            - A. If all of $\Delta S_9[1], \Delta S_9[2], \Delta S_9[4]$ are not 0, discard the pair.
            - B. For each key value of K_9' do the following: If the key suggests the current ΔS_9, increment the related counter.
4. Look over the counters, and choose the one with the maximum value.

After the step A, the number of remaining pairs is about $2^{104+128-64-24} = 2^{144}$ and the number of expected suggested keys is 1 for each pair. Therefore the S/N is equal to $4/(2^{144}/2^{168}) = 2^{26}$ which is large enough. The data complexity is 2^{105}, memory complexity is 2^{168} and time complexity is total of 2^{105} 11-round encryptions, 2^{235} 2-round decryption, 2^{168} P^{-1} operations and $2^{40} \cdot 2^{145} = 2^{187.32}$ S-box lookups.

Also we can attack 9-round and 10-round Camellia* using the same 8-round differential characteristic. Table 5 summarizes the complexity of these key recovery algorithms.

Table 61. Summery of the complexity of the differential attacks

Round	Key Length	Data	Time	Memory
9	128,192,256	2^{105}	2^{105}Enc.	2^{40}
10	192,256	2^{105}	$2^{165.7}$Enc.	2^{104}
11	256	2^{105}	$2^{231.5}$Enc.	2^{168}

Enc. - Encryptions

6.2 Linear Attack on 12-round 256-bit key Camellia*

Next, we show the steps of a linear attack on 12-round 256-bit key Camellia* using the 9-round linear approximation shown in Fig. 73.

$$(\mathit{\Gamma IL}_1, \mathit{\Gamma IR}_1) = (\texttt{0x0070003b70003b00}, \texttt{0xfbfb00fb00000000})$$
$$(\mathit{\Gamma OL}_9, \mathit{\Gamma OR}_9) = (\texttt{0xfbfb0000fbfb0000}, \texttt{0x0070000070000000}) \quad p = 2^{-119.02}$$

We use a same technique of Biham *et al.*'s algorithm to reduce time complexity of the key recovery algorithm by employing two counters [68]. The linear mask of $\mathit{\Gamma OR}_9$ only contain nonzero values at 2nd and 5th byte. When one round decryption is applied to the 10-th round, these two bytes are affected by the output of 7 S-boxes in the round. Let X'_{10} and K'_{10} be the input data and subkey corresponding to these 7 S-boxes in the 10th round.

The following key recovery algorithm finds the 56-bit subkey K'_{10} and whole of K_{11} and K_{12} in the 11th and 12th round.

Given 2^{120} plaintexts and ciphertexts, perform the following:

1. Initialize array of 2^{185} counters (C1)
2. For each plaintext and ciphertext pair (PL, PR, CL, CR) do the following:
    a) For each possible combination of K_{11} and K_{12} do the following:
        i. Decrypt CL, CR with K_{11}, K_{12}. Let OL_{10}, OR_{10} be the decrypted data.
        ii. Calculate the parity $a = (\mathit{\Gamma IL}_1 \cdot PL) \oplus (\mathit{\Gamma IR}_1 \cdot PR) \oplus (\mathit{\Gamma OL}_9 \cdot OR_{10}) \oplus (\mathit{\Gamma OR}_9 \cdot OL_{10})$
        iii. Advance the C1 counter which is related to the 185-bit value such that the parity a, the 56-bit value in OR_{10} corresponding to subkey K'_{10}, K_{11} and K_{12}.
3. Initialize array of 2^{184} counters (C2)
4. For each key K'_{10} and all possible 56-bit data v do the following:
    a) Regard v as a data at X'_{10}, then decrypt v using K'_{10} for 1-round.
    b) Let a' be the parity of the decrypted data with mask $\mathit{\Gamma OL}_9$.
    c) For each possible combination of K_{11} and K_{12} do the following:
        i. Increase the C2 counter related to K'_{10}, K_{11}, K_{12} by the number in the C1 counter related to a' and v, K_{11}, K_{12}, and decrease it by the number in C1 related to $a' \oplus 1$ and v, K_{11}, K_{12}.
5. Look over the counters, and choose the one with the maximum absolute value.

The data complexity is 2^{120}, memory complexity is $2^{184} + 2^{185}$ and the time complexity is 2^{248} times 2 round decryption and C1 count up operations and $2^{56} \cdot 2^{56} = 2^{112}$ decryption of 7 S-boxes, and 2^{248} C2 count up operations.

Also we can attack 10-round and 11-round Camellia* using the same 9-round linear approximation. Table 62 summarizes the complexity of these key recovery algorithms.

Table 62. Summery of the complexity of the linear attacks

Round	Key Length	Data	Time	Memory
10	128,192,256	2^{120}	2^{121}MA	$2^{56.6}$
11	192,256	2^{120}	$2^{181.5}$Enc.	$2^{120.6}$
12	256	2^{120}	$2^{245.4}$Enc.	$2^{184.6}$

MA-Memory Access, Enc-Encryptions

6.3 Boomerang and Rectangle Attack on 10-round 256-bit key Camellia

The boomerang attack and the rectangle attack use the differential attack as a building block [69,71,211,344,616]. Let E be a cipher which can be described as a cascade $E = E_f \circ E_1 \circ E_0 \circ E_b$ such that each of E_0 and E_1 has a differential with high probability. Biham *et al.* showed an efficient key recovery algorithm to find the subkey in the E_b and E_f [71].

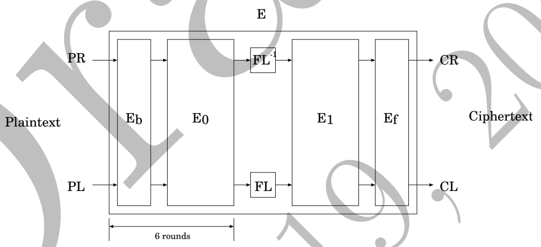


Fig. 74. Building blocks

It is important for this analysis that even though linear functions exist between E_0 and E_1, the functions don't affect on the boomerang and the rectangle attacks. Because the FL/FL^{-1} functions are designed as key dependent linear functions, we can attack Camellia if we can construct a decomposition including FL/FL^{-1} just between E_0 and E_1 as Fig. 74.

We use the following two types of round decomposition for these attacks: (E_b, E_0, E_1, E_f) = (1 round, 5 rounds, 3 rounds, 0 round) and (1 round, 5 rounds, 3 rounds, 1 round) with probabilities $\hat{p} = 2^{-50.98}$ for 5-round E_0 and $\hat{q} = 2^{-9.49}$ for 3-round of E_1, where the input difference α = (0xc1000000000000c1, 0x0000000000010101) and the output difference δ = (0x0100000000000000, 0xf1f1f100f10000f1) shown in Fig. 75. The '$*$' in the figure denotes the difference value which varies according to output difference values of active S-boxes in the last round and the first round, respectively.

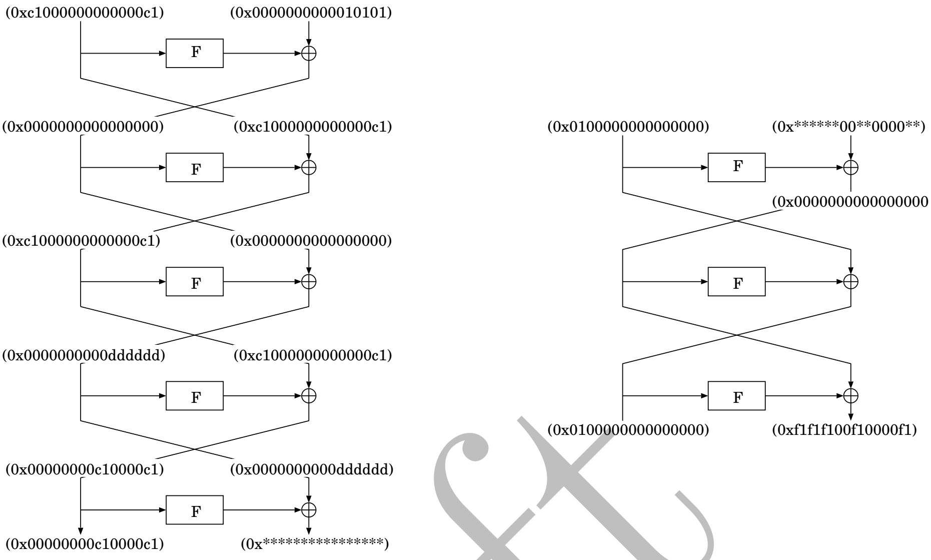


Fig. 75. Detail of 5-round E_0 and 3-round E_1 decompositions

With the above settings, we employ the exactly same steps of the boomerang and the rectangle attacks described in [71]. The complexity of the attacks can be estimated by the parameters $(r_b, t_b, m_b, r_f, t_f, m_f, n, \hat{p}, \hat{q})$ selected appropriately. In this case $n = 128$ and the parameters of E_b and E_f are as follows: The 5 bytes of the right half of α equal 0 and the number of active S-box in E_b is 3, then $r_b = 128 - 8 * 5 = 88, t_b = log_2 127^3 \approx 21, m_b = 8 * 3 = 24$. Similarly we set parameters of attack (r_f, t_f, m_f) of E_f as: $r_f = 104, t_f \approx 34.9, m_f = 40$.

Table 63 summarizes the parameters and complexity of the boomerang attack and the rectangle attack. These results show that 9-round Camellia is attackable by the rectangle and boomerang attacks with 192 and 256 bit keys, and 10-round Camellia is attackable by the rectangle attacks with 256 bit key.

Table 63. Parameters and Complexity of Rectangle and Boomerang Attacks

Round	E_b	E_0	E_1	E_f	r_b	m_b	t_b	r_f	m_f	t_f	$\hat{p}$	$\hat{q}$	Attack	Data	Time	Memory
9	1	5	3	0	88	24	21	0	0	0	$2^{-50.98}$	$2^{-9.49}$	R. B.	$2^{126.45}$ $2^{123.9}$	$2^{170.9}$MA $2^{169.9}$MA	$2^{126.45}$ 2^{72}
10	1	5	3	1	88	24	21	104	40	34.9	$2^{-50.98}$	$2^{-9.49}$	R.	$2^{126.45}$	$2^{240.9}$MA	$2^{126.45}$

MA-Memory Access, R. - Rectangle Attack, B. - Boomerang Attack

7 Conclusions

In this paper, we have proposed a new method to retrieve differential paths and linear approximations and successfully found differential paths with high DCP and linear approximation with high LCP and $\hat{p}, \hat{q}$ effective to the boomerang and rectangle attacks.

Table 64 summarizes previously published attacks applied to 256-bit key Camellia. As a result, we have shown that the linear attack is the largest round attack on Camellia*. And we have also shown that not only algebraic attacks but also probabilistic attacks can work for Camellia by using the boomerang and the rectangle attacks.

Table 64. Comparison of Attackable Rounds

Attack	Rounds	FL/FL^{-1}	Data	Time	Memory	Type
Square [292]	6	No	$2^{11.7}$	2^{112}Enc.	2^{104}	CP
Truncated Differential [392]	8	No	$2^{83.6}$	$2^{55.6}$Enc.	2^{16}	CP
Square [627]	9	Yes	$2^{60.5}$	$2^{202.2}$Enc.	2^{193}	CP
Boomerang (This paper)	9	Yes	$2^{123.9}$	$2^{169.9}$MA	2^{72}	ACPC
Rectangle (This paper)	10	Yes	$2^{126.5}$	$2^{240.9}$MA	$2^{126.45}$	CP
Higher Order Differential [339]	10	No	2^{21}	$2^{254.7}$Enc.	2^{240}	CP
Differential (This paper)	11	No	2^{104}	$2^{231.5}$Enc.	2^{168}	CP
Higher Order Differential [287]	11	Yes/No	2^{93}	$2^{255.6}$Enc.	2^{248}	CC/CP
Linear (This paper)	12	No	$2^{119.1}$	$2^{246.5}$Enc.	$2^{184.6}$	KP

CP-Chosen Plaintext,CC - Chosen Ciphertext, KP -Known Plaintext,
ACPC - Adaptive Chosen Plaintext and Ciphertext,
Enc. - Encryptions, MA-Memory Access

Acknowledgments

We are thankful to Kyoji Shibutani, Shoji Kanamaru and George Abe for interesting discussions and helpful comments.

Efficient FPGA Implementations of Block Ciphers Khazad and MISTY1

by François-Xavier Standaert, Gaël Rouvroy, Jean-Jacques Quisquater and Jean-Didier Legat *

Abstract. The technical analysis used in determining which of the NESSIE candidates will be selected as a standard block cipher includes efficiency testing of both hardware and software implementations of candidate algorithms. Reprogrammable devices such as Field Programmable Gate Arrays (FPGA's) are highly attractive options for hardware implementations of encryption algorithms and this report investigates the significance of FPGA implementations of the block ciphers Khazad and MISTY1. A strong focus is placed on high throughput circuits and we propose designs that unroll the cipher rounds and pipeline them in order to optimise the frequency and throughput results. In addition, we implemented solutions that allow to change the plaintext and the key on a cycle-by-cycle basis with no dead cycle. The resulting designs fit on a VIRTEX1000 FPGA and have throughput between 8 and 9 Gbits/s. This is an impressive result compared with existing FPGA implementations of block ciphers within similar devices.

1 Introduction

The NESSIE project is about to put forward a portfolio of strong cryptographic primitives that has been obtained after an open call and been evaluated using a transparent and open process. These primitives include block ciphers, stream ciphers, hash functions, MAC algorithms, digital signature schemes, and public-key encryption schemes. The technical analysis used in determining which of the NESSIE candidates will be selected as a standard block cipher includes efficiency testing of both hardware and software implementations of candidate algorithms.

NESSIE candidate Khazad is a 64-bit block cipher that accepts a 128-bit key. Although Khazad is not a Feistel cipher, its structure is designed so that by choosing all round transformations components to be involutions, the inverse operation of the cipher differs from the forward operation in the key scheduling part only. This property makes it possible to reduce the required chip area in hardware implementations. The overall cipher design follows the Wide Trail Strategy, favours component reuse, and permits a wide variety of implementation tradeoffs. Encryption algorithm MISTY1 is a 64-bit block cipher with a 128-bit key and a

* UCL Crypto Group
Laboratoire de Microélectronique
Université Catholique de Louvain
Place du Levant, 3, B-1348 Louvain-La-Neuve, Belgium
{standaert,rouvroy,quisquater,legat}@dice.ucl.ac.be

variable number of rounds. Mitsuru Matsui, the designer (1996), recommends a 8-round version. It is a Feistel cipher that allows very efficient hardware implementations. MISTY1 is designed on the basis of the theory of provable security against differential and linear cryptanalysis. In this report, we study the suitability of KHAZAD and MISTY1 for hardware implementations. Fast encryption modules are detailed and compared to AES RIJNDAEL and SERPENT in an effort to determine the efficiency of NESSIE candidates for hardware implementations within commercially available FPGA's.

This report is organised as follows. The description of the hardware, synthesis tools and implementation tools is in section 2. Section 3 gives a short mathematical description of KHAZAD and we propose a description of the diffusion layer that allows efficient pipelining. Our implementations of KHAZAD are in section 4. Section 5 gives a short mathematical description of MISTY1 and the corresponding implementations are in section 6. Comparisons with RIJNDAEL and SERPENT appear in section 7. Finally, conclusions are in section 8.

2 Hardware description

All our implementations were carried out on a XILINX VIRTEX1000BG560-6 FPGA. In this section, we briefly describe the structure of a VIRTEX FPGA as well as the synthesis and implementation tools that were used to obtain our results.

Configurable Logic Blocks (CLB's). The basic building block of the VIRTEX CLB is the logic cell (LC). A LC includes a 4-input function generator, carry logic and a storage element. The output from the function generator in each LC drives both the CLB output and the D input of the flip-flop. Each VIRTEX CLB contains four LC's, organised in two similar slices. Figure 76, shows a detailed view of a single slice. Virtex function generator are implemented as 4-input lookup tables (LUT's). In addition to operate as a function generator, each LUT can provide a 16×1-bit synchronous RAM. Furthermore, the two LUT's within a slice can be combined to create a 16×2-bit or 32×1-bit synchronous RAM or a 16×1-bit dual port synchronous RAM. The VIRTEX LUT can also provide a 16-bit shift register.

The storage elements in the VIRTEX slice can be configured either as edge-triggered D-type flip-flops or as level-sensitive latches. The D inputs can be driven either by the function generators within the slice or directly from slice inputs, bypassing function generators.

The F5 multiplexer in each slice combines the function generator outputs. This combination provides either a function generator that can implement any 5-input function, a 4:1 multiplexer, or selected functions of up to nine bits. Similarly, the F6 multiplexer combines the outputs of all four function generators in the CLB by selecting one of the F5-multiplexer outputs. This permits the implementation of any 6-input function, an 8:1 multiplexer, or selected functions up to 19 bits.

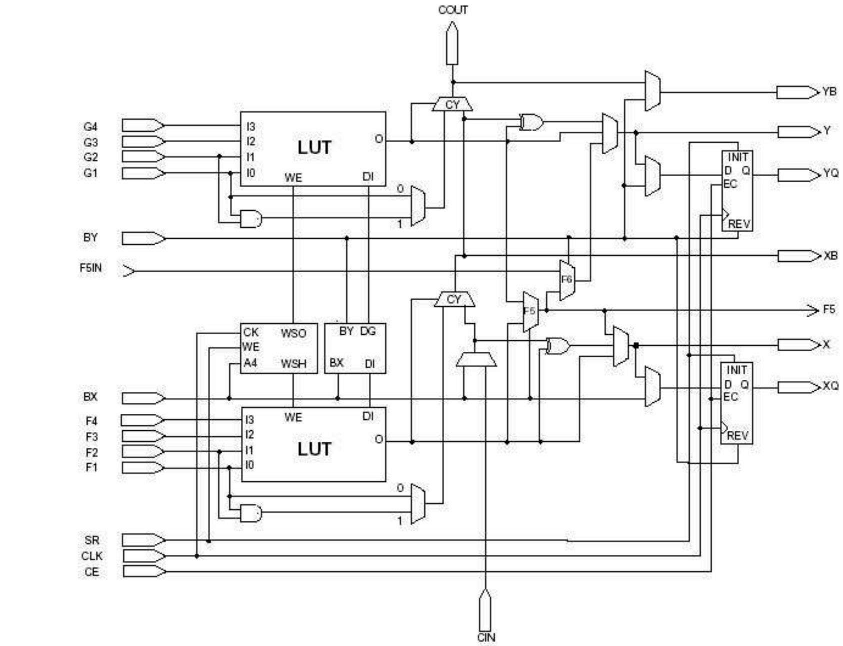


Fig. 76. The VIRTEX slice.

The arithmetic logic also includes a XOR gate that allows a 1-bit full adder to be implemented within an LC. In addition, a dedicated AND gate improves the efficiency of multiplier implementations.

Finally, VIRTEX FPGA's incorporate several large RAM blocks. These complement the distributed LUT implementations of RAM's. Every block is a fully synchronous dual-ported 4096-bit RAM with independent control signals for each port. The data widths of the two ports can be configured independently.

Target FPGA. A VIRTEX1000BG560-6 FPGA contains 12288 slices and 32 RAM blocks, which means 24576 LUT's and 24576 flip-flops. In the following report, we compare the number of LUT's, registers and slices. We also evaluate the delays and frequencies thanks to our synthesis and implementation tools. The synthesis was performed with FPGA Express (SYNOPSYS) and the implementation with XILINX ISE-4. Finally, our circuits models were described using VHDL.

3 Block cipher description: Khazad

Khazad is an iterated block cipher that operates on a 64-bit cipher state represented as vectors in $GF(2^8)^8$. It uses a 128-bit key represented as a vector in $GF(2^8)^{16}$, and consists of a series of applications of a key-dependent round transformation to the cipher state. In the following, we will individually define the component mappings and constants that build up Khazad, then specify the complete cipher in terms of these components.

Notation. Let s be a cipher state or a key $\in GF(2^8)^8$, then s_i is the i-th byte of the state s and $s_i(j)$ is the j-th bit of this byte.

The nonlinear layer $\boldsymbol{\gamma}$. Function $\gamma : GF(2^8)^8 \rightarrow GF(2^8)^8$ consist of the parallel application of a non-linear substitution box S:

$$\gamma(a) = b \Leftrightarrow b_i = S[a_i], 0 \leq i \leq 7 \tag{7}$$

The substitution box is illustrated on figure 77, where P and Q are 4-bit input × 4-bit output look up tables. They were defined in order to optimise the resistance against differential and linear cryptanalysis and allow efficient hardware implementations.

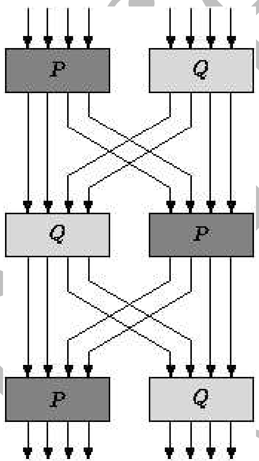


Fig. 77. The Khazad substitution box.

The diffusion layer $\boldsymbol{\theta}$. Function $\theta : GF(2^8)^8 \rightarrow GF(2^8)^8$ is a linear mapping based on a $[16, 8, 9]$ MDS[1] code:

$$\theta(a) = b \Leftrightarrow b = a.H \tag{8}$$

With:

$$H = \begin{bmatrix} 01 & 03 & 04 & 05 & 06 & 08 & 0B & 07 \\ 03 & 01 & 05 & 04 & 08 & 06 & 07 & 0B \\ 04 & 05 & 01 & 03 & 0B & 07 & 06 & 08 \\ 05 & 04 & 03 & 01 & 07 & 0B & 08 & 06 \\ 06 & 08 & 0B & 07 & 01 & 03 & 04 & 05 \\ 08 & 06 & 07 & 0B & 03 & 01 & 05 & 04 \\ 0B & 07 & 06 & 08 & 04 & 05 & 01 & 03 \\ 07 & 0B & 08 & 06 & 05 & 04 & 03 & 01 \end{bmatrix}$$

[1] MDS: Maximum Distance Separable.

We propose the following description of the diffusion layer that allows to introduce pipeline levels inside the layer:

$$b_0 = a_0 \oplus a_1 \oplus a_3 \oplus a_6 \oplus a_7 \oplus X(a_1 \oplus a_4 \oplus a_6 \oplus a_7) \oplus X^2(a_2 \oplus a_3 \oplus a_4 \oplus a_7) \oplus X^3(a_5 \oplus a_6)$$
$$b_1 = a_0 \oplus a_1 \oplus a_2 \oplus a_6 \oplus a_7 \oplus X(a_0 \oplus a_5 \oplus a_6 \oplus a_7) \oplus X^2(a_2 \oplus a_3 \oplus a_5 \oplus a_6) \oplus X^3(a_4 \oplus a_7)$$
$$b_2 = a_1 \oplus a_2 \oplus a_3 \oplus a_4 \oplus a_5 \oplus X(a_3 \oplus a_4 \oplus a_5 \oplus a_6) \oplus X^2(a_0 \oplus a_1 \oplus a_5 \oplus a_6) \oplus X^3(a_4 \oplus a_7)$$
$$b_3 = a_0 \oplus a_2 \oplus a_3 \oplus a_4 \oplus a_5 \oplus X(a_2 \oplus a_4 \oplus a_5 \oplus a_7) \oplus X^2(a_0 \oplus a_1 \oplus a_4 \oplus a_7) \oplus X^3(a_5 \oplus a_6)$$
$$b_4 = a_2 \oplus a_3 \oplus a_4 \oplus a_5 \oplus a_7 \oplus X(a_0 \oplus a_2 \oplus a_3 \oplus a_5) \oplus X^2(a_0 \oplus a_3 \oplus a_6 \oplus a_7) \oplus X^3(a_1 \oplus a_2)$$
$$b_5 = a_2 \oplus a_3 \oplus a_4 \oplus a_5 \oplus a_6 \oplus X(a_1 \oplus a_2 \oplus a_3 \oplus a_4) \oplus X^2(a_1 \oplus a_2 \oplus a_6 \oplus a_7) \oplus X^3(a_0 \oplus a_3)$$
$$b_6 = a_0 \oplus a_1 \oplus a_5 \oplus a_6 \oplus a_7 \oplus X(a_0 \oplus a_1 \oplus a_2 \oplus a_7) \oplus X^2(a_1 \oplus a_2 \oplus a_4 \oplus a_5) \oplus X^3(a_0 \oplus a_3)$$
$$b_7 = a_0 \oplus a_1 \oplus a_4 \oplus a_6 \oplus a_7 \oplus X(a_0 \oplus a_1 \oplus a_3 \oplus a_6) \oplus X^2(a_0 \oplus a_3 \oplus a_4 \oplus a_5) \oplus X^3(a_1 \oplus a_2)$$

Where $b_7, b_6, ..., b_0$ represent the eight bytes of the cipher state and X is defined at the byte level as: $X : GF(2^8) \rightarrow GF(2^8) : X(a) = b \Leftrightarrow$

$$\begin{aligned} b(7) &= a(6) \\ b(6) &= a(5) \\ b(5) &= a(4) \\ b(4) &= a(3) \oplus a(7) \\ b(3) &= a(2) \oplus a(7) \\ b(2) &= a(1) \oplus a(7) \\ b(1) &= a(0) \\ b(0) &= 0 \oplus a(7) \end{aligned}$$

Finally, we define functions $X^2 \equiv X \circ X$ and $X^3 \equiv X \circ X \circ X$.

The key addition $\boldsymbol{\sigma}$. The affine key addition $\sigma[k] : GF(2^8)^8 \rightarrow GF(2^8)^8$ consists of the bitwise addition (exor) of a key vector $k \in GF(2^8)^8$:

$$\sigma[k](a) = b \Leftrightarrow b_i = a_i \oplus k_i, 0 \leq i \leq 7 \tag{9}$$

The round constants. The constant for the r-th round is a vector $c^r \in GF(2^8)^8$, defined as:

$$c_i^r = S[8r + i], 0 \leq r \leq 8, 0 \leq i \leq 7 \tag{10}$$

The round function $\boldsymbol{\rho}$. The r-th round function is the composite mapping $\rho[k] : GF(2^8)^8 \rightarrow GF(2^8)^8$, parameterised by the key vector $k \in GF(2^8)^8$ and given by:

$$\rho[k] \equiv \sigma[k] \circ \theta \circ \gamma \tag{11}$$

The key schedule. The key schedule expands the cipher key $K \in GF(2^8)^{16}$ into a sequence of round keys $K^0, K^1, ...; K^8$, plus two initial values K^{-2} and K^{-1} corresponding to the most and least significant parts of the cipher key K. Every round key is an element of $GF(2^8)^8$ that we derive as follows:

$$K^r = \rho[c^r](K^{r-1}) \oplus K^{r-2}, 0 \leq r \leq 8 \tag{12}$$

The complete cipher. Khazad is defined for the cipher key K as the transformation Khazad$[K] = \alpha[K^0, K^1, ..., K^8]$ applied to the plaintext, where

$$\alpha[K^0, K^1, ..., K^8] = \sigma[K^8] \circ \gamma \circ (\bigcirc_{r=1}^{7} \rho[K^r]) \circ \sigma[K^0] \tag{13}$$

Our implementation is based on this description of Khazad.

4 Implementation: Khazad

4.1 Objectives

FPGA's are very efficient devices and they are suitable for high work frequencies. As opposed to custom hardware or software implementations, little work exist in the area of block cipher implementations within existing FPGA's. Results available in the public literature sometimes mention encryption rates comparable with software ones. We believe that these performances can be greatly improved using today's technology as soon as inherent constraints of FPGA's are taken into account.
The VIRTEX slice offers great flexibility to implement various logic functions, but it also constraints the designer to an efficient usage of its resources. Regarding the inner structure of Khazad, we determined that an optimal circuit should limit its critical path inside one slice, without consuming slices for register usage only.

4.2 Components

Table 65 evaluates the hardware cost of some basic elements of Khazad. Their structure, very close to the VIRTEX LUT, allow a direct and efficient implementation. Practically, we optimised our circuit by keeping its critical path inside the slice of Fig. 76, and making an efficient use of its registers.

Table 65. Some combinatorial components of Khazad.

Component	Nbr of LUT
X	3
S	24
γ layer	192
Key addition σ	64

Actually, the most critical function in terms of implementation is the diffusion layer θ. In the precedent section, we gave a combinatorial description of it that allow us to consider different pipeline levels. For efficiency purposes, we also combined θ with the key addition layer σ, because of their relevant compatibility. Figure 78 illustrates the computation of an output byte b_0 of the diffusion layer θ combined with key addition σ. The key point of this architecture is the central

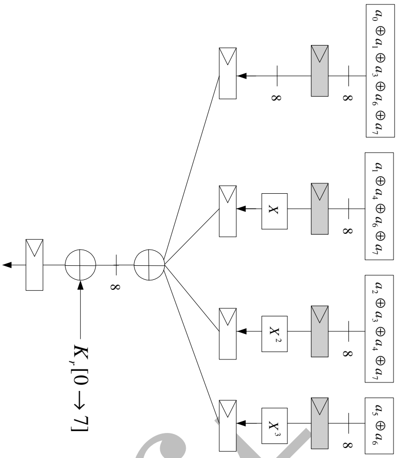

$a_0 \oplus a_1 \oplus a_3 \oplus a_6 \oplus a_7$
$a_1 \oplus a_4 \oplus a_6 \oplus a_7$
$a_2 \oplus a_3 \oplus a_4 \oplus a_7$
$a_5 \oplus a_6$
8
X
X^2
X^3
$K_r[0 \rightarrow 7]$

Fig. 78. KHAZAD: output byte b_0 of the diffusion layer θ combined with key addition σ.

bitwise XOR operation between 4 bytes. As the VIRTEX slice contains, a 4-input LUT and an additional XOR gate, we can efficiently combine this operation with the bitwise key addition and perform the resulting task in one cycle.

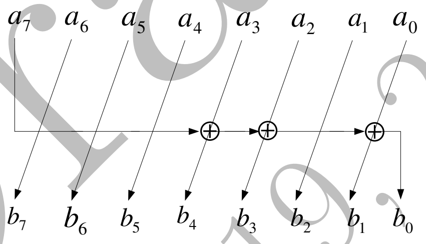

a_7 a_6 a_5 a_4 a_3 a_2 a_1 a_0
b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0

Fig. 79. The function X of KHAZAD.

The upper part of θ don't permit this kind of optimisation. Looking at Figures 78, 79, we see that the byte $a_2 \oplus a_3 \oplus a_4 \oplus a_7$ is an input of function $X^2 = X \circ X$. This can't be done inside one slice. Consequently, we considered two circuits depending on the diffusion-addition layer implemented:

1. A fast and expensive implementation using three registers levels inside the layer.
2. A slower but less expensive implementation where the grey register of θ is removed.

A tradeoff has to be done between a low-delay (pipeline of X, X^2, X^3 functions) and a low area where we avoid the implementation of useless registers in the left branch of θ.

Table 66. Khazad: implementations of the diffusion layer θ combined with key addition σ.

Type	Nbr of LUT	Nbr of registers	Estimated delay (ns)
Fast implementation	384	576	5.2
Low area implementation	384	320	7.3

Table 66 gives the implementation results of our two implementations. In this section, the delay is estimated after synthesis[2]. Note that in the fast implementation, the critical path corresponds to a look up table and an exor operation in the left branch of θ.

4.3 The round and key round functions

Based on the above components, we propose two solutions for the round and key round functions, with a difference of one register level. Figure 80 illustrates the round function of Khazad. Figure 81 illustrates its key round. Depending on the use of the grey register, we obtain the results of table 67.

Table 67. Khazad: implementations of the round and key round.

Type	Nbr of LUT	Nbr of registers	Estimated delay (ns)
Fast round	576	768	5.5
Low area round	576	512	7.3
Fast key round	768	832	5.6
Low area key round	756	576	7.3

4.4 The complete cipher

The implementation of the complete Khazad cipher directly results from the precedent descriptions. Our results are summarised by the next figure and table: Figure 82 illustrates our two versions of the complete cipher. Finally, table 68 summarises our implementation results for the block cipher Khazad. In this section, the frequency is estimated after synthesis[3] and implementation[4].

Table 68. Implementations of Khazad.

Type	Nbr of LUT	Nbr of registers	Nbr of slices	Latency (cycles)	Output every (cycles)	Freq. after Synt. (MHz)	Freq. after Impl. (MHz)
Fast Khazad	11328	13568	8800	62	1	175	148
Low area Khazad	11072	9600	7175	53	1	137	123

[2] FPGA Express (SYNOPSYS).

[3] FPGA Express (SYNOPSYS).

[4] Xilinx ISE4.

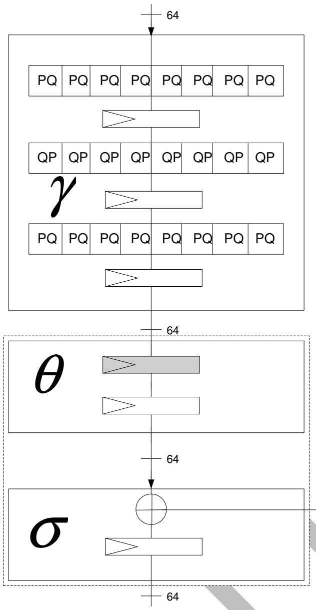


Fig. 80. The KHAZAD round function ρ.

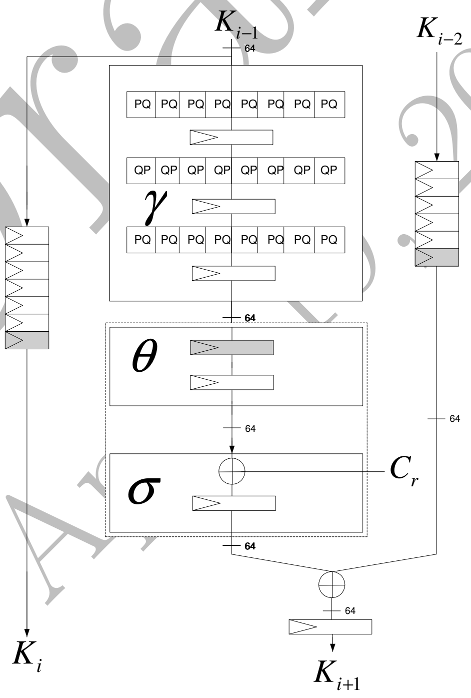


Fig. 81. The KHAZAD keyround.

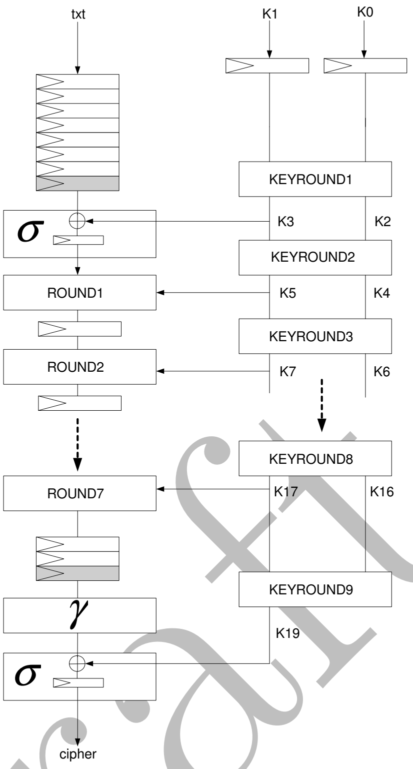


Fig. 82. KHAZAD.

From these results, we observe very high frequencies after synthesis. However critical delays mainly occurs when trying to place and route these synthesis results. The resulting implemented designs have surprising critical paths including 20% of logic and 80% of routes. We conclude that the real bottleneck of such large ciphers is in the difficulty of having an efficient place and route. Actually, constraints come from shift registers and high fanout. Implementation could probably be improved by replacing the last stage of shift registers by flip-flops, but the additional degree of freedom for the routes would be balanced with additional resources. Anyway, the resulting designs are very efficient as we will underline in section 7.

5 Block cipher description: MISTY1

MISTY1 is an iterated block cipher that operates on a 64-bit block with a 128-bit key and with a variable number of rounds n. We describe the algorithm with $n = 8$, as recommended in [425, 426]. In the following subsections, we describe the data randomising part and the key scheduling part of MISTY1 with their different components.

5.1 Data randomising part

Figure 84 shows the data randomising part of MISTY1[5]. The 64-bit plaintext P is divided in two 32-bit parts. Both parts are transformed into the 64-bit ciphertext using bitwise XOR operations, sub-functions FO_i $(1 \leq i \leq (n = 8))$ and sub-functions FL_i $(1 \leq i \leq (n+2 = 10))$.

***FO*$_i$ function.** Figure 83 shows the structure of FO_i[6]. This function split the input into two 16-bit strings. Then, it transforms both strings into the output with bitwise XOR operations and sub-functions FI_{ij} $(1 \leq j \leq 3)$. KO_{ij} $(1 \leq j \leq 4)$ and KI_{ij} $(1 \leq j \leq 3)$ are the left j-th 16 bits of KO_i and KI_i, respectively.

***FI*$_{ij}$ function.** Figure 83 also shows the structure of FI_{ij}. The input is divided into two parts: a 9-bit string and a 7-bit string. These strings are transformed into the output using bitwise XOR operations and substitutions tables S_7 and S_9. In the beginning and the end of FI_{ij} function, the 7-bit string is zero-extend to 9 bits, and in the middle part, the 9-bit string is truncated to 7 bits eliminating its highest two bits (MSB). KI_{ij1} and KI_{ij2} are the left 7 bits and the right 9 bits of KI_{ij}, respectively.

***FL*$_i$ function.** The structure of FL_i function is illustrated on figure 83. The 32-bit input is divided into two equal parts. The function transforms both parts into the output with bitwise AND, OR and XOR operations. KL_{ij1} $(1 \leq j \leq 2)$ is the left j-th 16 bits of KL_i.

S_7 and S_9 substitution functions. For the selection of S_7 and S_9 substitution functions, Matsui considers three criteria:

1. Their average differential/linear probability must be minimal,
2. Their delay time in hardware is as short as possible,
3. Their algebraic degree is high, if possible.

Based on these criteria, for the S_7 substitution function, Matsui chooses the following mathematical description:

$y_0 = x_0 + x_1x_3 + x_0x_3x_4 + x_1x_5 + x_0x_2x_5 + x_4x_5 + x_0x_1x_6 + x_2x_6 + x_0x_5x_6 + x_3x_5x_6 + 1$
$y_1 = x_0x_2 + x_0x_4 + x_3x_4 + x_1x_5 + x_2x_4x_5 + x_6 + x_0x_6 + x_3x_6 + x_2x_3x_6 + x_1x_4x_6 + x_0x_5x_6 + 1$
$y_2 = x_1x_2 + x_0x_2x_3 + x_4 + x_1x_4 + x_0x_1x_4 + x_0x_5 + x_0x_4x_5 + x_3x_4x_5 + x_1x_6x_3x_6 +$
$x_0x_3x_6 + x_4x_6 + x_2x_4x_6$
$y_3 = x_0 + x_1 + x_0x_1x_2 + x_0x_3 + x_2x_4 + x_1x_4x_5 + x_2x_6 + x_1x_3x_6 + x_0x_4x_6 + x_5x_6 + 1$
$y_4 = x_2x_3 + x_0x_4 + x_1x_3x_4 + x_5 + x_2x_5 + x_1x_2x_5 + x_0x_3x_5 + x_1x_6 + x_1x_5x_6 + x_4x_5x_6 + 1$
$y_5 =$
$x_0 + x_1 + x_2 + x_0x_1x_2 + x_0x_3 + x_1x_2x_3 + x_1x_4 + x_0x_2x_4 + x_0x_5 + x_0x_1x_5 + x_3x_5 + x_0x_6 + x_2x_5x_6$
$y_6 = x_0x_1 + x_3 + x_0x_3 + x_2x_3x_4 + x_0x_5 + x_2x_5 + x_3x_5 + x_1x_3x_5 + x_1x_6 + x_1x_2x_6 +$
$x_0x_3x_6 + x_4x_6 + x_2x_5x_6$

Based on the same above criteria, the S_9 function is defined as:

[5] Where registers needed for efficiency purposes are already mentioned.

[6] Where registers needed for efficiency purposes are already mentioned.

$$
\begin{aligned}
y_0 &= x_0x_4 + x_0x_5 + x_1x_5 + x_1x_6 + x_2x_6 + x_2x_7 + x_3x_7 + x_3x_8 + x_4x_8 + 1\\
y_1 &= x_0x_2 + x_3 + x_1x_3 + x_2x_3 + x_3x_4 + x_4x_5 + x_0x_6 + x_2x_6 + x_7 + x_0x_8 + x_3x_8 + x_5x_8 + 1\\
y_2 &= x_0x_1 + x_1x_3 + x_4 + x_0x_4 + x_2x_4 + x_3x_4 + x_4x_5 + x_0x_6 + x_5x_6 + x_1x_7 + x_3x_7 + x_8\\
y_3 &= x_0 + x_1x_2 + x_2x_4 + x_5 + x_1x_5 + x_3x_5 + x_4x_5 + x_5x_6 + x_1x_7 + x_6x_7 + x_2x_8 + x_4x_8\\
y_4 &= x_1 + x_0x_3 + x_2x_3 + x_0x_5 + x_3x_5 + x_6 + x_2x_6 + x_4x_6 + x_5x_6 + x_6x_7 + x_2x_8 + x_7x_8\\
y_5 &= x_2 + x_0x_3 + x_1x_4 + x_3x_4 + x_1x_6 + x_4x_6 + x_7 + x_3x_7 + x_5x_7 + x_6x_7 + x_0x_8 + x_7x_8\\
y_6 &= x_0x_1 + x_3 + x_1x_4 + x_2x_5 + x_4x_5 + x_2x_7 + x_5x_7 + x_8 + x_0x_8 + x_4x_8 + x_6x_8 + x_7x_8 + 1\\
y_7 &= x_1 + x_0x_1 + x_1x_2 + x_2x_3 + x_0x_4 + x_5 + x_1x_6 + x_3x_6 + x_0x_7 + x_4x_7 + x_6x_7 + x_1x_8 + 1\\
y_8 &= x_0 + x_0x_1 + x_1x_2 + x_4 + x_0x_5 + x_2x_5 + x_3x_6 + x_5x_6 + x_0x_7 + x_0x_8 + x_3x_8 + x_6x_8 + 1
\end{aligned}
$$

Both substitution boxes are defined as ROM tables in [425]. To optimise the number of logic cells used in FPGA implementations, we prefer to implement S_7 and S_9 functions directly as logical expressions. With enough pipelined stages, we keep the critical path of the design under control.

5.2 Key scheduling part

Figure 85 shows the key scheduling part of MISTY1[7]. K_i (1 $\leq$ i $\leq$ 8) is the left i-th 16 bits of the secret input key K. $K_i^{'}$ ($1 \leq i \leq 8$) corresponds to the output of FI_{ij} where the input of FI_{ij} is assigned to K_i and the key KI_{ij} is set to $K_{(i+1)mod8}$.

The assignment between key scheduling subkeys K_i/ $K_i^{'}$ and the round subkeys KO_{ij}, KI_{ij}, KL_{ij} is defined as follows, where i equals to $(i-8)$ when $(i > 8)$:

Table 69. Subkeys distribution.

Encrypt Round	KO_{i1}	KO_{i2}	KO_{i3}	KO_{i4}	KI_{i1}	KI_{i2}	KI_{i3}	KL_{i1}	KL_{i2}
Key round	K_i	K_{i+2}	K_{i+7}	K_{i+4}	$K^{'}_{i+5}$	$K^{'}_{i+1}$	$K^{'}_{i+3}$	$K_{\frac{i+1}{2}}(odd.i)$ $K^{'}_{\frac{i}{2}+1}(even.i)$	$K^{'}_{\frac{i+1}{2}+6}(odd.i)$ $K_{\frac{i}{2}+4}(even.i)$

This concludes the mathematical description of MISTY1 algorithm. The next section explains our FPGA design choices in order to be efficient in term of speed and resources used.

6 Implementation: MISTY1

In order to achieve the fastest FPGA implementation of MISTY1, we decided to limit the critical path to only one 4-input LUT and routes. Consequently, we do not use additional XOR's and multiplexors F5, F6 available in the VIRTEX slice. However, these functions could be used in order to reduce the area requirements of MISTY1.

Based on this delay constraint, we modified the mathematical description of the algorithm in order to regroup a maximum number of functions in a minimum number of 4-input LUT's. This strategy leads to very fast designs.

[7] Where registers needed for efficiency purposes are already mentioned.

6.1 S_7 and S_9 implementations

For S_7 and S_9 implementations, we used the logical expressions in place of substitution tables in order to reduce the number of logic cells used. The logical functions have to be pipelined in order to limit the critical path to only one 4-input LUT and routes. For S_7 and S_9, we designed two 2-stage pipelined versions. The next table shows the results that we obtained after synthesis:

Table 70. MISTY1: S_7 and S_9 synthesis results.

Component	Nbr of LUT's	Nbr of FF's	Nbr of pipelined stages
S_7	45	45	2
S_9	44	35	2

6.2 FO_i, FI_{ij}, FL_i implementations

Figure 83 details how we implemented FO_i, FI_{ij}, FL_i functions in order to limit the critical path to only one 4-input LUT and routes. As mentioned on the figure, we have to put an additional output pipeline stage into FI_{ij} function in order to correspond with the key scheduling part. The next table shows the results that we obtained after synthesis:

Table 71. MISTY1: FO_i, FI_{ij}, FL_i synthesis results.

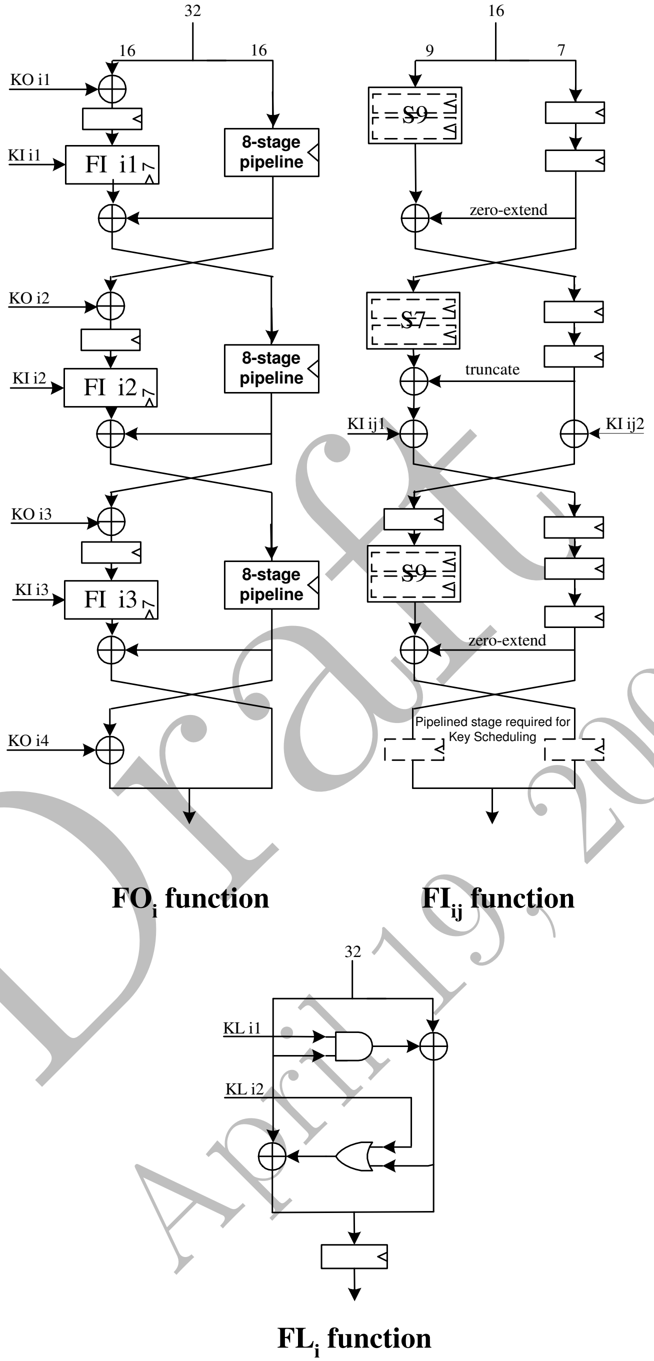
Component	Nbr of LUT's	Nbr of FF's	Nbr of pipelined stages
FO_i	565	633	24
FI_{ij}	158	195	7
FL_i	32	32	1

6.3 The data randomising part of MISTY1

For the same delay constraints, we obtained the design detailed in figure 84. Additional registers for input and output bits are needed to increase the speed performances and these are packed into IOBs. We finally get a 208-stage pipelined design.

6.4 The key scheduling part of MISTY1

Figure 85 shows the key scheduling part of MISTY1. Additional registers for input key bits are also packed into IOBs in order to increase performances. The assignment between key scheduling subkeys K_i/ $K_i^{'}$ and the round subkeys KO_{ij}, KI_{ij}, KL_{ij} is defined in table 5. We do the same in hardware putting the correct



FO_i function

FI_{ij} function

FL_i function

Fig. 83. MISTY1: FO_i, FI_{ij}, FL_i hardware designs.

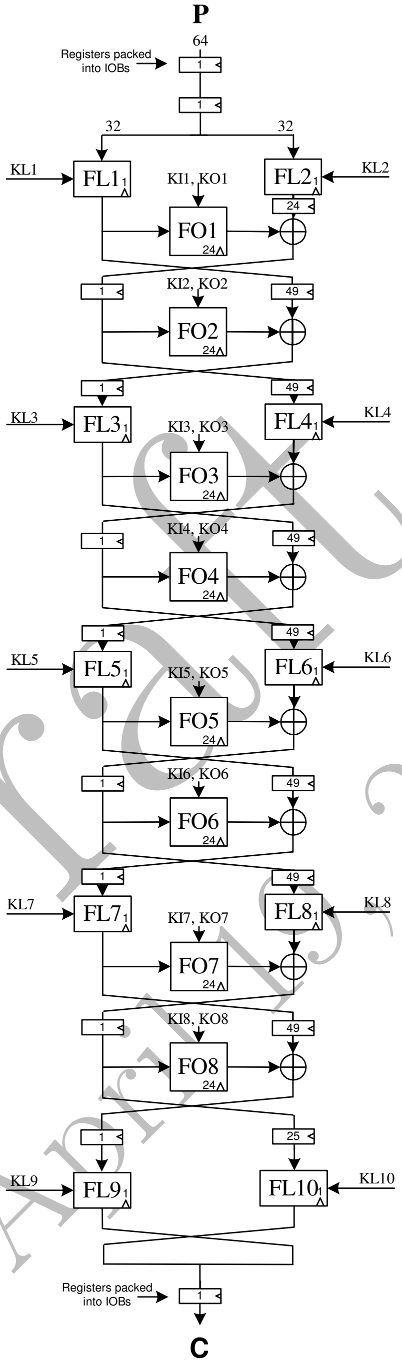


Fig. 84. The data randomising part of MISTY1.

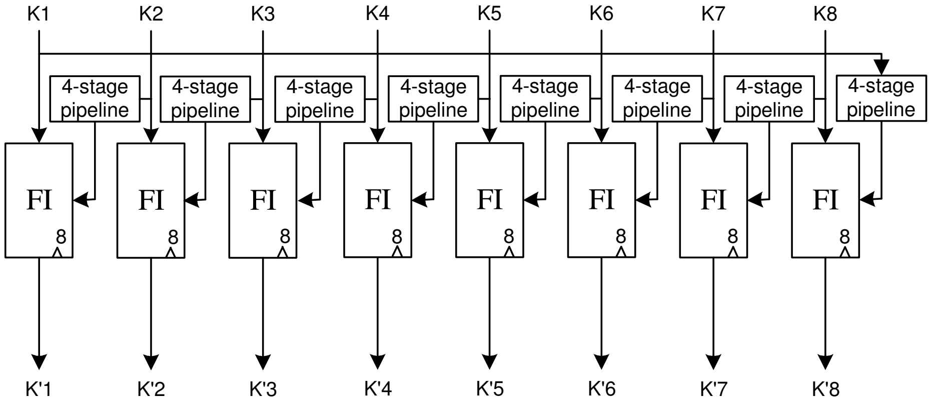


Fig. 85. MISTY1: Key Scheduling.

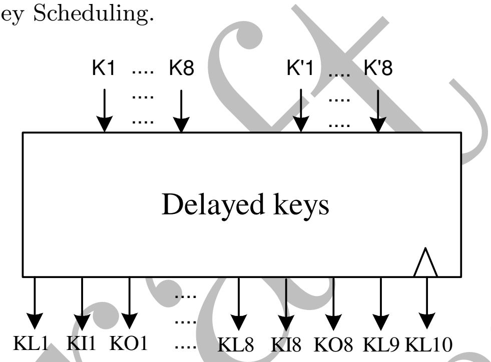


Fig. 86. MISTY1: Subkeys distribution.

number of pipelined stages for every round subkeys. Therefore, to achieve the key distribution, we use 16-bit shift registers, every one fitting in one LUT. Figure 86 represents the subkeys distribution.

Table 72 summarises the result that we obtained for the complete key scheduling part.

Table 72. MISTY1: Key scheduling synthesis results.

Nbr of LUT's	Nbr of FF's	Nbr of pipelined stages
4912	2352	208

6.5 The complete cipher

The complete MISTY1 combines the data randomising part and the key scheduling part from the precedent descriptions. Our results are summarised in table 73 where the latency is the number of pipelined stages. We propose a post-map[8]

[8] XILINX ISE4.

and a post-implementation[9] estimated frequency. The second one takes the routing delays into account. From this result, we observe a very high frequency after mapping phase but the final design only runs at 140 MHz. As we observed for Khazad, critical delays are mainly caused by the routing task. Because the critical path was limited to one 4-input LUT, the problem is even more critical than for Khazad. We conclude that:

Table 73. Implementation of MISTY1.

Nbr of LUT	Nbr of registers	Nbr of slices	Latency (cycles)	Output every (cycles)	Frequency (MHz) Post-map	Frequency (MHz) Post-implementation
10920	8480	8386	208	1	204	140

1. It is not so easy to deal with routing delays. They are no systematic tools to prevent these ones. All we can do, is to locate the problem and to redesign the global circuit. Nevertheless, routing problems usually come from shift registers and high fanout. Implementation could probably be improved by replacing the last stage of shift registers by flip-flops.
2. Trying to reduce the critical path to only one 4-input LUT is not always the best choice. Indeed, if a big part of the critical path is due to route, the use of additional XORs, F5,F6 can reduce the number of logic cells used without increasing the critical path.

Anyway, the resulting design is very efficient and suitable for FPGA, as proved in the next section.

7 Comparison with AES RIJNDAEL, SERPENT and MISTY1

In order to evaluate our implementation results and the hardware suitability of Khazad and MISTY1, we compare them with similar results obtained with the Advanced Encryption Standard RIJNDAEL and SERPENT [218]. We chose RIJNDAEL because of its status of new encryption standard and SERPENT because it seems that it was the best AES candidate regarding FPGA implementations. However, comparisons between Khazad and MISTY1 seem to be more relevant because they were implemented using the same methodology.
In [218], the Xilinx VIRTEX1000BG560-4 was selected as the target device for evaluation of AES candidates. Table 74 compares RIJNDAEL, SERPENT, MISTY1 and Khazad encryption circuits in terms of hardware cost, frequency and throughput for VIRTEX1000bg560-6 component. The hardware cost in LUT and registers is replaced by a number of slices. We also investigate the ratio Throughput/Area which is a good measurement of hardware efficiency.

[9] XILINX ISE4.

Table 74. Comparisons with RIJNDAEL, SERPENT and MISTY1.

Type	Nbr of slices	Output every (clk edges)	Estimated frequency (MHz)	Throughput (Mbits/s)	Throughput/Area (Mbits/s/slices)
RIJNDAEL [218]	10992	2.1	31.8	1938	0.18
SERPENT [218]	9004	1	38	4860	0.54
MISTY1	8386	1	140	8960	1.07
Fast Khazad	8800	1	148	9472	1.07
Low area Khazad	7175	1	123	7872	1.09

8 Conclusions

We propose efficient FPGA implementations of block ciphers Khazad and MISTY1. The structure of Khazad is very close to AES RIJNDAEL and offers comparable security. However, improvements have been done concerning implementation aspects and these allow very efficient FPGA implementations for high throughput applications. Although its keyround is still very expensive, Khazad is a very suitable block cipher for FPGA implementation in the context described for these experiments. MISTY1 offers similar performances and its main bottleneck is to be found in the routing delays. By avoiding these problems, we could greatly improve the design frequency.

Upon comparison, our implementations of Khazad and MISTY1 offer better results than those reported for RIJNDAEL and SERPENT in [218], but the implementation of RIJNDAEL with the design methodology described in this paper will deserve a forthcoming work and allow more relevant comparisons.

Concerning Khazad and MISTY1, we believe that both ciphers have interesting properties for hardware implementations. MISTY1 offers slight advantages in terms of hardware cost: it has the Feistel structure and low-cost substitution boxes. Its key scheduling is also less expansive than Khazad. However, the intensive use of shift registers to pipeline the Feistel network makes the algorithm structure more complex and the design more difficult to route. It results in a larger latency. Looking at the final results, the ratio *Throughput/Area* illustrates that both ciphers are very close and sufficiently efficient but potential improvements exist for MISTY1. It seems that reconfigurable hardware implementations will not be the bottleneck for the selection of Khazad or MISTY1 as a NESSIE cipher.

High Probability Linear Hulls in Q

by Liam Keliher, Henk Meijer and Stafford E. Tavares *

Abstract. In this paper, we demonstrate that the linear hull effect is significant for the Q cipher. The designer of Q performs preliminary linear cryptanalysis by discussing linear characteristics involving only a single active bit at each stage [434]. We present a simple algorithm that combines all such linear characteristics with identical first and last masks into a linear hull. The expected linear probability of the best such linear hull over 7.5 rounds (8 full rounds minus the first S-substitution) is $2^{-90.1}$. In contrast, the best known expected differential probability over the same rounds is $2^{-110.5}$ [73]. Choosing a sequence of linear hulls yields a straightforward attack that can recover a 128-bit key with success rate 98.4%, using 2^{97} known $\langle \text{plaintext}, \text{ciphertext} \rangle$ pairs and 2^{32} trial encryptions.

Keywords. Block cipher, Q, NESSIE, linear cryptanalysis, linear hulls.

1 Introduction

Q is a block cipher submitted to the NESSIE project by Leslie McBride [434]. Q has a straightforward SPN structure with s-boxes based on those in Rijndael (the AES winner) [470] and Serpent [17]. (The Serpent-like s-boxes can be implemented with an efficient bit-slicing technique [17]; for clarity, we will use the equivalent representation that involves bitwise permutations before and after application of these s-boxes).

The structure of the s-boxes and linear transformations allows the construction of linear characteristics with one active bit in each mask. We refer to such linear characteristics as *restricted characteristics*. Nyberg's *linear hull* concept [487] (the counterpart of differentials in differential cryptanalysis [388]) allows us to combine a large number of restricted linear characteristics into a single linear hull that can then be used to attack the cipher.

* lkeliher@mta.ca
Department of Mathematics and Computer Science. Mount Allison University, Sackville, New Brunswick, Canada, E4L 1E6.

henk@cs.queensu.ca
School of Computing. Queen's University at Kingston, Ontario, Canada, K7L 3N6.

tavares@ee.queensu.ca
Department of Electrical and Computer Engineering. Queen's University at Kingston, Ontario, Canada, K7L 3N6.

We present a simple algorithm for calculating the *expected linear probability* (ELP) of the linear hulls formed by this method over various numbers of rounds. The best such linear hull over 7.5 rounds (8 full rounds minus the first AES s-box substitution) has ELP $2^{-90.1}$. In contrast, the best known *expected differential probability* (EDP) over the same rounds is $2^{-110.5}$ [73].

2 Description of Q

2.1 Basic Components

The Q cipher is based on the substitution-permutation network (SPN) architecture [226,296,342,343]. Q has a block size of $N = 128$ bits. Q uses three different s-boxes, one 8×8 s-box named S (this is the AES s-box [470]), and two 4×4 s-boxes named A and B (B is used in Serpent [17], and A is "Serpent-like"). Each substitution stage uses multiple copies of a single s-box in parallel to process the 128-bit input (16 copies of S, or 32 copies of A or B).

Before continuing, we need to clarify the convention used for numbering consecutive bytes and words, namely that numbering begins at 0 with the object in the lowest memory location—this is also the least significant object, since Q uses "little-endian" ordering. This convention extends to numbering the bits of bytes/nibbles, i.e., the least significant bit is numbered 0. Pictorially, numbering always increases from left to right (it follows that the bits in a 128-bit block are numbered $0 \ldots 127$, left to right).

Three linear transformations are used in the cipher. The permutation P operates on a 128-bit block represented as four 32-bit words, W_0, W_1, W_2, W_3, as follows: W_0 is unchanged; W_1, W_2, andW_3 are right rotated by one byte, two bytes, and three bytes, respectively.

The other two linear transformations are bitwise permutations that we term PreSerpent( ) and PostSerpent( ), since they are located before and after each application of s-boxes A and B. If we again view the 128-bit block as consisting of words W_0, W_1, W_2, W_3, PreSerpent( ) sends the bits of W_0 to the first (leftmost) input bits of the 32 identical 4×4 s-boxes, the bits of W_1 to the second input bits of these s-boxes, and so on. This is represented in Figure 87. PostSerpent( ) is simply the inverse of PreSerpent( ).

2.2 High-Level Structure

Q accepts keys of any length, although keys longer than 256 bits are shortened to 256 bits. We will consider the version of Q that consists of 8 "full rounds" and uses a key of at most 128 bits. (A modified form of our attack can be applied for keys longer than 128 bits.) McBride also proposed a 9-round version of Q for "high security applications" [434].

The Q key-scheduling algorithm generates twelve 128-bit subkeys named $KW1$, KA, KB, $K0, K1, \ldots, K7$, $KW2$. Only two details of the key-scheduling algorithm are important for our attack. First, although KA and KB are 128 bits

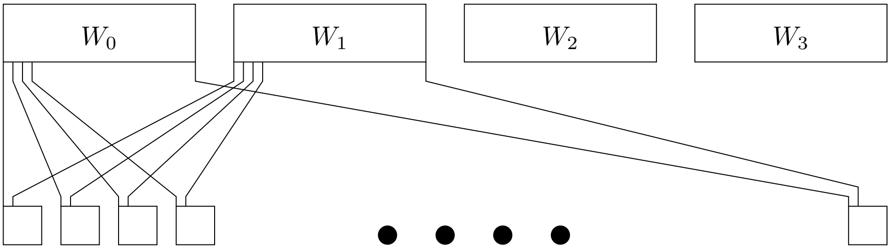


Fig. 87. PreSerpent( ) bitwise permutation

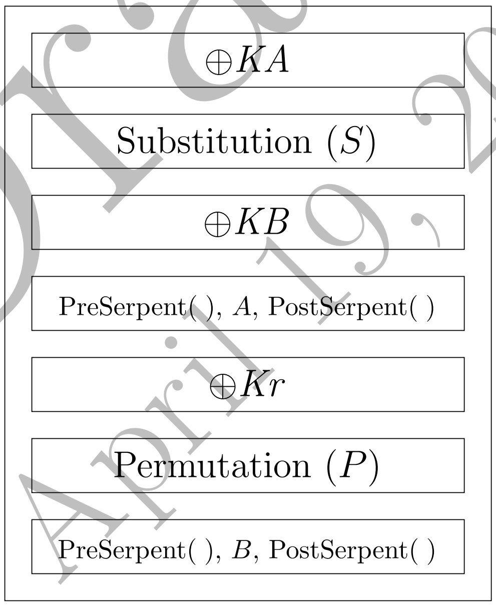

$\oplus KA$
Substitution (S)
$\oplus KB$
PreSerpent( ), A, PostSerpent( )
$\oplus Kr$
Permutation (P)
PreSerpent( ), B, PostSerpent( )

Fig. 88. Structure of a full round of Q

in length, each contains only 32 bits of information, since each is the concatenation of four 32-bit words, any two of which are rotations of each other. Second, the key-scheduling algorithm is reversible—as a consequence, given any subkey other than KA or KB, it is easy to determine all the remaining subkeys [434].

For $0 \leq r \leq 7$, round r has the structure in Figure 88.[1] Note that a full round actually contains three substitution stages (S, A, and B). The entire cipher is described by:

$$\oplus \mathit{KW1},\ \text{Round0}, \ldots, \text{Round7},\ \oplus \mathit{KA},\ \text{Substitution}(S),\ \oplus \mathit{KB},\ \oplus \mathit{KW2}\ .$$

It follows that the 8-round version of Q contains a total of 25 substitution stages. The use of KA and KB can be viewed as making the substitution with S key-dependent. However, Q also conforms to the standard SPN structure in which a subkey is XOR'd before each fixed substitution stage [343] (using the standard terminology, the version of Q we are considering consists of 25 rounds and has repeated subkeys).

3 Linear Probability

Given a bijective mapping $B : \{0,1\}^d \rightarrow \{0,1\}^d$, and *masks* $\mathbf{a}, \mathbf{b} \in \{0,1\}^d$, the associated *linear probability* (LP) value is defined as

$$LP(\mathbf{a}, \mathbf{b}) \overset{\text{def}}{=} \left(2 \cdot \text{Prob}\left\{\mathbf{a} \bullet \mathbf{X} = \mathbf{b} \bullet B(\mathbf{X})\right\} - 1\right)^2 ,$$

where $\mathbf{X}$ is a random variable uniformly distributed over $\{0,1\}^d$, and $\bullet$ denotes the inner product over GF(2). Note that $LP(\mathbf{a}, \mathbf{b}) \in [0,1]$; nonzero LP values indicate a correlation between the input and output of B, with higher values indicating a greater correlation.

If B is parameterized by a key, $\mathbf{k}$, we write $LP(\mathbf{a}, \mathbf{b}; \mathbf{k})$, and the *expected* LP (ELP) over the uniform distribution of keys is denoted

$$ELP(\mathbf{a}, \mathbf{b}).$$

3.1 LP Values for the Q S-boxes

In what follows, we will only be interested in LP values for the s-boxes of Q corresponding to masks containing a single 1. We give these values in Tables 75, 76, and 77. Entry $[i, j]$ is the LP value for input (output) mask with 1 in position i (j) and all other bits equal to 0. (Recall that we number bits from left to right starting at 0, with 0 indicating least significance.) For S, we denote this entry $LP_S[i, j]$ (entries for A and B are subscripted accordingly).

[1] This figure is taken from [434]; however, it disagrees slightly with the test code included in McBride's NESSIE submission, which specifies that permutation P should be applied *before* application of subkey Kr.

Table 75. LP values for s-box S

		0	1	2	3	4	5	6	7
0		$(\frac{6}{64})^2$	0	$(\frac{7}{64})^2$	$(\frac{6}{64})^2$	$(\frac{4}{64})^2$	$(\frac{2}{64})^2$	$(\frac{2}{64})^2$	$(\frac{6}{64})^2$
1		$(\frac{1}{64})^2$	$(\frac{4}{64})^2$	$(\frac{1}{64})^2$	$(\frac{3}{64})^2$	$(\frac{1}{64})^2$	$(\frac{4}{64})^2$	$(\frac{8}{64})^2$	$(\frac{1}{64})^2$
2		$(\frac{4}{64})^2$	$(\frac{1}{64})^2$	$(\frac{3}{64})^2$	$(\frac{3}{64})^2$	$(\frac{6}{64})^2$	$(\frac{8}{64})^2$	$(\frac{1}{64})^2$	$(\frac{1}{64})^2$
3		$(\frac{1}{64})^2$	$(\frac{1}{64})^2$	$(\frac{2}{64})^2$	0	$(\frac{6}{64})^2$	$(\frac{3}{64})^2$	$(\frac{1}{64})^2$	$(\frac{2}{64})^2$
4		$(\frac{6}{64})^2$	$(\frac{1}{64})^2$	$(\frac{3}{64})^2$	$(\frac{1}{64})^2$	$(\frac{4}{64})^2$	$(\frac{5}{64})^2$	0	$(\frac{4}{64})^2$
5		$(\frac{3}{64})^2$	$(\frac{5}{64})^2$	$(\frac{1}{64})^2$	$(\frac{6}{64})^2$	$(\frac{1}{64})^2$	0	$(\frac{4}{64})^2$	$(\frac{6}{64})^2$
6		$(\frac{2}{64})^2$	$(\frac{2}{64})^2$	$(\frac{6}{64})^2$	$(\frac{8}{64})^2$	$(\frac{3}{64})^2$	$(\frac{4}{64})^2$	$(\frac{6}{64})^2$	$(\frac{2}{64})^2$
7		$(\frac{6}{64})^2$	$(\frac{6}{64})^2$	$(\frac{8}{64})^2$	$(\frac{7}{64})^2$	$(\frac{4}{64})^2$	$(\frac{6}{64})^2$	$(\frac{2}{64})^2$	$(\frac{2}{64})^2$

Table 76. LP values for s-box A

		0	1	2	3
0		0	$\frac{1}{16}$	$\frac{1}{16}$	$\frac{1}{16}$
1		0	0	$\frac{1}{16}$	0
2		0	$\frac{1}{16}$	0	$\frac{1}{16}$
3		0	0	0	0

Table 77. LP values for s-box B

		0	1	2	3
0		$\frac{1}{16}$	0	0	0
1		$\frac{1}{16}$	$\frac{1}{16}$	0	$\frac{1}{16}$
2		0	$\frac{1}{16}$	$\frac{1}{16}$	0
3		0	0	$\frac{1}{16}$	$\frac{1}{16}$

4 Linear Cryptanalysis

Linear cryptanalysis (LC) is a known-plaintext attack due to Matsui [421]. We make use of the version known as Algorithm 2. We do not give the details of LC here (see [343] for a description of LC applied to SPNs). It suffices to say that the attacker attempts to find one or more of the outermost subkeys by choosing input/output masks $\mathbf{a}, \mathbf{b} \in \{0,1\}^N$ (recall that N is the block size) for a core subset of the substitution stages such that the corresponding value $LP(\mathbf{a}, \mathbf{b}; \mathbf{k})$ is relatively large (ideally, maximized).

In general, $LP(\mathbf{a}, \mathbf{b}; \mathbf{k})$ cannot be computed because $\mathbf{k}$ is unknown, so researchers have adopted the practice of using the expected value $ELP(\mathbf{a}, \mathbf{b})$ (over all independently chosen and uniformly random subkeys) [297, 342, 343, 613]. If

$$\mathcal{N}_L = \frac{c}{ELP(\mathbf{a}, \mathbf{b})}$$

is the number of known $\langle$plaintext, ciphertext$\rangle$ pairs used by the attacker (this is called the *data complexity*), the success rate of Algorithm 2 is given in Table 78. Note that this is the same as Table 3 in [421], except that the constant values differ by a factor of 4, since Matsui uses *bias* values, not LP values.[2]

Table 78. Success rates for Algorithm 2

c	8	16	32	64
Success rate	48.6%	78.5%	96.7%	99.9%

4.1 Linear Characteristics and Linear Hulls

Computing $ELP(\mathbf{a}, \mathbf{b})$ directly appears to be infeasible for most block ciphers, but an efficient first approximation can be found using *linear characteristics* (or simply *characteristics*). Let T denote the number of core substitution stages over which ELP values are required. A T-stage characteristic is a $(T+1)$-tuple of N-bit values, $\Omega = \langle \mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_T, \mathbf{a}_{T+1} \rangle$. We view $\mathbf{a}_t$ and $\mathbf{a}_{t+1}$ as input and output masks, respectively, for the t^{th} substitution stage ($1 \le t \le T$). The *expected linear characteristic probability* of Ω, denoted $ELCP(\Omega)$, is defined as

$$ELCP(\Omega) \overset{\text{def}}{=} \prod_{t=1}^{T} ELP(\mathbf{a}_t, \mathbf{a}_{t+1}) \,. \tag{14}$$

Note that $\mathbf{a}_t$ and $\mathbf{a}_{t+1}$ determine input/output masks for each s-box in round t. Those s-boxes having nonzero input and output masks are called *active s-boxes.*

[2] The corresponding table in [343] has an error, in that the constants have *not* been multiplied by 4 to reflect the use of LP values.

Moreover, the bits in any mask that are equal to 1 are called *active bits.* If a characteristic, Ω, results in any s-box having a zero input mask and a nonzero output mask, or vice versa, it is easy to show that the ELP for that substitution stage is 0, and therefore $ELCP(\Omega) = 0$ by (14). For simplicity, we exclude all such characteristics from further consideration.

The attacker typically finds the characteristic, $\Omega = \langle \mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_T, \mathbf{a}_{T+1} \rangle$, whose ELCP is maximal (called the *best* characteristic [423]), and then sets $\mathbf{a} = \mathbf{a}_1$, $\mathbf{b} = \mathbf{a}_{T+1}$, and uses the approximation

$$ELP(\mathbf{a}, \mathbf{b}) \approx ELCP(\Omega) . \tag{15}$$

However, more careful analysis requires the concept of *linear hulls,* due to Nyberg [487]. Given masks **a** and **b** for the T stages under consideration, the corresponding linear hull, denoted $ALH(\mathbf{a}, \mathbf{b})$,[3] is the set of all T-stage characteristics whose first mask is **a** and whose final mask is **b**. Nyberg then shows that

$$ELP(\mathbf{a}, \mathbf{b}) = \sum_{\Omega \in ALH(\mathbf{a}, \mathbf{b})} ELCP(\Omega).$$

It follows that (15) does not hold in general. The difference between the ELCP of a characteristic and the ELP value it is used to approximate is called the *linear hull effect.* If analysis is based on characteristics instead of linear hulls, the linear hull effect may result in an overestimation of the data complexity required for a given success rate—this is beneficial for an attacker, but potentially problematic for a cipher designer.

5 Computing Linear Hulls for Q

McBride performs preliminary linear cryptanalysis of Q by considering the formation of characteristics in which every N-bit mask contains a single active bit [434]. We call these *restricted characteristics.* McBride estimates that the ELCP of the best characteristic over 8 rounds is in the range of 2^{-118} (i.e., bias value $= 2^{-60}$). However, as we show below, it is straightforward to combine restricted characteristics into linear hulls[4] for which the ELP value is much higher than this, i.e., *the linear hull effect is significant for Q.*

In order to form linear hulls over T core stages, our algorithm uses a 3-dimensional data structure DS[ ] of size $128 \times (T+1) \times 128$, in which each entry is a record of two values: an integer Count, and a real value ELP.[5] After running the algorithm, DS$[i, t, j]$ contains information about the linear hull over substitution stages $1 \ldots t$ whose first (last) mask contains a single 1 in position

[3] Nyberg originally used the term *approximate linear hull*, hence the abbreviation ALH.

[4] Technically, we are building *sub*-linear hulls, since we are including only a subset of all the characteristics belonging to a particular linear hull.

[5] Our approach has strong similarities to the construction of differentials for Q by Biham et al. [73]; however, whereas Biham et al. use a linear algebraic approach, we opt for an algorithmic description.

i (j)—specifically, the Count field is the number of restricted characteristics in the linear hull, and the ELP field is the sum of the ELCP values of those restricted characteristics.

For the presentation of the algorithm, and for Theorem 0.1 below, we strip off the first and last S-substitution stages, so $T = 23$ (these results are easily generalized to any number of substitution stages). The algorithm is given in Figure 89 and Figure 90. (The pseudocode for subroutine ApplyB( ) is omitted, as it is symmetric to that for ApplyA( )—simply replace LP_A with LP_B. Also, the pseudocode for subroutine PostSerpent( ) is omitted, as it is simply the inverse of PreSerpent( ).) Note that we use the shorthand $x \mathrel{+}= y$ to mean $x \leftarrow x + y$. The values that are important for our attack on 8-round Q will be stored in the entries $\mathrm{DS}[i, 23, j]$.

Theorem 0.1. *If* DS[ ] *is filled using the algorithm in Figures 89 and 90, then for* $0 \le i, j \le 127$ *and* $0 \le t \le 23$, $\mathrm{DS}[i, t, j]$.Count *is the number of restricted characteristics over the first t of the 23 substitution stages whose first (last) mask contains a single 1 in position i (j), and for which ELCP* > 0*; and* $\mathrm{DS}[i, t, j]$.ELP *is the sum of the ELCPs of these characteristics.*

Proof. Let $0 \le i \le 127$ be fixed. The theorem is easily proven using induction on t. Trivially, the base case ($t = 0$) is made true for all j by the first For loop in the main program (Figure 89). We assume the statement holds for some $t \ge 0$ and demonstrate its truth for $(t+1)$. Note that the truth of the statement is not affected by the linear transformations (bitwise permutations) in Q; we simply perform the "bookkeeping" of permuting the elements of $\mathrm{DS}[i, t, \cdot]$ accordingly.[6] Therefore, we need only consider the effect of the $(t+1)^{\mathrm{st}}$ substitution stage on $\mathrm{DS}[i, t, \cdot]$, and without loss of generality we can limit our consideration to substitution using S. Further, without loss of generality we will consider only the effect of the first (leftmost) copy of S, denoted S_0. The inputs/outputs for S_0 are bits $0 \ldots 7$ of the respective blocks.

Let $\tilde{\jmath} \in \{0, \ldots, 7\}$. Consider all restricted characteristics over the first $(t+1)$ substitution stages whose first (last) mask has bit i ($\tilde{\jmath}$) active. Clearly all these characteristics make S_0 active. Therefore, in any of these characteristics, the mask *preceding* the $(t+1)^{\mathrm{st}}$ substitution stage must have the position of its (only) active bit in the range $0 \ldots 7$. It follows that

$$\mathrm{DS}[i, t+1, \tilde{\jmath}].\mathrm{Count} = \sum_{j=0}^{7} \mathrm{DS}[i, t, j].\mathrm{Count}, \tag{16}$$

with one proviso: if $\mathrm{LP}_S[j, \tilde{\jmath}] = 0$, then extending any t-stage restricted characteristic enumerated by $\mathrm{DS}[i, t, j]$.Count (for $0 \le j \le 7$) to $(t+1)$ stages will

[6] This works because given a mask *before* a linear transformation in Q, the corresponding mask *after* the linear transformation is obtained by processing the mask through the linear transformation. This applies to Q because all linear transformations are bitwise permutations. However, this does not hold in general—for an arbitrary linear transformation represented as a binary matrix, *output* masks are transformed to *input* masks via multiplication by the *transpose* of the linear transformation [177].

produce a value ELCP $= 0$ (by (14)). Therefore, we omit all such j by modifying (16) as follows:

$$\mathrm{DS}[i, t+1, \tilde{\jmath}].\mathrm{Count} \;=\; \sum_{\substack{0 \le j \le 7 \\ \mathrm{LP}_S[j,\tilde{\jmath}]>0}} \mathrm{DS}[i, t, j].\mathrm{Count} \tag{17}$$

(this is done via the If statement in subroutine ApplyS( )). It is easily seen that $\mathrm{DS}[i, t+1, \tilde{\jmath}]$.ELP is correctly assigned the sum of the ELCP values of all the characteristics enumerated by (17). □

5.1 Computational Results

We ran our algorithm for varying numbers of rounds by modifying the main program in Figure 89 appropriately. The best ELP values found are given in Table 79. For comparison, we include Table 80, which contains the corresponding best expected differential probability (EDP) values from [73]. The ELP values represent a minimum improvement in the exponent of approximately 17 relative to [73]; the improvement in the exponent is 20.4 for the case that is of primary interest to us: 7 full rounds with $A + B$ prepended, hereafter denoted $A + B + 7$.

Table 79. Best ELP values

Number of rounds	Full rounds only	With additional S appended	With additional $A + B$ prepended
6	$2^{-72.3}$	$2^{-77.2}$	$2^{-78.8}$
7	$2^{-83.7}$	$2^{-88.6}$	$2^{-90.1}$
8	$2^{-95.1}$	$2^{-100.0}$	$2^{-101.5}$
9	$2^{-106.4}$	$2^{-111.3}$	-

Table 80. Corresponding best EDP values from Biham et al. [73]

Number of rounds	Full rounds only	With additional S appended	With additional $A + B$ prepended
6	$2^{-92.9}$	$2^{-105.35}$	$2^{-95.5}$
7	$2^{-107.9}$	$2^{-120.35}$	$2^{-110.5}$
8	$2^{-122.9}$	$2^{-135.35}$	$2^{-125.5}$
9	$2^{-137.9}$	$2^{-150.35}$	-

```
Initialize all Count and ELP fields in DS[ ] to 0
For $i = 0$ to 127
    DS[$i, 0, i$].Count $\leftarrow 1$
    DS[$i, 0, i$].ELP $\leftarrow 1$

For $i = 0$ to 127
    $t \leftarrow 0$
    ApplyA $(i, t)$;   $t$ += 1
    Permute $(i, t)$
    ApplyB $(i, t)$;   $t$ += 1

    For Round = 1 to 7
        ApplyS $(i, t)$;   $t$ += 1
        ApplyA $(i, t)$;   $t$ += 1
        Permute $(i, t)$
        ApplyB $(i, t)$;   $t$ += 1
```

```
Subroutine ApplyS $(i, t)$
    $\mathcal{J} \leftarrow \{j :$ DS[$i, t, j$].Count $> 0\}$
    For $j \in \mathcal{J}$
        BoxIndex $\leftarrow j$ div 8
        InBit $\leftarrow j$ mod 8
        For OutBit = 0 to 7
            If   LP$_S$[InBit, OutBit] $> 0$
                $\tilde{\jmath} \leftarrow 8 \times$ BoxIndex + OutBit
                DS[$i, t+1, \tilde{\jmath}$].Count += DS[$i, t, j$].Count
                DS[$i, t+1, \tilde{\jmath}$].ELP += DS[$i, t, j$].ELP $\times$ LP$_S$[InBit, OutBit]
```

Fig. 89. Pseudocode for computation of linear hulls over 23 core stages

```
Subroutine ApplyA $(i, t)$

    PreSerpent $(i, t)$
    $\mathcal{J} \leftarrow \{j : \mathrm{DS}[i, t, j].\mathrm{Count} > 0\}$
    For $j \in \mathcal{J}$
        BoxIndex $\leftarrow j$ div 4
        InBit $\leftarrow j$ mod 4
        For OutBit = 0 to 3
            If $\mathrm{LP}_A[\mathrm{InBit}, \mathrm{OutBit}] > 0$
                $\tilde{\jmath} \leftarrow 4 \times \mathrm{BoxIndex} + \mathrm{OutBit}$
                $\mathrm{DS}[i, t+1, \tilde{\jmath}].\mathrm{Count} \mathrel{+}= \mathrm{DS}[i, t, j].\mathrm{Count}$
                $\mathrm{DS}[i, t+1, \tilde{\jmath}].\mathrm{ELP} \mathrel{+}= \mathrm{DS}[i, t, j].\mathrm{ELP} \times \mathrm{LP}_A[\mathrm{InBit}, \mathrm{OutBit}]$
    PostSerpent $(i, t+1)$
```

```
Subroutine PreSerpent $(i, t)$

    For $j = 0$ to 127
        $\mathrm{Temp}[j] \leftarrow \mathrm{DS}[i, t, j]$
    For $j = 0$ to 127
        $\tilde{\jmath} \leftarrow 4 \times (j \bmod 32) + (j \operatorname{div} 32)$
        $\mathrm{DS}[i, t, \tilde{\jmath}] \leftarrow \mathrm{Temp}[j]$
```

```
Subroutine Permute $(i, t)$

    Partition $\mathrm{DS}[i, t, \cdot]$ into 4 "words" of size 32 $(W_0, W_1, W_2, W_3)$:
        $W_s \leftarrow \langle \mathrm{DS}[i, t, 32s], \ldots, \mathrm{DS}[i, t, 32s+31] \rangle$,   for $s = 0 \ldots 3$
    Leave $W_0$ unchanged
    Right rotate $W_1$ by 8, $W_2$ by 16, $W_3$ by 24
```

Fig. 90. Pseudocode for other subroutines

5.2 Recovering the Full Key

A linear hull over $A+B+7$ with input/output masks that each contain a single 1 can be used to derive two bytes of keying information: the byte XOR'd before the active copy of S in the first substitution stage, and the byte XOR'd after the active copy of S in the last substitution stage. Therefore we need 2^{16} counters to carry out linear cryptanalysis using such a linear hull. Note that the bytes we obtain are in fact pieces of the 128-bit vectors $(KW1 \oplus KA)$ and $(KB \oplus KW2)$, respectively, i.e., they do not give us subkey bytes directly.

By carrying out linear cryptanalysis with 16 different linear hulls, each of which activates a different copy of S in the last substitution stage, we can systematically recover the bytes of $(KB \oplus KW2)$. Using our algorithm, we found the best linear hull for attacking each of these bytes; these are given in Table 81. The smallest of the 16 ELP values is approximately 2^{-91}, so opting for a 99.9% success rate for each linear hull requires a data complexity of $\frac{64}{2^{-91}} = 2^{97}$ (Table 78). Assuming that the success rates of the 16 individual attacks are independent, the overall success rate is $(0.999)^{16} \approx 98.4\%$.

Once we have determined $(KB \oplus KW2)$, we can exhaustively search all 2^{32} bits of information in KB (see Section 2.2). For each guess of KB we obtain a guess of $KW2$, and this yields the remaining subkeys by running the key-scheduling algorithm backward. A trial encryption can be used to discard each wrong guess of KB, so at most 2^{32} trial encryptions are required.

6 Conclusion

We have considered the vulnerability of Q to linear cryptanalysis based on linear hulls. We present a straightforward algorithm that combines all characteristics consisting of masks containing a single 1 (termed restricted characteristics) into the corresponding linear hulls, and we compute the expected linear probability (ELP) of each such linear hull (since we limit our consideration to restricted characteristics, the value we obtain is actually a lower bound on the ELP value of the full linear hull). The ELP of the best such linear hull over 7.5 rounds (8 full rounds minus the first S-substitution) is $2^{-90.1}$, a significant improvement over the best known expected differential probability (EDP) value for the same rounds, namely $2^{-110.5}$ [73]. We can use the linear hulls found by our algorithm to recover a 128-bit key with success rate 98.4%, using 2^{97} known $\langle \text{plaintext}, \text{ciphertext} \rangle$ pairs and 2^{32} trial encryptions.

There are a number of reasons for the success of our approach. First, each of the three s-boxes in Q has multiple nonzero LP values corresponding to input/output masks containing a single 1. In contrast, one of the design criteria for A and B was that no single-bit input *difference* can produce a single-bit output difference with nonzero probability [434]. This is the main reason that our ELP values are superior to the EDP values in [73]. Second, the linear transformations in Q have (very) low diffusion, allowing a mask containing a single 1 to be transformed into a mask also containing a single 1. Third, it is easy to combine a large

Table 81. Best linear hulls for attacking bytes of $KW2$

Byte of $KW2$	Active bits (input,output)	ELP	Number of characteristics in linear hull
0	(31, 3)	$2^{-91.1}$	94,726,326
1	(7, 11)	$2^{-91.1}$	94,726,326
2	(15, 19)	$2^{-91.1}$	94,726,326
3	(23, 27)	$2^{-91.1}$	94,726,326
4	(31, 35)	$2^{-90.1}$	191,795,706
5	(7, 43)	$2^{-90.1}$	191,795,706
6	(15, 51)	$2^{-90.1}$	191,795,706
7	(23, 59)	$2^{-90.1}$	191,795,706
8	(23, 67)	$2^{-90.2}$	188,281,125
9	(31, 75)	$2^{-90.2}$	188,281,125
10	(7, 83)	$2^{-90.2}$	188,281,125
11	(15, 91)	$2^{-90.2}$	188,281,125
12	(7, 99)	$2^{-90.2}$	183,092,934
13	(15, 107)	$2^{-90.2}$	183,092,934
14	(7, 115)	$2^{-90.2}$	183,092,934
15	(15, 123)	$2^{-90.2}$	183,092,934

number of restricted characteristics into a single linear hull, enabling the attack presented above. Finally, the cryptanalyst's job is made easier by the fact that finding a 128-bit subkey such as $KW2$ allows all the subkeys to be recovered.

Improved Analysis of the BMGL Keystream Generator

by Johan Håstad and Mats Naslund *

Abstract. In this paper we give an improved security analysis of the NESSIE submission BMGL. The new analysis improves also asymptotically some of the theoretical results on which the BMGL keystream generator is based. We also give an alternative, bootstrapped version of the generator which is implementation-wise very close to the original generator and offers even stronger provable security properties.

1 Introduction

The BMGL keystream generator, [285], was submitted in response to the NESSIE call for primitives, [633]. The construction is based on theoretical results due to Blum and Micali [101], and Goldreich and Levin [269] (hence the acronym). BMGL uses the Rijndael/AES block cipher [181] as the cryptographic core. In the sequel, we fix a (known) plaintext block and identify Rijndael/AES with the natural mapping of keys onto ciphertexts. BMGL can then be shown to have certain provable security properties under the (weakest possible) assumption: that the above function does not significantly loose its *one-way function* properties when iterated.

The main contribution in [285] is an "exact" analysis for fixed security parameter (key size) of the results in [101, 269]. By also introducing completely new elements to the analysis when many bits are output per application of the underlying core, an order of magnitude of improvement also in the asymptotical analysis is obtained, and these things together enables the authors to obtain relatively strong, provable properties already for keysizes around 100–200 bits. It also gives the (thus far) fastest generator with such provable properties: the equivalent of about two Rijndael applications are needed per 40 bits of output keystream.

The proofs in [285] reduces the (assumed) failure of BMGL to pass a "statistical test", to an inversion algorithm for the iterated Rijndael cipher, using the

* `johanh@nada.kth.se`
NADA
Royal Inst. of Technology
SE-10044 Stockholm, Sweden
(work partially supported by the Göran Gustafsson foundation and NSF grant CCR-9987077).

`mats.naslund@era.ericsson.se`
Communications Security Lab
Ericsson Research
SE-16480 Stockholm, Sweden

statistical test as a black box. The reduction relies on the existence of a certain location in the output keystream where there is concentration of the test's advantage. To get a uniform reduction, the main part of the workload in that reduction is to actually determine this location. In this paper we provide a new method to find this location, or "index". In fact, we can provide evidence that the new method is optimal up to a small constant.

A second ingredient in the original proof is an averaging argument that is used to show that the inversion succeeds on average, for a random input, rather than on a "small" subset of all inputs (which would normally be the result by a careless application of Markov's inequality). Here, we improve this averaging argument even further.

Finally, as mentioned, the original BMGL generator iterates the Rijndael cipher. The provable security goes down with the number of iterations needed, i.e. with the output keystream length. Using a result by Goldreich, Goldwasser, and Micali, [268], on the construction of *pseudo-random functions* we give an alternative, "bootstrapped" version of BMGL, which we call GGM (or "BMGL2"). This generator can use BMGL as a black box, and has the benefit that it never iterates Rijndael more than a constant number of times, giving an even stronger security preservation in the reduction. (For the suggested parameters, 16 iterations are enough for most practical purposes.)

2 Preliminaries

The length of binary string x is denoted $|x|$, and by $\{0,1\}^n$ we denote the set of x such that $|x| = n$. We write $\mathcal{U}_n$ for the uniform distribution on $\{0,1\}^n$. Except otherwise noted, log refers to logarithm in base 2.

Let $G : \{0,1\}^n \to \{0,1\}^{L(n)}$ and let A be an algorithm with binary output. We say that A is a $(L(n), T(n), \delta(n))$-*distinguisher* for G, if A runs in time $T(n)$ and $|\Pr_{x \in \mathcal{U}_n}[A(G(x)) = 1] - \Pr_{y \in \mathcal{U}_{L(n)}}[A(y) = 1]| \geq \delta(n)$. (We call $\delta(n)$ the *advantage* of A.) If no such A exists, G is called $(L(n), T(n), \delta(n))$-*secure.*

Our model of computation is slightly generous but realistic. We assume that simple operations like arithmetical operations and exclusive-ors on small[1] size integers can be done in unit time.

3 Improved Analysis of BMGL

We shall assume the reader is familiar with [285]. For self-containment however, we again give the definition of BMGL.

Definition 4. *Let n, and m, L, λ be integers such that $L = \lambda m$ and let $f : \{0,1\}^n \to \{0,1\}^n$. Let $\mathfrak{M}_m$ be the set of all $m \times n$ boolean matrices and for $x \in \{0,1\}^n$, $R \in \mathfrak{M}_m$, let $B_R^m(x)$ denote the binary matrix-vector product $R \cdot x$.*

[1] We need words of size n where n is size of the input on which we apply our one-way function, e.g. $n = 128$ or 256 for a typical block cipher.

The generator $BMGL^f_{n,m,L}(x,R)$ stretches $n+nm$ bits to L bits as follows. The input is interpreted as $x_0 = x$ and $R \in \mathfrak{M}_m$. Let $x_i = f(x_{i-1})$, $i = 1,2,\ldots,\lambda$ and let the output be $\{B^m_R(x_i)\}^{\lambda}_{i=1}$.

In BMGL the choice of f is

$$f(x) \triangleq \text{Rijndael}(x, P),$$

the Rijndael encryption of a fixed, known plaintext P (e.g. $P = 00\ldots0$) using key x, and where Rijndael is configured to have the block size equal to the key size, n.

We relate the difficulty of inverting an iterated function f to that of distinguishing outputs of $BMGL^f_{n,m,L}$ from random bits. Our measure of success is

Definition 5. *For a function $f : \{0,1\}^n \to \{0,1\}^n$, let $f^{(i)}(x)$ denote f iterated i times, $f^{(i)}(x) \triangleq f(f^{(i-1)}(x))$, $f^{(0)}(x) \triangleq x$.*

Let A be a probabilistic algorithm which takes an input from $\{0,1\}^n$ and has output in the same range. We then say that A is a (T,δ,i)-inverter for f if when given $y = f^{(i)}(x)$ for an x chosen uniformly at random, in time T with probability δ it produces z such that $f(z) = y$.

It is in Theorem 3 of [285] shown that a random function can be expected to be (T,δ,i)-invertible for $T/\delta \sim 2^n/i$. This leads to the following definition:

Definition 6. *A σ-*secure one-way function *is an efficiently computable function f that maps $\{0,1\}^n \to \{0,1\}^n$, such that the average time over success ratio for inverting the ith iterate is at most $\sigma 2^n/i$. That is, f cannot be (T,δ,i)-inverted for any $T/\delta < \sigma 2^n/i$.*

A block cipher, $f(p,k)$, $|p| = |k| = n$, is called σ-secure if the function $f_p(k)$, for fixed, known plaintext p, is a σ-secure one-way function of the key k.

Hence, for our "practical" choice, $f =$ Rijndael, we expect it to be about 1-secure in the above terminology.

The objective is to show that if $BMGL^f_{n,m,L}$ is not (L,T,δ)-secure for "practical" values of L,T,δ, then there is also a practical attack on the underlying one-way function f. In particular, we can show the following theorem (an improvement to Theorem 4 in [285]).

Theorem 2. *Suppose that $G = BMGL^f_{n,m,L}$ is based on an n-bit function f, computable by E operations, and that G produces L bits in time S. Suppose that this generator can be (L,T,δ)-distinguished. Then, setting $\delta' = \frac{\delta m}{L}$, there exists integers $i \le L/m \triangleq \lambda$, $0 \le j \le 2\log\delta'^{-1}$, such that for $k = \max(m, 1+\log((2n+1)\delta'^{-2}) - j)$, f can be $(T', d_j/2, i)$-inverted, where d_j is given by Eq. (22) and Eq. (23), page 774, and T' equals*

$$(1+o(1))2^{m+k}(2m+k+1+T+S+E)(n+1).$$

Values of i and j such that f can be $((8+o(1))T', d_j/16, i)$-inverted can, with probability at least 1/4, be found in time $O(\delta'^{-2}(T+S))$. The $O(\cdot)$ hides an absolute constant, numerically estimated by Eq. (20), p. 772.

The improvement is in the running time (preprocessing) to find i and a better total average time/success ratio of the inversion. We shall in Section 5 see exactly how this translates into the concrete security of BMGL when concrete examples are studied (and values for d_j are calculated). The time-success ratio for most ranges of δ and T is worst when the value of j is small. For $j \in O(1)$ and $m, k, E \leq S \leq O(T)$ the ratio is $O(n^2 L^2 \delta^{-2} 2^m T)$. The preprocessing time (to find i, j) is small compared to the running time except in the cases when j is large. In those cases the time to find j is still smaller than the running time of the inverter while the running time to find i might be larger for some choices of the parameters.

Before proceeding, for completeness we outline the key steps of the proof to the above theorem (i.e. to Theorem 4 of [285]) and indicate where we are able to make improvements.

– We show that an assumed distinguisher translates into an algorithm, $P^{(i)}$, that given $f^{(i)}(x)$ distinguishes $B_R^m(f^{(i-1)}(x))$ from randomness with some advantage. We also show how to find i. (This is Lemma 5 in the original paper, and is here improved in Lemma 3).
– We show that such a distinguisher can be used to produce an algorithm that given $y = f^{(i)}(x)$ *and* the advantage, ϵ_x, of the distinguisher for this particular x, produces a "small" set of candidates for a z with $f(z) = y$. (This is Theorem 6 in the original paper.)
– The proof of the original Theorem 6 relies on a result enabling us to generate a number of random, pairwise independent matrices, $\{R_t\} \subset \mathfrak{M}_m$ for which we already know each $B_{R_t}^m(x)$, or rather, a small set of candidates thereof. (This is Lemma 7 in the original paper, and a slight improvement appears below in Lemma 4.)
– The proof of Theorem 4 in the original paper follows from the above and a careful strategy to "guess" the correct ϵ_x to use in the application of the original Theorem 6. We here give an improved guessing-strategy that in a similar way establishes Theorem 2 above.

The other needed results (Lemma 8 and 9 in [285]) are needed also here, but are essentially unchanged. We would like to point out a small error here. The (fixed) S_i in Lemma 9 should be replaced by a set of matrices $\{S_i^j\}_{j=1}^{2^k}$ defined by $S_i^j = s_i^j \otimes v_i$, where $\{s_i^j\}_{j=1}^{2^k}$ are (pairwise) independent, random m-bit strings and where S_i^j is used in the jth sample. This ensures the pairwise independence of the samples, which otherwise only holds when $\langle x, v_i \rangle_2 = 0$. Then, S in Lemma 8 is replaced by the *set* $S_i = \{S_i^j\}$, given i.

In the original paper, an improved inversion algorithm based on error-correcting codes was also given. This is applicable here too, offering corresponding further improvements. However, we omit the details. We now proceed to give the new, improved analysis details.

As mentioned, the first, and main step in the improvement is the lemma below which provides us with a more efficient method for finding the i-value mentioned in Theorem 2.

Lemma 3. *Let $L = \lambda m$. Suppose that $BMGL^f_{n,m,L}$ runs in time $S(L)$. If this generator is not $(L, T(L), \delta)$-secure, then there is an algorithm $P^{(i)}$, $1 \leq i \leq L/m$ that, using $T(L)+S(L)$ operations, given $f^{(i)}(x), R$, for random $x \in \mathcal{U}_n$, $R \in \mathfrak{M}_m$, distinguishes $B^m_R(f^{(i-1)}(x))$ from $\mathcal{U}_m$ with advantage $\delta' = \frac{\delta m}{L}$.*

$P^{(i)}$ depends on an integer i, and using $c_1\delta'^{-2}(T(L)+S(L))$ operations, where c_1 is the constant given by Eq. (20), page 772, a value of i achieving advantage $\delta_i \geq \delta'/2$ can be found with probability at least $1/2$.

All (some rather lengthy) proofs appears in the Appendix.

We conjecture that the time needed to find i is optimal up to the value of the constant c_1. Even if a good value i was given to us at no cost, the straightforward way by sampling to verify that it actually is as good as claimed would take time $\Omega(\delta'^{-2}(T(L) + S(L)))$. It is not difficult to see that the below proof can be modified to find an i with δ_i arbitrarily close to δ', rather than just $\delta'/2$. The cost is simply an increase in the constant c_1.

Finally, another slight improvement is in the following Lemma which corresponds to Lemma 7 in [285]. The improvement is in a more careful analysis of the matrix-generation which reduces the running time by a factor k^3.

Lemma 4. *Fix any $x \in \{0,1\}^n$. For $m < k$, from $m + k$ randomly chosen $a_0, \ldots, a_{m-1}$ and $b_0, \ldots, b_{k-1} \in \{0,1\}^n$, it is possible in time $2m2^k + k^2 + m + 4k$ to generate a set of 2^k uniformly distributed, pairwise independent matrices $R^1, \ldots, R^s \in \mathfrak{M}_m$. Furthermore, there is a collection of $m \times (m + k)$ matrices $\{M_j\}_{j=1}^{2^k}$ and a vector $z \in \{0,1\}^{m+k}$ such $B^m_{R^j}(x) = M_j z$ for all j.*

The construction generalises that of Rackoff for the case $m = 1$, see [266]. If $k < m$, we use $k' = m$ above and then simply only take the first 2^k matrices.

Based on the above the (new, improved) proof of Theorem 2 is given in the Appendix.

4 Applying the GGM construction

As shown, the BMGL generator can produce any number of output bits. We here investigate an alternative way, inspired by a construction of *pseudo random functions* due to Goldreich, Goldwasser and Micali, [268]. It has the advantage that we iterate f fewer times and hence the assumption needed for security is weaker.

The construction can be based on any PRG, $G : \{0,1\}^n \to \{0,1\}^{2n}$, though we for concreteness think of $G = G(x, R) = BMGL^f_{n,m,2n}(x, R)$ for some f. For simplicity of notation, we shall exclude R from it, keeping in mind that probabilities should be taken also over the choice of R. First, let us assume that we know in advance how may output bits that are desired. We apply [268] to obtain $2^d n$ output bits (where d is given) from $n(m + 1)$ bits.

Definition 7. *Fix $n, d \in \mathbb{N}$. Let $G(x)$ be a generator, stretching n bits to $2n$ bits, and let $G_0(x)$ $(G_1(x))$ be the first (last) n bits of $G(x)$. For $x \in \{0,1\}^n$,*

$s \in \{0,1\}^d$ put $g_x(s) \triangleq G_{s_d}(G_{s_{d-1}}(\cdots G_{s_2}(G_{s_1}(x))\cdots))$, and define $GGM^G_{d,n} : \{0,1\}^n \to \{0,1\}^{2^d n}$ by

$$GGM^G_{d,n}(x) \triangleq g_x(00\dots0), g_x(00\dots1), \cdots, g_x(11\dots1)$$

(the concatenation of g_x applied to all d-bit inputs).

The construction can be pictured as a full binary tree $T = (V, E)$ of depth n. Associate $v \in V$ with its breadth-first order number; the root is 1 and the children of v are $2v, 2v + 1$. Given x, the root is first labelled by $\mathcal{L}(1) = x$. For a non-leaf v labelled $\mathcal{L}(v) = y \in \{0,1\}^n$, label its children by $\mathcal{L}(2v) = G_0(y)$, $\mathcal{L}(2v + 1) = G_1(y)$, respectively. The output of $GGM^G_{d,n}$ is simply the concatenation of all the "leaves" of the tree.

Notice an advantage of the above method when $G = BMGL^f_{n,m,2n}$. To produce $L = 2^d n$ bits, each application of G iterates f $2n/m$ times instead of $2^d n/m$, which, in light of Theorem 3 of [285], retains more of the one-wayness of f.

Lemma 5. *Suppose that D_1 is a $(2^d n, T, \delta)$-distinguisher for $GGM^G_{d,n}(x)$ where G can be computed in time S. Then, there is an integer $i \leq 2^d$ and algorithm D^i that is an $(2n, T + 2^d S, 2^{-d}\delta)$-distinguisher for G.*

D^i depends on i, and a value of i achieving advantage $\delta_i \geq 2^{-(d+1)}\delta$ can be found with probability at least $1/2$ in time $c_1 2^{2d}\delta^{-2}(T + 2^d S)$ where c_1 is the constant given by Eq. (20), page 772.

Proof (Proof sketch.). Consider a binary tree, T, describing a computation of $GGM^G_{d,n}$ as above. The tree has depth d, $2^d - 1$ internal vertices and 2^d leaves. We construct hybrid distributions $H^0, \dots, H^{2^d-1}$ on the vertex-labels of such trees. Again, associate each $v \in V$ by its breadth-first order number. Then, H^i is defined by a simulation algorithm, $GGM^i(x)$, which on input x, assigns labels as follows. Assign the root, $v = 1$, the label x. For $v \in V$, $v = 1, 2, \dots, i$, label v's children by letting $\mathcal{L}(2v), \mathcal{L}(2v + 1)$ be independent, random n-bit strings. Then, for $v = i + 1, \dots, 2^d - 1$: $\mathcal{L}(2v) = G_0(\mathcal{L}(v))$, $\mathcal{L}(2v + 1) = G_1(\mathcal{L}(v))$. Finally return labels of the leaves in T.

Observe that H^{2^d-1} gives the uniform distribution over the labels (in particular the leaves) and H^0 labels the vertices exactly as $GGM^G_{n,d}$ does on a random seed x. Since D_1 distinguishes $GGM^G_{d,n}(x)$ from random $2^d n$-bit strings with advantage δ, for some $i \leq 2^d$, it must be the case that D_1 distinguishes H^i, H^{i+1} with advantage at least $2^{-d}\delta$.

To find i, we use the same analysis as in the proof of Lemma 7 (see appendix) with $\lambda = 2^d$, $m = 2n$, and $F_i(x, z)$ defined by the labelling according to H^i.

We now construct D^i: when D^i gets input $\gamma \in \{0,1\}^{2n}$, it selects random x and feeds D_1 a value y, computed as $GGM^{i+1}(x)$ with the following exception: $i + 1$ is not assigned any label[2], and the children of $i + 1$ are assigned the left/right n-bit half of γ respectively. It is not too hard to see that if γ is random, we give D_1

[2] As the labels of non-leaves are never exposed, one can conceptually think of the process as labelling $i + 1$ afterwards.

a value according to exactly the same distribution as H^{i+1}, whereas if $\gamma = G(x')$, D_1 is given a value from the same distribution as $GGM^i(x)$, i.e. H^i. Thus, by returning D_1's answer to y, D^i's advantage equals that of D_1.

4.1 Unknown Output Length

If the length of the "stream" is unknown beforehand, we let the basic generator G expand n bits to $3n$ bits. Apply the tree-construction as above, labelling left/right children by the first, respectively second n-bit substring of G's output. The remaining n bits are used to produce an output at each vertex as we traverse the tree breadth-first. The analysis is analogous. To save memory, the traversal can be implemented in iterative depth-first fashion.

5 Concrete Examples

What does all this say? Suppose that we base the construction on Rijndael and that we want to generate $L = 2^{30}$ bits, applying our construction with $m = 40$ (40 bits per iteration). One choice of parameters gives the following corollary.

Corollary 6. *Consider $G = BMGL^{Rijndael}_{256,40,2^{30}}$ (using key/block length 256) and where Rijndael is computable by E operations, and assume that G runs in time S. If G can be $(2^{30}, T, 2^{-32})$-distinguished, then there is $i < 2^{25}$, and $0 \leq j \leq 114$ such that setting $k = \max(40, 123 - j)$, Rijndael can be (T', d_j, i)-inverted for $T' = 2^{49+k}(81 + k + T + S + E)$.*

Similarly, setting $G' = BMGL^{Rijndael}_{256,40,512}$ and then using $GGM^{G'}_{22,256}$ (to generate the same length outputs), the result holds for some $i < 13$.

This is simply substituting the parameters and noting that the $o(1)$ in Theorem 2 comes from disregarding the time to construct the matrices described in Lemma 4 and for the current choice of parameters using $(1 + o(1))(n + 1) \leq 2^9$ is an overestimate.

Assuming that we have a simple statistical test such as Diehard tests, [416], or those by Knuth, [372], it is reasonable to assume[3] that $81 + k + T + E \leq S$. From the first part of the corollary, then, the essential part of computing the generator comes from the roughly 2^{25} computations of Rijndael and we end up with a time for the inverter equivalent to at most 2^{75+k} Rijndael computations. The maximum of $2^k(d_j/2)^{-1}$ is obtained for $j = 5$ in which case it equals $2^{124} \cdot 7.5 \leq 2^{127}$. We conclude that in this case we get a time-success ratio that is equivalent to at most 2^{202} computations of Rijndael and since $i \leq 2^{25}$, Rijndael would not be 2^{-29}-secure. (This is a factor 16 better than the security bound obtained in [285].) As the situation is now, this is probably not a "catastrophe" for AES, but had such a flaw been known at the time AES was selected, we are convinced that Rijndael would not have been chosen.

[3] Common "practical" tests are almost always much faster than the generator tested.

Alternatively, bootstrapping BMGL by the GGM method, we conclude from the second part of the corollary that such a test would mean that Rijndael cannot be even 2^{-50}-secure. Thus, though somewhat more cumbersome to implement, the GGM method is more security preserving.

The level of protection against other distinguishers can easily be deduced from the formulas: allowing the distinguisher (for GGM) to use time equivalent to 2^{50} Rijndael applications, we deduce that Rijndael would not be 2^{-16}-secure.

If we want to find the values of i and j efficiently the time-success ratio increases by a factor about 2^6. Note that for the case with small j the time needed to find i and j is much smaller than the running time of the inverter.

6 Summary and Conclusions

We have given a stronger analysis of the concrete BMGL generator and also improved some theoretical results on which the construction is based. An alternative construction, BMGL2/GGM has also been given and we have seen that although somewhat more difficult to implement, even stronger security can be obtained.

Acknowledgement. We thank Gustav Hast, Bernd Meyer, and anonymous reviewers for helpful comments.

A Proofs

Proof sketch of Lemma 3. The proof uses an optimised version of the so called *universality of the next-bit-test*, by Yao [626], see also [101].

Using standard "hybrid argument", we define efficiently sampleable distributions H^i, $i = 0, 1, \ldots, \lambda$, on $\{0,1\}^L$ such that H^0 equals the distribution of outputs of the generator, and so that H^λ is the uniform distribution on $\{0,1\}^L$. We simply iteratively apply f and R, $\lambda - i$ times, to random strings to obtain H^i. By assumption, D distinguishes H^0 and H^λ with advantage δ, and let δ_i be the advantage D has in distinguishing H^{i-1} and H^i for $i = 1, \ldots \lambda$. From the triangle inequality follows the existence of an i, with $\delta_i \geq \delta'$. With this i known, one easily constructs the algorithm $P^{(i)}$.

We need to find such an i efficiently. This is treated next in Lemma 7, by setting $F_i(x,z) = B_R(f(x)), B_R(f^{(2)}(x)), \ldots, B_R(f^{(\lambda-i)}(x))$.

Lemma 7. *Let $\{F_i(x,z)\}_{i=0}^{\lambda}$ be a set of functions, $F_i : \{0,1\}^n \times (\{0,1\}^m)^i \to (\{0,1\}^m)^{\lambda-i}$ and where each F_j is computable in time $\leq S$. Let H^i be the distribution on $(\{0,1\}^m)^\lambda$ induced by $(z, F_i(x,z))$ when $x \in \mathcal{U}_n$, $z \in (\mathcal{U}_m)^i$.*

Suppose that H^0, H^λ $(= (\mathcal{U}_m)^\lambda)$ are distinguishable with advantage δ, by an algorithm D running in time T. Then, a value of $i < \lambda$ for which H^i, H^{i+1} can be distinguished with advantage $\delta' = \frac{\delta}{2\lambda}$, can with probability at least $\frac{1}{4}$, be found in time $c_1 \delta'^{-2}(T + S)$ where c_1 is an absolute constant.

Proof. Let δ_i be the advantage on H^i, H^{i+1}. The problem is that even though $\mathrm{E}_i[\delta_i] = \delta'$, there is a large number of possibilities for the individual δ_i. Basically, these possibilities all lie between the two extreme cases: (1) There are a few large δ_i, while most are close to 0. (2) All δ_i are about the same, but none is very large. Suppose we try random i's. In the first case, we may need to try many i, but it can be done with a rather low sampling accuracy. In the second case, we expect to find a fairly good i rather quickly, but we need a higher precision in the sampling. The idea is therefore to divide the sampling into a number stages, $\{S(j)\}_{j \geq 0}$, each with different sampling accuracy. Stage $S(j)$ chooses some random i-values and samples D on H^i, H^{i+1}. As soon as a sufficiently "good" i is detected, the procedure terminates. Below we quantify the needed accuracy and the criterion for selecting the good i.

For $j \in \{0, 1, \ldots, -2\log\delta'\}$ let a_j be the fraction of i such that $\delta_i \geq 2^{(j-1)/2}\delta'$. By the assumption of the lemma we have

$$a_0 + \sum_{j=1}^{\infty} a_j (2^{(j-1)/2} - 2^{(j-2)/2}) \geq 1 - 2^{-1/2}. \tag{18}$$

Define b_0 to be $\lceil 4(1 - 2^{-1/2})^{-1} \rceil$ and

$$b_j = \lceil 4(1 - 2^{-1/2})^{-1} (2^{(j-1)/2} - 2^{(j-2)/2}) \rceil = \lceil 2^{(j+3)/2} \rceil,$$

for $j > 0$. The b_j-values, together with a parameter T_j now define the sampling accuracy. Given these values, we determine i as follows.

In stage $S(j)$, $j = -2\log\delta', -2\log\delta' - 1, \ldots, 0$ choose b_j different random values of i and sample H^i and H^{i+1} each $T_j\delta'^{-2}$ times and run D on each of the samples. If the difference in the number of 1-outputs is at least $(2^{(j-1)/2}T_j - \sqrt{T_j/2})\delta'^{-1}$ choose this i and halt. If no i is ever chosen halt with failure. We need to analyse the procedure and determine T_j.

Suppose that at stage j an i is picked such that $\delta_i \geq 2^{(j-1)/2}\delta'$. We claim that the algorithm halts with this i as output with probability at least 1/2. To establish this first consider the following fact, the proof of which we leave to the reader.

Fact 8. *Let X be a random variable with mean μ and standard deviation σ. Then we have*

$$\Pr[X \leq \mu - \sigma] \leq 1/2.$$

From this, the above claim now follows since the expected difference in the number of 1-outputs when $\delta_i \geq 2^{(j-1)/2}\delta'$ is at least $2^{(j-1)/2}T_j\delta'^{-1}$ and the standard deviation (being the sum of $T_j\delta'^{-2}$ variables each being the difference of two 0/1-valued variables) is at most $\delta'^{-1}\sqrt{T_j/2}$. This implies that the probability that the algorithm halts for an individual iteration during stage j is at least $a_j/2$. The probability that algorithm will fail to output any number is thus bounded by

$$\prod_j (1 - a_j/2)^{b_j} \leq e^{-\sum_j a_j b_j/2} \leq e^{-2},$$

where the last inequality follows from (18) and the definition of b_j.

We must bound the probability that algorithm terminates with an i such that $\delta_i \leq \delta'/2$. Let us analyse the probability that such an i would be output during an individual run of stage j provided that it is chosen as a candidate. The expected difference of the number of 1-outputs in the two experiments is at most $T_j\delta'^{-1}/2$ and we have to estimate the probability that it is at least $(T_j 2^{(j-1)/2} - \sqrt{T_j/2})\delta'^{-1}$. This is, provided

$$T_j(2^{(j-1)/2} - 1/2) - \sqrt{T_j/2} \geq 0, \tag{19}$$

by a simple invocation of Chernoff bounds, at most

$$e^{-\frac{(T_j(2^{(j-1)/2}-1/2)-\sqrt{T_j/2})^2}{2T_j}}.$$

Let us call this probability p_j. The overall probability of ever outputting an i with $\delta_i \leq \delta'/2$ is bounded by

$$\sum_j b_j p_j.$$

We now define T_j to be the smallest number satisfying (19) such that $p_j < 2^{-(j+3)} b_j^{-1}$ and such that $T_j\delta'^{-2}$ is an integer. We get that with this choice the probability of outputting an i with $\delta_i \leq \delta'/2$ is at most $1/4$ and hence the probability that we do get a good output is at least $(1 - e^{-2})\frac{3}{4} \geq .64$. The total number of samples of the algorithm is bounded by $c_1\delta'^{-2}$, where

$$c_1 \triangleq 2\sum_j b_j T_j. \tag{20}$$

Note that this sum converges since $T_j \in O(j2^{-j})$ and $b_j \in O(2^{j/2})$. In fact, it can numerically be calculated to be bounded by 5300. Moreover, the sum is completely dominated by the first term which is over 4600, and the sum of all but the first three terms is bounded by 250. Thus, a more careful analysis what to do for small j could lead to considerable improvements in this constant.

Before we continue let us make some needed definitions. Let $\text{bin}(i)$ be the map that sends the integer i, $0 \leq i < 2^m$ to its binary representation as an m-bit string. In the sequel, we perform some computations in $\mathbb{F}_{2^k}$, the finite field of 2^k elements, represented as $\mathbb{Z}_2[t]/(q(t))$ where $q(t)$ is a polynomial of degree k, irreducible over $\mathbb{Z}_2$. We assume that such q is available to us. If not, it can be found in expected time at most k^4 which is negligible compared to our other running times considered. Viewing $\mathbb{F}_{2^k}$ as a vector space over $\mathbb{F}_2$, for any $\gamma = \sum_{i=0}^{k-1} \gamma_i t^i \in \mathbb{F}_{2^k}$, we let in the natural way $\text{bin}(\gamma)$ denote the vector $(\gamma_0, \ldots, \gamma_{k-1})$ corresponding to γ's representation over the standard polynomial basis. Note also that $\text{bin}(\gamma)$ can be interpreted as a subset of $[0..k-1]$ in the obvious way.

Proof of Lemma 4. Pick randomly and independently a set of m strings, $a_0, \ldots, a_{m-1}$, and k strings, $b_0, \ldots, b_{k-1}$, each of length n. The jth matrix, R^j is now defined by $\{a_i\}$, $\{b_l\}$, and an element $\alpha_j \in \mathbb{F}_{2^k}$ as follows. Its ith row, R_i^j, $0 \le i < m$, is defined by

$$R_i^j \triangleq a_i \oplus \left(\oplus_{l \in \mathrm{bin}(\alpha_j \cdot t^i)} b_l\right),$$

where α_j is the lexicographically jth element of $\mathbb{F}_{2^k}$ (i.e. the lexicographically jth binary string), and the multiplication, $\alpha_j \cdot t^i$, is carried out in $\mathbb{F}_{2^k}$, and $\oplus$ is bitwise addition mod 2.

Clearly the matrices are uniformly distributed, since the a_i are chosen at random. To show pairwise independence it suffices to show that an exclusive-or of any subset of elements from any two matrices is unbiased. Since the columns are independent, it is enough to show that the exclusive-or of any non-empty set of rows from two distinct matrices R^{j_1} and R^{j_2} is unbiased. Take such a set of rows, $S_1 \subset R^{j_1}$, and $S_2 \subset R^{j_2}$. We may actually assume that $S_1 = S_2 = S$, say, since otherwise, the a-vectors makes the result uniformly distributed. In this case the xor can be written as

$$\oplus_{i \in S} \oplus_{l \in \mathrm{bin}((\alpha_{j_1} + \alpha_{j_2}) \cdot t^i)} b_l,$$

but this is the same as

$$\oplus_{l \in \mathrm{bin}((\alpha_{j_1} + \alpha_{j_2}) \cdot (\sum_{i \in S} t^i))} b_l,$$

which is unbiased if, and only if, $\mathrm{bin}((\alpha_{j_1} + \alpha_{j_2}) \cdot (\sum_{i \in S} t^i)) \neq 0$. However, $\sum_{i \in S} t^i \neq 0$, and as $\alpha_{j_1} \neq \alpha_{j_2}$, $\alpha_{j_1} + \alpha_{j_2} \neq 0$ too, so we have two nonzero elements and hence their product is nonzero.

Notice that if we know $\sum_i a_{li} x_i$ and $\sum_i b_{li} x_i$ mod 2 for all a_l, b_l (a total of $m + k$ bits), then by the linearity of the above construction, we also know the matrix-vector products $R^j x$ for all j. To calculate all the matrices we first compute the reduction of t^i for all $i = k+1, \ldots, 2k$ in $GF[2^k]$. Using an iterative procedure this can be done with $3k$ operations on k bit words and since we only care about $k \le n$ these can be done in unit time. Now generate the vectors a and b in time $m + k$ operations. Then we compute $\oplus_{l \in bin(t^i)} b_l$ for each $i = 0, \ldots, 2k$ using k^2 operations. By using a gray-code construction each row of a matrix can now be generated with two operations and thus the total number of operations is $2m2^k + k^2 + m + 4k$.

Proof of Theorem 2. First we apply Lemma 3 to see that there is an i for which we have an algorithm $P^{(i)}$ that when given $f^{(i)}(x)$ runs in time $S(L) + T(L)$ and distinguishes $B_R^m(f^{(i-1)}(x))$ from random bits with advantage at least δ'', where δ'' is $\delta'/2$ or δ' depending on whether we want to find i efficiently, or only show existence (i.e. uniform/non-uniform algorithm). Since δ'' is an average over all x we need to do some work before we can apply the equivalent of Theorem 6 in the original paper.

For each x we have an advantage δ_x. Let a_j be the fraction of x with $\delta_j \ge 2^{(j-1)/2}\delta''$. Since the expected value of δ_x is δ'' we have

$$a_0 + \sum_{j=1}^{\infty} a_j (2^{(j-1)/2} - 2^{(j-2)/2}) \ge 1 - 2^{-1/2}. \tag{21}$$

Now define

$$d_0 \triangleq \frac{1}{2}(1 - 2^{-1/2}) \tag{22}$$

and

$$d_j \triangleq (2j(j+1)2^{(j-1)/2})^{-1} \tag{23}$$

for $j \geq 1$. Since

$$d_0 + \sum_{j=1}^{\infty} d_j (2^{(j-1)/2} - 2^{(j-2)/2}) = 1 - 2^{-1/2}, \tag{24}$$

we must have $a_j \geq d_j$ for some j and this is our choice for j in the existential part. We now apply Theorem 6 in the original paper with $\epsilon = 2^{(j-1)/2}\delta'$. To eliminate the list we apply f to each element in it to see if it is a correct preimage in which case it is output. Since whenever $\delta_x \geq \epsilon$ we have a probability $1/2$ of having $f^{(i-1)}(x)$ in the list and hence the probability of being successful for a random x is at least $d_j/2$.

To get a uniform algorithm, we need to sample to find a suitable value of j. Consider the following procedure for parameters d and T_j to be determined.

For $j = -2\log\delta'', -2\log\delta'' - 1, \ldots, 0$ choose $d(j+3)d_j^{-1}$ different random values of x and run $P^{(i)}$, for each x, $T_j\delta''^{-2}$ each on the two distributions given by choosing the m extra bits as $B_R^m(f^{(i-1)}(x))$ or as random bits. If the difference in the number of 1-outputs for the two distributions is at least $(2^{(j-1)/2}T_j - \sqrt{T_j/2})\delta''^{-1}$ for at least $d(j+3)/4$ different values, choose this j and apply the algorithm of Theorem 6 from [285] with $\epsilon = 2^{(j-2)/2}\delta'' = 2^{(j-4)/2}\delta'$.

First we analyse the probability that the algorithm outputs j if it ever gets to a stage where $a_j \geq d_j$. For each x chosen, the probability that it will satisfy $\delta_x \geq 2^{(j-1)/2}\delta''$ and yield the desired difference is by the choice of j and Fact 8, at least $a_j/2 \geq d_j/2$. Thus, for sufficiently large d, with probability at least $1 - 2^{-(j+3)}$, this desirable distance will be detected $d(j+3)/4$ times and j will be output. Hence, except with this probability the algorithm will produce some output and we have to analyse the probability that a worse j is output at an earlier stage.

We claim that unless $a_{j-1} \geq d_j/8$, the probability of j being output is $2^{-(j+3)}$. Suppose that $a_{j-1} < d_j/8$ and consider an individual execution in stage j. For a suitable choice of T_j we will prove that the probability that we observe a difference greater than $(2^{(j-1)/2}T_j - \sqrt{T_j/2})\delta''^{-1}$ is bounded by $d_j/6$. This is sufficient, for large enough d, to establish the claim.

By assumption $\delta_x \leq 2^{(j-2)/2}\delta''$ except with probability $d_j/8$ and thus we need to prove that given that this inequality is true, the probability to get the desired difference is at most $d_j/24$. By assumption the expected value of the observed difference is $2^{(j-2)/2}T_j\delta''^{-1}$, and by applying Chernoff bounds it is hence sufficient to choose T_j large enough so that

$$e^{-\frac{(T_j(2^{(j-1)/2} - 2^{(j-2)/2}) - \sqrt{T_j/2})^2}{2T_j}} \leq \frac{d_j}{24}.$$

This can be done with $T_j = O((j+3)2^{-j})$. The expected number of samples computed, given that j_0 is the largest value such that $a_{j_0} \geq d_{j_0}$, is at most

$$\sum_{j=j_0}^{\infty} d(j+3)d_j^{-1}T_j\delta''^{-2} + 2^{-(j_0+3)}\sum_{j=0}^{j_0-1} d(j+3)d_j^{-1}T_j\delta''^{-2},$$

which is $O(j_0^4 2^{-j_0/2}\delta'^{-2})$.

In the case where we efficiently find i and j, the final value of ϵ for which we call upon Theorem 6 (in [285]) is a factor $2^{-3/2}$ smaller than in the existential case, and hence the increase in the running time is increased by a factor $8 + o(1)$, where the $o(1)$ comes from the increase in the additive term k. By the above argument the guarantee for the fraction of the inputs for which the procedure has probability at least $1/2$ of finding the inverse image, is at least $1/8$ of that in the existential case.

Some modes of use of the GPS identification scheme

by Marc Girault, Guillaume Poupard and Jacques Stern *

Abstract. In this note, we present various modes of use of the GPS identification scheme, so that the reader can better evaluate the potentialities of this scheme. Some of them are not new, while other ones result from very recent research[1].

Keywords. Identification, digital signatures, discrete logarithm, on-line/off-line, on the fly, RSA, server-aided verification.

1 Introduction

GPS is an identification scheme, which has been submitted for evaluation to NESSIE project. In essence, it is a (statistically) zero-knowledge protocol based on both discrete logarithm and integer factorisation. As in many other discrete-logarithm-based schemes, GPS can be used in an *on-line/off-line* manner [222]: almost all the computations can be performed by the prover before the interaction with the verifier. But contrary to all the other discrete-logarithm-based schemes, it can be used in an *on the fly* [526] manner: the prover only has one multiplication and one addition to do, *without any modular reduction*, after he received the challenge from the verifier.

* Marc.Girault@francetelecom.com
France Telecom R&D, 42 rue des Coutures,
BP 6243, F-14066 Caen Cedex 4, France.

Guillaume.Poupard@m4x.org
DCSSI Crypto Lab, 51 boulevard de La Tour-Maubourg,
75700 Paris 07 SP, France.

Jacques.Stern@ens.fr
École normale supérieure, Département d'informatique,
45 rue d'Ulm, F-75230 Paris Cedex 05, France.

[1] Actually, this note should be viewed as a complement of the NESSIE submission document, and therefore does not replicate many contents of this document, such as motivation, notations, ways of achieving short coupons etc. It should also be seen as a review, rather than a research paper; in particular, security proofs are outside of its scope.

In this note, we wish to point out another important advantage of GPS, namely its great *flexibility*, due to its mathematical specific features[2], but leading to very practical consequences. We illustrate this flexibility by presenting several variants and/or options of the basic GPS scheme, called modes of use, and which will allow the reader to better evaluate the potentialities of this scheme. In brief, we show how to (1) convert it into a digital signature scheme, (2) use it in another group where computing discrete logarithms is hard ($\mathbb{Z}_p^*$ with p prime, elliptic curves, etc.), (3) accelerate the verification with the aid of a (possibly untrusted) server, (4) make it at least as secure as RSA, (4bis) make it at least as secure as RSA using a public key identical to a RSA public key, (4ter) make it at least as secure as RSA using a key pair identical to a RSA key pair.

Note that (1), (2), (4) and (4bis) are not new while (3) is dated 2002 (rump session of Eurocrypt 2002 [264]) and (4ter) is dated 2003 [261]. The latter mode of use can be viewed as an *on the fly* variant of RSA (in the sense that its key pairs are the same as RSA ones), which naturally integrates the server-aided verification option (in the sense that no extra parameter or key is required to offer this option).

We also recall the extensive use of this scheme (even though rarely referred to) in other areas than identification/signature, such as group signatures, electronic cash, fair encryption etc., more generally in many protocols which use complex proofs of knowledge.

2 Basic GPS scheme

We first recall in Fig. 91 the basic GPS identification scheme, as described in the NESSIE submission "*GPS, an asymmetric identification scheme for on the fly authentication of low cost smart cards, version 2.0 (October 12, 2001)*". The security of this scheme is based on the intractability of extracting ("short" if $S < n$) discrete logarithms. (Note that, since n is composite, this problem is closely related to factorisation problem, see section 3.4). Assuming that, the following properties can be proven [526]:

- an honest user is always accepted
- given a public key v, if an attacker is accepted with non-negligible probability, then he can be used to efficiently compute discrete logarithms modulo n in base g (in other words, if we assume that the discrete logarithm problem is intractable, such attacks cannot exist)
- even if a prover is identified many times, essentially no information about his secret can be learned by passive eavesdroppers or cheating verifiers.

Typically:

- n is at least 1024-bit long, so that factorisation algorithms are inefficient[3]

[2] Mainly: it can be implemented as it is in any (finite) group, and when this group is $\mathbb{Z}_n^*$, it may rely on a harder problem than factorisation.

[3] Actually, we must distinguish two cases: 1) n is a "universal" or "system" parameter, i.e. the same for all the users of a common system or application; 2) n is a "user"

Parameters: n a composite modulus
$g \in \mathbb{Z}_n^*$
A, B and S three integers such that $A >> B.S$

Secret key: $s \in [0, S[$
Public key: $v = g^{-s} \pmod{n}$

Prover		**Verifier**
choose $r \in [0, A[$ compute $x = g^r \pmod{n}$		
	$\xrightarrow{x}$	
		choose $c \in [0, B[$
	$\xleftarrow{c}$	
check $c \in [0, B[$ compute $y = r + cs$		
	$\xrightarrow{y}$	
		check $y \in [0, A + (B-1)(S-1)[$ and $g^y v^c = x \pmod{n}$

Fig. 91. Basic GPS identification scheme.

- s is at least 160-bit long (i.e. $S \geq 2^{160}$) so that discrete logarithm algorithms are inefficient
- c is 16, 32 or 64-bit long (i.e. $B = 2^{16}$, 2^{32} or 2^{64})
- $>>$ means "64 or 80-bit more"
- $g = 2$, in order to speed-up exponentiations.

More precisely, n and g must be chosen so that the order of g modulo n be "sufficiently" large (and the discrete logarithm modulo n in base g a "sufficiently" hard problem). Preferably, g will be of order close or equal to the maximum possible order $\lambda(n)$, where $\lambda(n)$ denotes the "Carmichaël function" of n. For example, we will choose n as the product of two distinct large safe primes p and q [4], and g of order $\lambda(n) = \frac{(p-1)(q-1)}{2}$ or $\frac{1}{2}\lambda(n)$. It happens that, for such a choice of n, "almost all" integers between 2 and n are such "good" values of g, and that a very simple test allows to check it, namely : $gcd(g-1, n) = gcd(g+1, n) = 1$. Moreover, $g = 2$ is *always* good (since $gcd(1, n) = gcd(3, n) = 1$).

When n is 1024-bit long, s 160-bit and c 64-bit, the following performances have been demonstrated :

- y can be computed in less than 1 millisecond by a low-cost microprocessor card at 10 MHz frequency (in a negligible time by a crypto-processor card or a personal computer)
- the computation of x and the verification can be done in less than 100 milliseconds by a crypto-processor card or a personal computer at 500 MHz frequency

parameter, i.e. is distinct for each user and is part of his public key. In the first case, n should be chosen still larger, e.g. 2048 bits. In the second case, we can assume that the user knows the factorisation of n, which allows him to speed-up the computation of x by using the well-known Chinese Remainder Technique.

[4] A prime p is safe if $\frac{p-1}{2}$ is prime.

with standard Java libraries, and less than 10 milliseconds in a personal computer with specific C libraries[5].

3 Six modes of use of GPS

Now, we describe six variants and/or options of GPS. Note that all these modes of use are, to some extent, independent to each other, and consequently several of them can be combined (e.g. 3.1 with any other one, 3.3 with 3.4.1 or 3.4.2 etc.).

3.1 Digital signature scheme

By using a standard method from Fiat and Shamir [229], GPS identification scheme can be easily turned into a digital signature scheme as shown in Fig. 92 (we denote by H a collision-free hash-function, of output less than B, and by m the message to be signed). The same sizes/values of parameters can be used, except c, which should be at least 160-bit long, in order to prevent from finding collisions on H by using a birthday attack.

Parameters: n a composite modulus
$g \in \mathbb{Z}_n^*$
A, B and S three integers such that $A >> B.S$

Secret key: $s \in [0, S[$
Public key: $v = g^{-s} \pmod n$

Prover
choose $r \in [0, A[$
compute $x = g^r \pmod n$
compute $c = H(x, m)$
compute $y = r + cs$

$$\xrightarrow{c,y}$$

Verifier
check:
$c \in [0, B[$
$y \in [0, A + (B-1)(S-1)[$ and
$H(g^y v^c \pmod n, m) = c$

Fig. 92. GPS signature scheme.

Let us recall [526] that if the hash function H is replaced by a random function and if an attacker is able to forge a valid signature under an adaptively chosen message attack, then he must be able to compute discrete logarithms modulo n in base g. Once again, if we assume the intractability of short discrete logarithms, then forgery of valid signatures is infeasible.

[5] Even without using the Chinese Remainder Technique.

3.2 Using other groups

Actually, the exponentiations involved in GPS can be performed in any (finite) group where computing discrete logarithms is assumed to be hard, no matter its order is known or not. This allows variants using groups with interesting cryptographic properties, such as the multiplicative group of $\mathbb{Z}_p$, elliptic curves or even class groups of imaginary quadratic orders [129].

3.2.1 Schnorr *on the fly*

If we use the group $\mathbb{Z}_p^*$, with p a prime, instead of $\mathbb{Z}_n^*$, with n a composite integer, then we obtain a variant of GPS, which is solely based on the discrete logarithm problem. Moreover, if the parameters are chosen in the same way as those of the Schnorr scheme [560], then we obtain a scheme which is compatible with Schnorr's one in that the verification equation is exactly the same (and which consequently can be considered as the *on the fly* version of the Schnorr scheme, in that the computation of y involves no modular reduction).

Parameters:	p a prime q a prime such that $q \vert p-1$ g of order q modulo p A, B two integers such that $A >> B.q$	
Secret key:	$s \in [0, q[$	
Public key:	$v = g^{-s} \pmod p$	

Prover		**Verifier**
choose $r \in [0, A[$ compute $x = g^r \pmod p$		
	$\xrightarrow{x}$	
		choose $c \in [0, B[$
	$\xleftarrow{c}$	
check $c \in [0, B[$ compute $y = r + cs$		
	$\xrightarrow{y}$	
		check: $y \in [0, A + (B-1)(q-1)[$ [optionally reduce y mod q] check $g^y v^c = x \pmod p$

Fig. 93. Schnorr *on the fly* identification scheme.

3.2.2 Elliptic curve variant

Since the group operation is traditionally denoted by an addition in elliptic curves, we can (straight-forwardly) rewrite the elliptic-curve version of GPS in Fig. 94. The elliptic curves to be recommended are exactly the same as for any other discrete-logarithm-based scheme. For example, an elliptic curve on the field $\mathbb{Z}_p$, with p a (at least) 160-bit prime may be an appropriate choice. We refer the reader to the abundant literature on this topic.

Parameters: $(EC, +)$ an elliptic curve
G a point of EC
A, B and S three integers such that $A >> B.S$

Secret key: $s \in [0, S[$
Public key: $V = -sG$

Prover		**Verifier**
choose $r \in [0, A[$ compute $X = rG$		
	$\xrightarrow{X}$	
		choose $c \in [0, B[$
	$\xleftarrow{c}$	
check $c \in [0, B[$ compute $y = r + cs$		
	$\xrightarrow{y}$	
		check: $y \in [0, A + (B-1)(S-1)[$ and $yG + cV = X$

Fig. 94. GPS elliptic-curve identification scheme.

It has also been proposed to benefit from the fact that knowing the cardinality of the group in which GPS works is not required, by using elliptic curves without necessary counting their cardinality [152]. However the efficiency of modern counting algorithms is such that those variants are today of little interest.

3.3 Server-aided verification

Verification step is not so fast in basic GPS scheme, since it involves an exponentiation with a somewhat large exponent. In contrast, verification is fast in the GQ (Guillou-Quisquater) identification scheme [277].

A key observation made in [264] is that GPS verification step can be transformed into a GQ verification step, provided the exponentiation can be delegated to a (powerful) server. This implies to add at least a new (public) parameter to those already existing, namely the "GQ exponent". Another observation is that this server can be any third party, even an untrusted one. This is useful in environments where (a) the whole transaction must be very rapid, (b) the (secure) verification chip is not powerful enough but is embedded in a device including another (insecure but powerful) chip. Such a situation may occur e.g. in a card-reader device or in a mobile telephone.

The resulting scheme is described in Fig. 95. The security of this option is based on the intractability of the RSA problem if the order of g is close to the maximal order modulo n and if s is "full-size" (see section 3.4.1). The basic idea of the proof is that a collusion between a dishonest prover and a third party able to be accepted by an honest verifier can be used as an extractor of e^{th} roots modulo n (and therefore a RSA forger).

Parameters:	n a composite modulus e a prime integer $f, g \in \mathbb{Z}_n^*$ with $g = f^e \pmod n$ A, S two integers such that $A >> e.S$
Secret key:	$s \in [0, S[$
Public key:	$v = g^{-s} \pmod n$

Prover		**Verifier**		**Third party**
choose $r \in [0, A[$ compute $x = g^r \pmod n$				
	$\xrightarrow{x}$			
		choose $c \in [0, e[$		
	$\xleftarrow{c}$			
check $c \in [0, e[$ compute $y = r + cs$				
	$\xrightarrow{y}$			
		check: $y \in [0, A + (e-1)(S-1)[$ (all the operations below are performed modulo n)		
			$\xrightarrow{y}$	
				$Y = f^y$
			$\xleftarrow{Y}$	
		check $Y^e f^c = x$		

Fig. 95. GPS, server-aided verification mode of use.

.

3.4 RSA security as a minimum

We now focus on versions which are at least as secure as RSA. Note that basic GPS scheme is not necessarily so, since the underlying assumption is the hardness of computing a short ($< S$) discrete logarithm, as opposed to a full-size discrete logarithm, which is not (a priori) harder than factorisation nor e^{th} root modulo n problem.

3.4.1 Full-size variant

Of course, the most straight-forward way to make GPS at least as secure as RSA consists to choose a full-size secret s, i.e. to choose S in the order of magnitude of n. In that case, the underlying problem is computing a (full-size) discrete logarithm modulo n, which is well known to be at least as hard than factorisation and therefore at least as hard as RSA.

Note that, since the data r and y are much greater than n in such a set-up, they could be reduced modulo $\lambda(n)$ [6] – in case n is a user parameter, see Section 2 – so as to speed up the computation of x and/or the verification step. On the other hand, computation of y will be substantially slower since there is now a modular reduction to perform.

3.4.2 Half-size variant

In [529], the authors present a variant of GPS, whose security is equivalent to factorisation, while the secret key is only half the size of n and the public key is reduced to a RSA modulus (see Fig. 96). This is achieved by choosing a particular value of s (as a function of n), instead of being a random integer in some prescribed interval. Of course, this set-up implies that n is a user parameter and not a universal one. Let us recall the main properties of this scheme:

- an honest user is accepted with overwhelming probability
- given a public key n, if an attacker is accepted with non-negligible probability, then he can be used to efficiently factor n. In other words, if we assume that factoring large integers is intractable, such attacks cannot exist
- even if a prover is identified many times, no information about his secret can be learned by eavesdroppers or verifiers.

3.4.3 RSA *on the fly*

We end with a mode of use which is in some sense the convergence of modes presented in sections 3.3 and 3.4.2 (see Fig. 97). The result [261] can be viewed as an *on the fly* version of RSA, in that keys of this mode coincide with RSA keys (not in the sense that the signature would be a RSA signature). Moreover, it is at least as secure as RSA. These features make it a good alternative to RSA in environments where generating a RSA signature is too slow, without having to change the keys (hence without having to change the key management primitives: key generation, key certificates etc.), and without taking any security risk. There is only one particular extra-requirement: the RSA-like public exponent must be large enough, as the level of security will be dependent on its value. This is not

[6] Or modulo $\varphi(n)$, the Euler function of n.

Parameters: n a composite modulus
$g \in \mathbb{Z}_n^*$
A, B two integers such that $A >> B.\sqrt{n}$

Secret key: $s = n - \varphi(n)$
Public key: n

Prover		**Verifier**
choose $r \in [0, A[$ compute $x = g^r \pmod{n}$		
	$\xrightarrow{x}$	
		choose $c \in [0, B[$
	$\xleftarrow{c}$	
check $c \in [0, B[$ compute $y = r + cs$		
	$\xrightarrow{y}$	
		check: $y \in [0, A + 4B\sqrt{n}[$ and $g^{y-nc} = x \pmod{n}$

Fig. 96. Half-size variant.

always a restriction: 65537 is a very common RSA exponent, while it may provide an adequate level of security in zero-knowledge authentication.

In addition, the fact that a public exponent is already part of the public key allows to integrate the server-aided verification option without modifying the set-up: the decision to use (option 2) or not (option 1) this possibility can be made at the very last moment by the verifier, and it therefore needs not be anticipated. With option 1, the security of the scheme is equivalent to factorisation, while in option 2 it is equivalent to RSA. It is worth rephrasing this scheme by choosing $f = 2$, which implies $g = 2^e \pmod{n}$. The description of the scheme becomes Fig. 98.

4 Using GPS as a proof of knowledge

The basic GPS scheme has been extensively used in many areas. Mainly, this is because one often has to prove the knowledge of a discrete logarithm in a group the order of which one ignores. GPS is essentially the only scheme to achieve that. Another reason is that the absence of modular reduction naturally allows to prove that two different exponentials, possibly computed in different groups, have the same discrete logarithm. We now briefly recall some of these applications.

4.1 Bounded range commitment

In 1989, Schnorr [560] proposed his famous signature scheme, which may be viewed as a proof of knowledge of a discrete logarithm modulo a prime number. Since then, many authors have tried to adapt the scheme in order to get control

Parameters: n a composite modulus
e a prime integer and d an integer such that $ed = 1 \pmod{\varphi(n)}$
$f, g \in \mathbb{Z}_n^*$ with $g = f^e \pmod n$
A an integer such that $A >> e.n$

Secret key: d
Public key: (n, e)

Prover		**Verifier**		**Third party**
choose $r \in [0, A[$ compute $x = g^r \pmod n$				
	$\xrightarrow{x}$			
		choose $c \in [0, e[$		
	$\xleftarrow{c}$			
check $c \in [0, e[$ compute $y = r - cd$				
	$\xrightarrow{y}$			
		check $y \in [-(e-1)(n-2), A[$ (all the operations below are performed modulo n) **Option 1** check $g^y f^c = x$ **Option 2**		
			$\xrightarrow{y}$	$Y = f^y$
			$\xleftarrow{Y}$	
		check $Y^e f^c = x$		

Fig. 97. RSA *on the fly.*

Parameters: n a composite modulus
e a prime integer and d an integer such that $ed = 1 \pmod{\varphi(n)}$
A an integer such that $A >> e.n$

Secret key: d
Public key: (n, e)

Prover		**Verifier**		**Third party**
choose $r \in [0, A[$ compute $x = 2^{er} \pmod n$				
	$\xrightarrow{x}$			
		choose $c \in [0, e[$		
	$\xleftarrow{c}$			
check $c \in [0, e[$ compute $y = r - cd$				
	$\xrightarrow{y}$			
		check $y \in [-(e-1)(n-2), A[$ (all the operations below are performed modulo n)		
		Option 1 check $2^{ey+c} = x$		
		Option 2		
			$\xrightarrow{y}$	
				$Y = 2^y$
			$\xleftarrow{Y}$	
		check $Y^e 2^c = x$		

Fig. 98. RSA *on the fly* (with f=2).

over the size of the secret value. Such a bounded-range commitment has many applications and has been used for group signature by Camenisch and Michels [132], electronic cash by Chan, Frankel and Tsiounis [139], verifiable secret sharing by Fujisaki and Okamoto [240] and finally for proving that a modulus is the product of two safe primes by Camenisch and Michels [133]. Furthermore, Boudot [117] proposed efficient proofs that a committed number lies in an interval.

All those proposals use the GPS scheme even if most of them do not refer to. Except for Boudot's result, these protocols are only able to prove that the discrete logarithm is not "too far" from a fixed range. Their analysis is complex (and sometimes erroneous as in the Eurocrypt '98 version of [133] or in [32]) and their security is often based on non-standard assumptions such as the so-called "*strong RSA assumption*" needed to make proofs efficient [35, 240].

4.2 Fair encryption of asymmetric secret keys

The GPS scheme has also been used in [528] and [118] to design proofs of fair encryption of secret keys, for any encryption scheme based on the discrete logarithm problem or on the intractability of the factorisation, including RSA and its variants. The asymmetric secret keys can be encrypted using a homomorphic public key cryptosystem like Paillier's scheme [510].

More precisely, the protocol proposed in [528] allows proving that a ciphertext enables a third party to recover the El Gamal secret key related to a public one. Such a proof is very short and the workload of the third party during recovery is very small. A scheme for fair encrypting the factorisation of a public modulus is also described.

4.3 Short proof of knowledge for factoring

Proofs of knowledge for the factorisation of an integer n have been known for a long time. But, even if they are claimed efficient according to complexity theoretical arguments, none of them can be considered practical for many applications because of their significant communication complexity: the proof is much longer than the object it deals with. A new strategy has been used in [529]. The protocol is a proof of knowledge of a small common discrete logarithm of $(z^n \pmod{n})$ for a few randomly chosen elements z modulo n. This scheme is very efficient: only three modular exponentiations both for the prover and the verifier are needed to obtain a very high level of security.

The scheme can be viewed as a parallelised version of the PS scheme [527] with a few randomly chosen bases instead of just one.

5 Conclusion

We have presented various modes of use of the identification scheme GPS. Some of them can be viewed as particular instances of the basic scheme, such as submitted to NESSIE project, and have the following (and unique to our knowledge) attractive feature: they combine factorisation-based security and *on the fly* computation. Moreover, they all support the so-called "server-aided verification" option, which is of interest when transaction time is a critical parameter. Other ones are transpositions to other groups than $\mathbb{Z}_n^*$, which offer an alternative to factorisation as an underlying hard problem. All of them can be converted into digital signature schemes using the Fiat-Shamir heuristic. Finally the wide utilisation of GPS in other contexts than identification and signature has also been demonstrated. All these modes of use give strong evidence of the great flexibility and the numerous potentialities of the GPS scheme.

Acknowledgements

We acknowledge the contribution of Jean-Jacques Quisquater and Jean-Claude Paillès (to resp. schemes 3.3 and 3.4.3). We thank the anonymous reviewer for pertinent remarks and corrections, and Sébastien Canard for helping editing the LaTeX version.

Book V

References

1. "3GPP Standard.". Available from http://www.3gpp.org/. [p. 501]
2. M. Abdalla, M. Bellare, and P. Rogaway, "DHAES: An encryption scheme based on the Diffie-Hellman problem." Technical report, Submitted to IEEE P1363, 1998. Available from http://grouper.ieee.org/groups/1363/P1363a/Encryption.html. [p. 793]
3. M. Abdalla, M. Bellare, and P. Rogaway, "The Oracle Diffie-Hellman assumptions and an analysis of DHIES." 2001. Available at http://www-cse.ucsd.edu/users/mihir/papers/dhies.html. Earlier version in [2]. [p. 239]
4. M. Abdalla and L. Reyzin, "A new forward-secure digital signature scheme." in *Proceedings of Asiacrypt'00* (T. Okamoto, ed.), no. 1976 in Lecture Notes in Computer Science, pp. 116–129, Springer-Verlag, 2000. Full version available at http://eprint.iacr.org/2000/002/. [p. 262]
5. M. Abe and T. Okamoto, "A signature scheme with message recovery as secure as discrete logarithms." in *Proceedings of Asiacrypt'99* (K.-Y. Lam, E. Okamoto, and C. Xing, eds.), no. 1716 in Lecture Notes in Computer Science, pp. 378–389, Springer-Verlag, 1999. [p. 282, 290]
6. G. B. Agnew, R. C. Mullin, and S. A. Vanstone, "A fast elliptic curve cryptosystem." in *Proceedings of Eurocrypt'89* (J.-J. Quisquater and J. Vandewalle, eds.), vol. 434 of *Lecture Notes in Computer Science*, pp. 706–708, Springer-Verlag, 1990. [p. 393]
7. G. B. Agnew, R. C. Mullin, and S. A. Vanstone, "On the development of a fast elliptic curve cryptosystem." in *Proceedings of Eurocrypt'92* (R. A. Rueppel, ed.), no. 658 in Lecture Notes in Computer Science, pp. 482–487, Springer-Verlag, 1992. [p. 393]
8. G. B. Agnew, R. C. Mullin, and S. A. Vanstone, "An implementation of elliptic curve cryptosystem over $f_{2^{155}}$." *IEEE Journal on Selected Areas in Communications*, vol. 11, no. 5, pp. 804–813, June 1993. [p. 393]
9. D. Agrawal, B. Archambeault, J. R. Rao, and P. Rohatgi, "The EM side-channel(s)." in *Proceedings of CHES'02* (B. S. Kaliski, Çetin Kaya Koç, and C. Paar, eds.), no. 2535 in Lecture Notes in Computer Science, Springer-Verlag, 2002. [p. 339]
10. M. Aigner and E. Oswald, "Power analysis tutorial." Technical report, IAIK, TU-GRAZ, 2002. Available at http://www.iaik.tugraz.at/aboutus/people/oswald/papers/dpa_tutorial.pdf. [p. 338]
11. M.-L. Akkar, R. Bevan, P. Dischamp, and D. Moyart, "Power analysis, what is now possible." in *Proceedings of Asiacrypt'00* (T. Okamoto, ed.), no. 1976 in Lecture Notes in Computer Science, pp. 489–502, Springer-Verlag, 2000. [p. 338]
12. M.-L. Akkar, N. T. Courtois, R. Duteuil, and L. Goubin, "A fast and secure implementation of SFLASH." in *Proceedings of Public Key Cryptography – PKC'03* (Y. Desmedt, ed.), no. 2567 in Lecture Notes in Computer Science, pp. 267–278, Springer-Verlag, 2003. Also in *Proceedings of the Third NESSIE Workshop*, 2002. [p. 309, 340, 670, 673, 676, 677]
13. Algorithms group of the EESSI-SG, "Elliptic Curve German Digital Signature Algorithm." Described in [310], 1999. [p. 289]
14. J. H. An, Y. Dodis, and T. Rabin, "On the security of joint signature and encryption." in *Proceedings of Eurocrypt'02* (L. R. Knudsen, ed.), no. 2332 in Lecture Notes in Computer Science, pp. 83–107, Springer-Verlag, 2002. [p. 261]
15. R. J. Anderson, "Two remarks on public key cryptology (invited lecture)." in *Conference on Computer and Communications Security – CCS'97*, 1997. Summary available at http://www.cl.cam.ac.uk/users/rja14/. [p. 262]
16. R. J. Anderson and S. Skorobogatov, "Optical fault induction attacks." in *Proceedings of CHES'02* (B. S. Kaliski, Çetin Kaya Koç, and C. Paar, eds.), no. 2535 in Lecture Notes in Computer Science, Springer-Verlag, 2002. Also available from http://www.cl.cam.ac.uk/~sps32/. [p. 344]

17. R. Anderson, E. Biham, and L. R. Knudsen, "Serpent: A flexible block cipher with maximum assurance." in *Proceedings of the First Advanced Encryption Standard Conference*, 1998. [p. 749, 750]
18. ANSI X9.19, "Financial institution retail message authentication." American Bankers Association, 1996. [p. 201]
19. ANSI X9.30.2-1997, "Public key cryptography for the financial services industry — Part 2: The Secure Hash Algorithm (SHA-1)." American Bankers Association, 1997. [p. 180]
20. ANSI X9.31-1998, "Public key cryptography using reversible algorithms for the financial services industry (rDSA)." American Bankers Association, 1998. [p. 180, 302, 706]
21. ANSI X9.62, "Public key cryptography for the financial services industry: the elliptic curve digital signature algorithm (ECDSA)." American Bankers Association, 1999. [p. 302, 665]
22. ANSI X9.71, "Keyed hash message authentication code (MAC)." American Bankers Association, 2000. [p. 201]
23. A. Antipa, D. R. L. Brown, A. Menezes, R. Struik, and S. A. Vanstone, "Validation of elliptic curve public keys." in *Proceedings of Public Key Cryptography – PKC'03* (Y. Desmedt, ed.), no. 2567 in Lecture Notes in Computer Science, pp. 211–223, Springer-Verlag, 2003. [p. 223, 346, 634, 646]
24. K. Aoki, T. Ichikawa, M. Kanda, M. Matsui, S. Moriai, J. Nakajima, and T. Tokita, "Camellia: A 128-bit block cipher suitable for multiple platforms." Primitive submitted to NESSIE by NTT Corp., Sept. 2000. See also `http://info.isl.ntt.co.jp/camellia/` and [25]. [p. 37, 52, 103, 105, 357, 390, 395, 527, 715, 717, 718]
25. K. Aoki, T. Ichikawa, M. Kanda, M. Matsui, S. Moriai, J. Nakajima, and T. Tokita, "Camellia: A 128-bit block cipher suitable for multiple platforms – design and analysis." in *Proceedings of Selected Areas in Cryptography – SAC'00* (D. R. Stinson and S. E. Tavares, eds.), no. 2012 in Lecture Notes in Computer Science, pp. 39–56, Springer-Verlag, 2001. [p. 105, 528, 715, 794]
26. A. O. L. Atkin and F. Morain, "Elliptic curve and primality proving." *Mathematics of Computation*, vol. 61, pp. 29–68, 1993. [p. 697]
27. S. Babbage, "Improved "exhaustive search" attacks on stream ciphers." in *ECOS 95 (European Convention on Security and Detection)*, no. 408 in IEE Conference Publication, May 1995. [p. 708, 709]
28. S. Babbage, "Cryptanalysis of the LILI-128 stream cipher." in *Proceedings of the Second NESSIE Workshop*, 2001. `NES/DOC/EXT/WP3/001`. [p. 169]
29. S. Babbage, "Simple attack on BMGL faster than exhaustive key search." 23 Jan. 2002. NESSIE discussion forum. [p. 160]
30. S. Babbage and J. Lano, "Probabilistic factors in the SOBER-t stream ciphers." in *Proceedings of the Third NESSIE Workshop*, 2002. [p. 166]
31. F. Bahr, J. Franke, and T. Kleinjung, "2.953+ c158 is factorized.". See `http://www.loria.fr/~zimmerma/records/gnfs158`. [p. 270]
32. F. Bao, "An efficient verifiable encryption scheme for encryption of discrete logarithms." in *Proceedings of CARDIS'98* (J.-J. Quisquater and B. Schneier, eds.), no. 1820 in Lecture Notes in Computer Science, pp. 213–220, Springer-Verlag, 2000. [p. 788]
33. F. Bao, R. H. Deng, Y. Han, A. B. Jeng, A. D. Narasimhalu, and T.-H. Ngair, "Breaking public key cryptosystems on tamper resistant devices in the presence of transient faults." in *Proceedings of Security Protocols Workshop 1997* (B. Christianson, B. Crispo, T. M. A. Lomas, and M. Roe, eds.), no. 1361 in Lecture Notes in Computer Science, pp. 115–124, Springer-Verlag, 1998. [p. 305, 345]
34. F. Bao, R. H. Deng, M. Joye, and J.-J. Quisquater, "RSA-type signatures in the presence of transient faults." in *Proceedings of Cryptography and Coding – CC'97*

(M. Darnell, ed.), no. 1355 in Lecture Notes in Computer Science, pp. 155–160, Springer-Verlag, 1997. [p. 345]

35. N. Baric and B. Pfitzmann, "Collision-free accumulators and fail-stop signature schemes without trees." in *Proceedings of Eurocrypt'97* (W. Fumy, ed.), no. 1233 in Lecture Notes in Computer Science, pp. 480–494, Springer-Verlag, 1997. [p. 788]
36. E. Barkan and E. Biham, "In how many ways can you write Rijndael?." in *Proceedings of Asiacrypt'02* (Y. Zheng, ed.), no. 2501 in Lecture Notes in Computer Science, pp. 160–175, Springer-Verlag, 2002. `NES/DOC/TEC/WP5/025`. Also in *Proceedings of the Third NESSIE Workshop*, 2002. [p. 115, 116]
37. P. S. L. M. Barreto and V. Rijmen, "The Anubis block cipher." Primitive submitted to NESSIE, Sept. 2000. [p. 37, 52, 130, 131, 148, 357]
38. P. S. L. M. Barreto and V. Rijmen, "The Whirlpool hashing function." Primitive submitted to NESSIE, Sept. 2000. [p. 38, 53, 180, 181, 358, 563, 566]
39. P. S. L. M. Barreto and V. Rijmen, "The Khazad legacy-level block cipher." Primitive submitted to NESSIE, Sept. 2000. [p. 37, 52, 84, 86, 87, 138, 357]
40. P. S. L. M. Barreto and V. Rijmen, "Recursive S-boxes for Anubis, Khazad, and Whirlpool." Private report, NESSIE, 2001. [p. 130]
41. P. S. L. M. Barreto, V. Rijmen, J. Nakahara, Jr, B. Preneel, J. Vandewalle, and H. Y. Kim, "Improved SQUARE attacks against reduced-round Hierocrypt." in *Proceedings of Fast Software Encryption – FSE'01* (M. Matsui, ed.), no. 2355 in Lecture Notes in Computer Science, pp. 165–173, Springer-Verlag, 2001. `NES/DOC/KUL/WP3/005`. [p. 75, 126, 127, 132, 133, 147, 148]
42. O. Baudron, H. Gilbert, L. Granboulan, H. Handschuh, A. Joux, P. Q. Nguyen, F. Noilhan, D. Pointcheval, T. Pornin, G. Poupard, J. Stern, and S. Vaudenay, "Report on the AES candidates." in *Proceedings of the Second Advanced Encryption Standard Conference*, pp. 53–67, NIST, 1999. Available at `http://csrc.nist.gov/encryption/aes/round1/conf2/papers/baudron1.pdf`. [p. 69, 111]
43. M. Bellare, A. Boldyreva, and S. Micali, "Public-key encryption in a multi-user setting: Security proofs and improvements." in *Proceedings of Eurocrypt'00* (B. Preneel, ed.), no. 1807 in Lecture Notes in Computer Science, pp. 259–274, Springer-Verlag, 2000. Full paper available at `http://www-cse.ucsd.edu/users/mihir/papers/musu.html`. [p. 263]
44. M. Bellare, R. Canetti, and H. Krawczyk, "Keying hash functions for message authentication." in *Proceedings of Crypto'96* (N. Koblitz, ed.), no. 1109 in Lecture Notes in Computer Science, pp. 1–15, Springer-Verlag, 1996. Also available at `http://www-cse.ucsd.edu/users/mihir/papers/hmac.html`. [p. 200, 210, 623, 624, 820]
45. M. Bellare, A. Desai, D. Pointcheval, and P. Rogaway, "Relations among notions of security for public-key encryption schemes." in *Proceedings of Crypto'98* (H. Krawczyk, ed.), no. 1462 in Lecture Notes in Computer Science, pp. 26–45, Springer-Verlag, 1998. Also available at `http://www-cse.ucsd.edu/users/mihir/papers/relations.html`. [p. 218]
46. M. Bellare and S. Micali, "How to sign given any trapdoor function." in *Proceedings of Crypto'88* (S. Goldwasser, ed.), no. 403 in Lecture Notes in Computer Science, pp. 200–215, Springer-Verlag, 1988. Also appeared in [47, 48]. [p. 275]
47. M. Bellare and S. Micali, "How to sign given any trapdoor function (extended abstract)." in *Proceedings of Symposium on Theory of Computing – STOC'88*, ACM Press, 1988. Full paper in [48]. [p. 795]
48. M. Bellare and S. Micali, "How to sign given any trapdoor function." *Journal of the ACM*, vol. 39, pp. 214–233, Jan. 1992. Also available at `http://www-cse.ucsd.edu/users/mihir/papers/ds.html`. [p. 795]
49. M. Bellare and S. Miner, "A forward-secure digital signature scheme." in *Proceedings of Crypto'99* (M. J. Wiener, ed.), no. 1666 in Lecture Notes in Computer Science, pp. 431–448, Springer-Verlag, 1999. Also available at `http://www-cse.ucsd.edu/users/mihir/papers/fsig.html`. [p. 262]

50. M. Bellare and C. Namprempre, "Authenticated encryption: relations among notions and analysis of the generic composition paradigm." in *Proceedings of Asiacrypt'00* (T. Okamoto, ed.), no. 1976 in Lecture Notes in Computer Science, pp. 531–545, Springer-Verlag, 2000. Full paper available at http://eprint.iacr.org/2000/025/. [p. 261]
51. M. Bellare and A. Palacio, "GQ and Schnorr identification schemes: Proofs of security against impersonation under active and concurrent attacks." in *Proceedings of Crypto'02* (M. Yung, ed.), no. 2442 in Lecture Notes in Computer Science, pp. 162–177, Springer-Verlag, 2002. Full paper available at http://www-cse.ucsd.edu/users/mihir/papers/gq.html. [p. 321, 336]
52. M. Bellare and P. Rogaway, "Random oracles are practical: A paradigm for designing efficient protocols." in *Proceedings of Conference on Computer and Communications Security – CCS'93*, pp. 62–73, ACM Press, Nov. 1993. Full paper available at http://www-cse.ucsd.edu/users/mihir/papers/ro.html. [p. 222, 272, 275, 276, 278, 279]
53. M. Bellare and P. Rogaway, "Optimal asymmetric encryption – how to encrypt with RSA." in *Proceedings of Eurocrypt'94* (A. De Santis, ed.), no. 950 in Lecture Notes in Computer Science, pp. 92–111, Springer-Verlag, 1995. Also available at http://www-cse.ucsd.edu/users/mihir/papers/oaep.html. [p. 256, 257, 272]
54. M. Bellare and P. Rogaway, "The exact security of digital signature – how to sign with RSA and Rabin." in *Proceedings of Eurocrypt'96* (U. Maurer, ed.), no. 1070 in Lecture Notes in Computer Science, pp. 399–416, Springer-Verlag, 1996. Revised version available at http://www-cse.ucsd.edu/users/mihir/papers/exactsigs.html. [p. 272, 277, 279, 280, 310, 657]
55. D. J. Bernstein, "Circuits for integer factorization." Web page. http://cr.yp.to/nfscircuit.html. [p. 269]
56. R. Bevan and E. Knudsen, "Ways to enhance differential power analysis." in *Proceedings of ICISC'02* (K. Kim, ed.), no. 2587 in Lecture Notes in Computer Science, Springer-Verlag, 2002. [p. 338]
57. A. Bibliowicz, P. Cohen, and E. Biham, "A system for assisting analysis of some block ciphers." Public report, NESSIE, 2002. NES/DOC/TEC/WP2/007. [p. 74]
58. I. Biehl, B. Meyer, and V. Müller, "Differential fault attacks on elliptic curve cryptosystems." in *Proceedings of Crypto'00* (M. Bellare, ed.), vol. 1880 of *Lecture Notes in Computer Science*, pp. 131–146, Springer-Verlag, 2000. [p. 223, 305, 346, 634, 646]
59. E. Biham, "New types of cryptoanalytic attacks using related keys." in *Proceedings of Eurocrypt'93* (T. Helleseth, ed.), no. 765 in Lecture Notes in Computer Science, pp. 398–409, Springer-Verlag, 1993. [p. 67, 68]
60. E. Biham, "How to forge DES-encrypted messages in 2^{28} steps." Technical report, Comp. Sci. Dept., Technion, 1996. Available from http://www.cs.technion.ac.il/Reports/. [p. 102, 140]
61. E. Biham, "Cryptanalysis of Ladder-DES." in *Proceedings of Fast Software Encryption – FSE'97* (E. Biham, ed.), no. 1267 in Lecture Notes in Computer Science, pp. 134–138, Springer-Verlag, 1997. [p. 68]
62. E. Biham, "Observations on the relations between the bit-functions of many S-boxes." in *Proceedings of the Third NESSIE Workshop*, 2002. NES/DOC/TEC/WP5/027. [p. 71, 87, 93, 101, 102, 114, 118, 127, 136, 137, 141]
63. E. Biham, "Optimization of IDEA." in *Proceedings of the Third NESSIE Workshop*, 2002. NES/DOC/TEC/WP6/026. [p. 81, 383]
64. E. Biham and A. Biryukov, "An improved Davis' attack on DES." *Journal of Cryptology*, vol. 10, no. 3, pp. 195–205, 1997. [p. 68]
65. E. Biham, A. Biryukov, and A. Shamir, "Cryptanalysis of Skipjack reduced to 31 rounds using impossible differentials." in *Proceedings of Eurocrypt'99* (J. Stern, ed.), no. 1592 in Lecture Notes in Computer Science, pp. 12–23, Springer-Verlag, 1999. [p. 64, 94, 143]

66. E. Biham, A. Biryukov, and A. Shamir, "Miss in the middle attacks on IDEA, Khufu, and Khafre." in *Proceedings of Fast Software Encryption – FSE'99* (L. R. Knudsen, ed.), no. 1636 in Lecture Notes in Computer Science, pp. 124–138, Springer-Verlag, 1999. [p. 82–84, 138]

67. E. Biham, O. Dunkelman, V. Furman, and T. Mor, "Preliminary report on the NESSIE submissions: Anubis, Camellia, Khazad, IDEA, MISTY1, Nimbus, and Q." Public report, NESSIE, 2001. NES/DOC/TEC/WP3/011. [p. 87, 107, 143, 528]

68. E. Biham, O. Dunkelman, and N. Keller, "Linear cryptanalysis of reduced round Serpent." in *Proceedings of Fast Software Encryption – FSE'01* (M. Matsui, ed.), no. 2355 in Lecture Notes in Computer Science, pp. 16–27, Springer-Verlag, 2001. [p. 106, 528, 726]

69. E. Biham, O. Dunkelman, and N. Keller, "The rectangle attack – rectangling the Serpent." in *Proceedings of Eurocrypt'01* (B. Pfitzmann, ed.), no. 2045 in Lecture Notes in Computer Science, pp. 340–357, Springer-Verlag, 2001. [p. 106, 528, 722, 727]

70. E. Biham, O. Dunkelman, and N. Keller, "Enhancing differential-linear cryptanalysis." in *Proceedings of Asiacrypt'02* (Y. Zheng, ed.), no. 2501 in Lecture Notes in Computer Science, pp. 254–266, Springer-Verlag, 2002. NES/DOC/TEC/WP5/017. [p. 65, 140]

71. E. Biham, O. Dunkelman, and N. Keller, "New results on boomerang and rectangle attacks." in *Proceedings of Fast Software Encryption – FSE'02* (J. Daemen and V. Rijmen, eds.), no. 2365 in Lecture Notes in Computer Science, pp. 1–16, Springer-Verlag, 2002. NES/DOC/TEC/WP5/018. [p. 722, 727, 728]

72. E. Biham and V. Furman, "Differential cryptanalysis of Nimbus." Public report, NESSIE, 2001. NES/DOC/TEC/WP3/009. [p. 128, 147, 808]

73. E. Biham, V. Furman, M. Misztal, and V. Rijmen, "Differential cryptanalysis of Q." in *Proceedings of Fast Software Encryption – FSE'01* (M. Matsui, ed.), no. 2355 in Lecture Notes in Computer Science, pp. 174–186, Springer-Verlag, 2001. NES/DOC/TEC/WP3/012. [p. 134, 135, 148, 749, 750, 755, 757, 760]

74. E. Biham and N. Keller, "Cryptanalysis of reduced variants of Rijndael." in *Official public comment for Round 2 of the Advanced Encryption Standard development effort*, 2000. Available at http://csrc.nist.gov/encryption/aes/round2/conf3/papers/35-ebiham.pdf. [p. 87, 112, 114, 138, 144]

75. E. Biham, N. Keller, and O. Dunkelman, "Differential-linear cryptanalysis of Serpent." in *Proceedings of Fast Software Encryption – FSE'03* (T. Johansson, ed.), Lecture Notes in Computer Science, Springer-Verlag, 2003. NES/DOC/TEC/WP5/032. [p. 65]

76. E. Biham, N. Keller, and O. Dunkelman, "Rectangle attacks on SHACAL-1." in *Proceedings of Fast Software Encryption – FSE'03* (T. Johansson, ed.), Lecture Notes in Computer Science, Springer-Verlag, 2003. NES/DOC/TEC/WP5/031. [p. 124, 146, 189]

77. E. Biham and A. Shamir, "Differential cryptanalysis of DES-like cryptosystems." in *Proceedings of Crypto'90* (A. Menezes and S. A. Vanstone, eds.), no. 537 in Lecture Notes in Computer Science, pp. 2–21, Springer-Verlag, 1990. [p. 63, 715]

78. E. Biham and A. Shamir, *Differential cryptanalysis of the Data Encryption Standard*. Springer-Verlag, 1993. [p. 9, 175]

79. E. Biham and A. Shamir, "Differential fault analysis of secret key cryptosystems." in *Proceedings of Crypto'97* (B. S. Kaliski, Jr, ed.), no. 1294 in Lecture Notes in Computer Science, pp. 513–525, Springer-Verlag, 1997. [p. 13, 346]

80. E. Biham and A. Shamir, "Power analysis of the key scheduling of the AES candidates." in *Proceedings of the Second Advanced Encryption Standard Conference*, NIST, 1999. [p. 339]

81. A. Biryukov, "Analysis of involutional ciphers: Khazad and Anubis." in *Proceedings of Fast Software Encryption – FSE'03* (T. Johansson, ed.), Lecture Notes in Computer Science, Springer-Verlag, 2003. [p. 71, 88]

82. A. Biryukov and C. De Cannière, "Block ciphers and systems of quadratic equations." in *Proceedings of Fast Software Encryption – FSE'03* (T. Johansson, ed.), Lecture Notes in Computer Science, Springer-Verlag, 2003. Also in *Proceedings of the Third NESSIE Workshop*, 2002. [p. 70, 85, 89, 95, 107, 112, 115]

83. A. Biryukov, C. De Cannière, A. Braeken, and B. Preneel, "A toolbox for cryptanalysis: Linear and affine equivalence algorithms." in *Proceedings of Eurocrypt'03* (E. Biham, ed.), Lecture Notes in Computer Science, Springer-Verlag, 2003. NES/DOC/KUL/WP5/029. To appear. [p. 71, 87]

84. A. Biryukov, C. De Cannière, and G. Dellkrantz, "Cryptanalysis of SAFER++." in *Proceedings of Crypto'03* (D. Boneh, ed.), Lecture Notes in Computer Science, Springer-Verlag, 2003. NES/DOC/KUL/WP5/028. Full version available at http://eprint.iacr.org/2003/109/. [p. 119, 144]

85. A. Biryukov and E. Kushilevitz, "Improved cryptanalysis of RC5." in *Proceedings of Eurocrypt'98* (K. Nyberg, ed.), no. 1403 in Lecture Notes in Computer Science, pp. 85–99, Springer-Verlag, 1998. [p. 108]

86. A. Biryukov, J. Nakahara, Jr, B. Preneel, and J. Vandewalle, "New weak-key classes of IDEA." in *Proceedings of ICICS'02* (R. H. Deng, S. Qing, F. Bao, and J. Zhou, eds.), no. 2513 in Lecture Notes in Computer Science, pp. 315–326, Springer-Verlag, 2002. NES/DOC/KUL/WP5/019. Also in *Proceedings of the Third NESSIE Workshop*, 2002. [p. 82, 83, 85, 138]

87. A. Biryukov and A. Shamir, "Cryptanalytic time/memory/data tradeoffs for stream ciphers." in *Proceedings of Asiacrypt'00* (T. Okamoto, ed.), no. 1976 in Lecture Notes in Computer Science, pp. 1–13, Springer-Verlag, 2000. [p. 152, 708]

88. A. Biryukov and A. Shamir, "Structural cryptanalysis of SASAS." in *Proceedings of Eurocrypt'01* (B. Pfitzmann, ed.), no. 2045 in Lecture Notes in Computer Science, pp. 394–405, Springer-Verlag, 2001. [p. 68, 69]

89. A. Biryukov, A. Shamir, and D. Wagner, "Real time cryptanalysis of A5/1 on a PC." in *Proceedings of Fast Software Encryption – FSE'00* (B. Schneier, ed.), no. 1978 in Lecture Notes in Computer Science, pp. 1–18, Springer-Verlag, 2000. [p. 708]

90. A. Biryukov and D. Wagner, "Slide attacks." in *Proceedings of Fast Software Encryption – FSE'99* (L. R. Knudsen, ed.), no. 1636 in Lecture Notes in Computer Science, pp. 245–259, Springer-Verlag, 1999. [p. 68, 94, 502]

91. J. Black, S. Halevi, H. Krawczyk, T. Krovetz, and P. Rogaway, "UMAC: Fast and provably secure message authentication." in *Proceedings of Crypto'99* (M. J. Wiener, ed.), no. 1666 in Lecture Notes in Computer Science, pp. 216–233, Springer-Verlag, 1999. Full version available at http://www.cs.ucdavis.edu/~rogaway/umac/. [p. 385]

92. J. Black and P. Rogaway, "Ciphers with arbitrary finite domains." in *Proceedings of CT-RSA'02* (B. Preneel, ed.), no. 2271 in Lecture Notes in Computer Science, pp. 114–130, Springer-Verlag, 2002. Also available from http://www.cs.ucdavis.edu/~rogaway/papers/subset.htm. [p. 295]

93. J. Black, P. Rogaway, and T. Shrimpton, "Black-box analysis of the block-cipher-based hash-function constructions from PGV." in *Proceedings of Crypto'02* (M. Yung, ed.), no. 2442 in Lecture Notes in Computer Science, pp. 320–335, Springer-Verlag, 2002. Also available at http://www.cs.colorado.edu/~jrblack/papers.html. [p. 173, 183]

94. S. Blake-Wilson and A. J. Menezes, "Unknown key-share attacks on the station-to-station (sts) protocol." in *Proceedings of Public Key Cryptography – PKC'99* (H. Imai and Y. Zheng, eds.), no. 1560 in Lecture Notes in Computer Science, pp. 154–170, Springer-Verlag, 1999. [p. 263]

95. M. Blaze, W. Diffie, R. L. Rivest, B. Schneier, T. Shimomura, E. Thompson, and M. Wiener, "Minimal key lengths for symmetric ciphers to provide adequate commercial security." Jan. 1996. Available at http://www.counterpane.com/keylength.html. [p. 269]

96. D. Bleichenbacher, “Generating ElGamal signatures without knowing the secret key.” in *Proceedings of Eurocrypt’96* (U. Maurer, ed.), no. 1070 in Lecture Notes in Computer Science, pp. 10–18, Springer-Verlag, 1996. [p. 264, 304]

97. D. Bleichenbacher, “Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS#1.” in *Proceedings of Crypto’98* (H. Krawczyk, ed.), no. 1462 in Lecture Notes in Computer Science, pp. 1–12, Springer-Verlag, 1998. [p. 342]

98. J. Blömer and A. May, “Low secret exponent RSA revisited.” in *Cryptography and Lattices - Proceedings of CaLC’01* (J. H. Silverman, ed.), vol. 2146 of *Lecture Notes in Computer Science*, pp. 4–19, Springer-Verlag, 2001. [p. 336]

99. J. Blömer and J.-P. Seifert, “Fault based cryptanalysis of the Advanced Encryption Standard (AES).” in *Proceedings of Financial Cryptography – FC’03*, Lecture Notes in Computer Science, Springer-Verlag, 2003. Available at `http://eprint.iacr.org/2002/075/`. [p. 344, 346]

100. L. Blum, M. Blum, and M. Shub, “A simple unpredictable pseudorandom number generator.” *SIAM Journal on Computing*, vol. 15, no. 2, pp. 364–383, May 1986. [p. 157]

101. M. Blum and S. Micali, “How to generate cryptographically strong sequences of pseudo-random bits.” *SIAM Journal on Computing*, vol. 15, no. 2, pp. 850–864, May 1986. [p. 159, 763, 770]

102. M. Blunden and A. Escott, “Related key attacks on reduced round KASUMI.” in *Proceedings of Fast Software Encryption – FSE’01* (M. Matsui, ed.), Lecture Notes in Computer Science, pp. 277–285, Springer-Verlag, 2001. [p. 139]

103. D. Boneh, “Simplified OAEP for the RSA and Rabin functions.” in *Proceedings of Crypto’01* (J. Kilian, ed.), no. 2139 in Lecture Notes in Computer Science, pp. 275–291, Springer-Verlag, 2001. Also available at `http://crypto.stanford.edu/~dabo/papers/saep.ps`. [p. 244, 258, 473]

104. D. Boneh, R. A. DeMillo, and R. J. Lipton, “On the importance of checking cryptographic protocols for faults.” in *Proceedings of Eurocrypt’97* (W. Fumy, ed.), no. 1233 in Lecture Notes in Computer Science, pp. 37–51, Springer-Verlag, 1997. [p. 13, 312, 334, 336, 343, 345, 346]

105. D. Boneh, X. Ding, G. Tsudik, and B. Wong, “Instantaneous revocation of security capabilities.” in *Proceedings of USENIX Security Symposium 2001*, Aug. 2001. [p. 312]

106. D. Boneh and G. Durfee, “Cryptanalysis of RSA with private key d less than $n^{0.292}$.” *IEEE Transactions on Information Theory*, vol. IT-46, no. 4, pp. 1339–1349, July 2000. [p. 249, 257, 336, 640]

107. D. Boneh, G. Durfee, and Y. Frankel, “An attack of RSA given a small fraction of the private key bits.” in *Proceedings of Asiacrypt’98* (K. Ohta and D. Pei, eds.), no. 1514 in Lecture Notes in Computer Science, pp. 25–34, Springer-Verlag, 1998. Also available from `http://crypto.stanford.edu/~dabo/abstracts/bits_of_d.html`. [p. 312]

108. D. Boneh, G. Durfee, and N. Howgrave-Graham, “Factoring $n = p^r q$ for large r.” in *Proceedings of Crypto’99* (M. J. Wiener, ed.), no. 1666 in Lecture Notes in Computer Science, pp. 326–337, Springer-Verlag, 1999. [p. 221, 244, 268]

109. D. Boneh and M. Franklin, “Efficient generation of shared RSA keys.” in *Proceedings of Crypto’97* (B. S. Kaliski, Jr, ed.), no. 1294 in Lecture Notes in Computer Science, pp. 425–439, Springer-Verlag, 1997. [p. 697]

110. D. Boneh and R. Venkatesan, “Breaking RSA may not be equivalent to factoring.” in *Proceedings of Eurocrypt’98* (K. Nyberg, ed.), no. 1403 in Lecture Notes in Computer Science, pp. 59–71, Springer-Verlag, 1998. Full paper available from `http://crypto.stanford.edu/~dabo/abstracts/no_rsa_red.html`. [p. 266, 312]

111. N. Borisov, M. Chew, R. Johnson, and D. Wagner, “Multiplicative differentials.” in *Proceedings of Fast Software Encryption – FSE’02* (J. Daemen and V. Rijmen,

eds.), no. 2365 in Lecture Notes in Computer Science, pp. 17–33, Springer-Verlag, 2002. [p. 83, 129]

112. J. Borst, "The block cipher Grand Cru." Primitive submitted to NESSIE, Sept. 2000. [p. 37, 52, 131, 132, 357]

113. J. Borst, L. R. Knudsen, and V. Rijmen, "Two attacks on reduced IDEA (extended abstract)." in *Proceedings of Eurocrypt'97* (W. Fumy, ed.), no. 1233 in Lecture Notes in Computer Science, pp. 1–13, Springer-Verlag, 1997. [p. 82, 84]

114. J. Borst, B. Preneel, and J. Vandewalle, "Linear cryptanalysis of RC5 and RC6." in *Proceedings of Fast Software Encryption – FSE'99* (L. R. Knudsen, ed.), no. 1636 in Lecture Notes in Computer Science, pp. 16–30, Springer-Verlag, 1999. [p. 111]

115. W. Bosma and M.-P. van der Hulst, "Faster primality testing." in *Proceedings of Eurocrypt'89* (J.-J. Quisquater and J. Vandewalle, eds.), no. 435 in Lecture Notes in Computer Science, pp. 652–656, Springer-Verlag, 1990. [p. 697]

116. A. Bosselaers and B. Preneel, eds., *Integrity Primitives for Secure Information Systems, Final Report of RACE Integrity Primitives Evaluation RIPE-RACE 1040*. No. 1007 in Lecture Notes in Computer Science, Springer-Verlag, 1995. [p. 5, 9, 605]

117. F. Boudot, "Efficient proofs that a committed number lies in an interval." in *Proceedings of Eurocrypt'00* (B. Preneel, ed.), no. 1807 in Lecture Notes in Computer Science, pp. 431–444, Springer-Verlag, 2000. [p. 788]

118. F. Boudot and J. Traoré, "Efficient publicly verifiable secret sharing schemes with fast or delayed recovery." in *Proceedings of ICICS'99* (V. Varadharajan and Y. Mu, eds.), no. 1726 in Lecture Notes in Computer Science, pp. 87–102, Springer-Verlag, 1999. [p. 788]

119. J. Brandt, I. Damgård, P. Landrock, and T. P. Pedersen, "Zero-knowledge authentication scheme with secret key exchange." in *Proceedings of Crypto'88* (S. Goldwasser, ed.), no. 403 in Lecture Notes in Computer Science, pp. 583–588, Springer-Verlag, 1988. [p. 800]

120. J. Brandt, I. Damgård, P. Landrock, and T. P. Pedersen, "Zero-knowledge authentication scheme with secret key exchange." *Journal of Cryptology*, vol. 11, no. 3, pp. 147–159, 1998. Journal version of [119]. [p. 329]

121. J. Brandt and I. B. Damgård, "On generation of probable primes by incremental search." in *Proceedings of Crypto'92* (E. F. Brickell, ed.), no. 740 in Lecture Notes in Computer Science, pp. 358–370, Springer-Verlag, 1993. [p. 695]

122. J. Brandt, I. B. Damgård, and P. Landrock, "Speeding up prime number generation." in *Proceedings of Asiacrypt'91* (H. Imai, R. L. Rivest, and T. Matsumoto, eds.), no. 739 in Lecture Notes in Computer Science, pp. 440–449, Springer-Verlag, 1993. [p. 695, 697]

123. E. Brickell, D. Pointcheval, S. Vaudenay, and M. Yung, "Design validations for discrete logarithm based signature schemes." in *Proceedings of Public Key Cryptography – PKC'00* (H. Imai and Y. Zheng, eds.), no. 1751 in Lecture Notes in Computer Science, pp. 276–292, Springer-Verlag, 2000. Also available at `http://www.di.ens.fr/~pointche/pub.php?reference=BrPoVaYu00`. [p. 262, 285, 287, 289, 291, 292, 295]

124. D. R. L. Brown, "The exact security of ECDSA." Available at `http://grouper.ieee.org/groups/1363/Research/contributions/ECDSASec.ps`, improved in [126], 2000. [p. 272]

125. D. R. L. Brown, "A security analysis of the elliptic curve digital signature algorithm." in *Proc. 5th Workshop on Elliptic Curve Cryptography (ECC'01)*, 2001. Abstract available at `http://www.cacr.math.uwaterloo.ca/conferences/2001/ecc/abstracts.html`. [p. 294]

126. D. R. L. Brown, "Generic groups, collision resistance, and ECDSA." Available at `http://eprint.iacr.org/2002/026/`, 2002. [p. 173, 272, 293, 295, 304, 800]

127. D. R. L. Brown, "How much "provable security" does ECDSA have?." 12 Dec. 2002. NESSIE discussion forum. [p. 284, 312]

128. B. Buchberger, "Gröbner bases : an algorithmic method in polynomial ideal theory." in *Multidimensional systems theory* (N.-K. Bose, ed.), no. 16 in Mathematics and its Applications, ch. 6, pp. 184–232, D. Reidel Pub. Co., 1985.
Based on his PhD thesis: *Ein Algorithmus zum Auffinden der Basiselemente des Restklassenringes nach einem nulldimensionalen Polynomideal*, U. Innsbruck, Austria, 1965. [p. 308, 670]
129. J. Buchmann and S. Hamdy, "A survey on IQ cryptography." in *Public-Key Cryptography and Computational Number Theory* (W. de Gruyter, ed.), pp. 1–15, 2001. [p. 781]
130. M. Burmester, "An almost-constant round interactive zero-knowledge proof." *Information Processing Letters*, vol. 42, no. 2, pp. 81–87, 1992. [p. 336]
131. J. Camenisch and A. Lysyanskaya, "A signature scheme with efficient protocols." in *Proceedings of SCN'02* (S. Cimato, C. Galdi, and G. Persiano, eds.), vol. 2576 of *Lecture Notes in Computer Science*, Springer-Verlag, 2002. Also available at `http://theory.lcs.mit.edu/~cis/pubs/lysyanskaya/cl02b.ps.gz`. [p. 300]
132. J. Camenisch and M. Michels, "A group signature scheme with improved efficiency." in *Proceedings of Asiacrypt'98* (K. Ohta and D. Pei, eds.), no. 1514 in Lecture Notes in Computer Science, pp. 160–174, Springer-Verlag, 1998. [p. 788]
133. J. Camenisch and M. Michels, "Proving in zero-knowledge that a number is the product of two safe primes." in *Proceedings of Eurocrypt'99* (J. Stern, ed.), no. 1592 in Lecture Notes in Computer Science, pp. 107–122, Springer-Verlag, 1999. [p. 788]
134. R. Canetti, O. Goldreich, S. Goldwasser, and S. Micali, "Resettable Zero-Knowledge." *Electronic Colloquium on Computational Complexity (ECCC)*, vol. 42, 1999. Also available from `ftp://ftp.eccc.uni-trier.de/pub/eccc/reports/1999/TR99-042/index.html`. [p. 326]
135. R. Canetti, O. Goldreich, and S. Halevi, "The random oracle methodology, revisited." in *Proceedings of Symposium on Theory of Computing – STOC'98*, pp. 209–218, ACM Press, 1998. Also available at `http://theory.lcs.mit.edu/~oded/rom.html`. [p. 222, 272]
136. Certicom Research, "Standards for Efficient Cryptography - SEC 1: Elliptic curve cryptography." 2000. Available at `http://www.secg.org/`. [p. 224, 237, 269, 666, 667]
137. Certicom Research, "Standards for Efficient Cryptography - SEC 2: Recommended elliptic curve domain parameters." 2000. Available at `http://www.secg.org/`. [p. 303, 666]
138. F. Chabaud and A. Joux, "Differential collisions in SHA-0." in *Proceedings of Crypto'98* (H. Krawczyk, ed.), no. 1462 in Lecture Notes in Computer Science, pp. 56–71, Springer-Verlag, 1998. [p. 122, 187, 188]
139. A. H. Chan, Y. Frankel, and Y. Tsiounis, "Easy come - easy go divisible cash." in *Proceedings of Eurocrypt'98* (K. Nyberg, ed.), no. 1403 in Lecture Notes in Computer Science, pp. 561–575, Springer-Verlag, 1998. Available as a GTE Tech report. [p. 788]
140. J. H. Cheon, M. Kim, K. Kim, J.-Y. Lee, and S. Kang, "Improved impossible differential cryptanalysis of Rijndael and Crypton." in *Proceedings of ICISC'01* (K. Kim, ed.), no. 2288 in Lecture Notes in Computer Science, pp. 39–49, Springer-Verlag, 2001. [p. 114]
141. M. Ciet, G. Martinet, and F. Sica, "ACE: Advanced cryptographic engine." Public report, NESSIE, 2001. `NES/DOC/ENS/WP3/008`. [p. 235]
142. C. Clavier, J.-S. Coron, and N. Dabbous, "Differential power analysis in the presence of hardware countermeasures." in *Proceedings of CHES'00* (Çetin Kaya Koç and C. Paar, eds.), no. 1965 in Lecture Notes in Computer Science, pp. 252–263, Springer-Verlag, 2000. [p. 340]

143. S. Contini, R. L. Rivest, M. J. B. Robshaw, and Y. L. Yin, “The security of the RC6 block cipher.” Technical report, RSA Laboratories, Aug. 1998. Available at `http://www.rsa.com/rsalabs/aes/`. [p. 47, 65, 109–111, 143]

144. D. Coppersmith, “Fast evaluation of logarithms in fields of characteristic two.” *IEEE Transactions on Information Theory*, vol. IT-30, no. 4, pp. 587–594, 1984. [p. 221, 268]

145. D. Coppersmith, “Impact of Courtois and Pieprzyk results.” NIST AES Discussion Forum, General Cryptanalysis Issues, Sept. 2002. Available from `http://aes.nist.gov/aes/`. [p. 115]

146. D. Coppersmith, “Personal communication to the authors of [452].” Apr. 2002. [p. 112, 115]

147. D. Coppersmith, S. Halevi, and C. S. Jutla, “Cryptanalysis of stream ciphers with linear masking.” in *Proceedings of Crypto’02* (M. Yung, ed.), no. 2442 in Lecture Notes in Computer Science, pp. 515–532, Springer-Verlag, 2002. Also available at `http://eprint.iacr.org/2002/020/`. [p. 161]

148. D. Coppersmith and S. Winograd, “Matrix multiplication via arithmetic progressions.” in *Proceedings of Symposium on Theory of Computing – STOC’87*, ACM Press, 1987. [p. 709]

149. J.-S. Coron, “On the exact security of Full Domain Hash.” in *Proceedings of Crypto’00* (M. Bellare, ed.), no. 1880 in Lecture Notes in Computer Science, pp. 229–235, Springer-Verlag, 2000. [p. 272, 276]

150. J.-S. Coron, “Resistance against differential power analysis for elliptic curve cryptosystems.” in *Proceedings of CHES’99* (Çetin Kaya Koç and C. Paar, eds.), no. 1717 in Lecture Notes in Computer Science, pp. 292–302, Springer-Verlag, 2000. Also available at `http://www.gemplus.com/smart/r_d/publi_crypto/download/515698_1290115520.pdf`. [p. 342]

151. J.-S. Coron, “Optimal security proofs for PSS and other signature schemes.” in *Proceedings of Eurocrypt’02* (L. R. Knudsen, ed.), no. 2332 in Lecture Notes in Computer Science, pp. 272–287, Springer-Verlag, 2002. Also available at `http://eprint.iacr.org/2001/062/`. [p. 277, 279, 280, 657]

152. J.-S. Coron, H. Handschuh, and D. Naccache, “ECC: Do we need to count?.” in *Proceedings of Asiacrypt’99* (K.-Y. Lam, E. Okamoto, and C. Xing, eds.), no. 1716 in Lecture Notes in Computer Science, pp. 122–134, Springer-Verlag, 1999. [p. 782]

153. J.-S. Coron and L. Goubin, “On boolean and arithmetic masking against differential power analysis.” in *Proceedings of CHES’00* (Çetin Kaya Koç and C. Paar, eds.), no. 1965 in Lecture Notes in Computer Science, pp. 231–237, Springer-Verlag, 2000. [p. 340]

154. N. T. Courtois, “Higher order correlation attacks, XL algorithm, and cryptanalysis of Toyocrypt.” in *Proceedings of ICISC’02* (K. Kim, ed.), no. 2587 in Lecture Notes in Computer Science, Springer-Verlag, 2002. Also available at `http://eprint.iacr.org/2002/087/`. [p. 153]

155. N. T. Courtois, “Generic attacks and the security of Quartz.” in *Proceedings of Public Key Cryptography – PKC’03* (Y. Desmedt, ed.), no. 2567 in Lecture Notes in Computer Science, pp. 351–364, Springer-Verlag, 2003. Also available at `http://www.minrank.org/quartzbounds.pdf`. An earlier version appeared in *Proceedings of the Second NESSIE Workshop*, 2001. [p. 281, 282]

156. N. T. Courtois, “Algebraic attacks over $GF(2^k)$, application to HFE challenge 2 and Sflash-v2.” in *Proceedings of Public Key Cryptography – PKC’04*, Lecture Notes in Computer Science, Springer-Verlag, 2004. [p. 308, 669]

157. N. T. Courtois, M. Daum, and P. Felke, “On the security of HFE, HFEv- and Quartz.” in *Proceedings of Public Key Cryptography – PKC’03* (Y. Desmedt, ed.), no. 2567 in Lecture Notes in Computer Science, pp. 337–350, Springer-Verlag, 2003. Also available at `http://eprint.iacr.org/2002/138/`, and also in *Proceedings of the Third NESSIE Workshop*, 2002. [p. 268, 310]

158. N. T. Courtois, L. Goubin, W. Meier, and J.-D. Tacier, "Solving underfined systems of multivariate quadratic equations." in *Proceedings of Public Key Cryptography – PKC'02* (D. Naccache and P. Paillier, eds.), no. 2274 in Lecture Notes in Computer Science, pp. 211–227, Springer-Verlag, 2002. [p. 112, 115]

159. N. T. Courtois, L. Goubin, and J. Patarin, "Quartz, an asymmetric signature scheme for short signatures on PC (first version)." Primitive submitted to NESSIE, Sept. 2000. See also `http://www.minrank.org/quartz/` or [160]. [p. 310]

160. N. T. Courtois, L. Goubin, and J. Patarin, "Quartz, 128-bit long digital signature." in *Proceedings of CT-RSA'01* (D. Naccache, ed.), no. 2020 in Lecture Notes in Computer Science, pp. 282–297, Springer-Verlag, 2001. [p. 803]

161. N. T. Courtois, L. Goubin, and J. Patarin, "Quartz, an asymmetric signature scheme for short signatures on PC (second revised version)." Primitive submitted to NESSIE, Sept. 2001. See also `http://www.minrank.org/quartz/`. [p. 39, 53, 277, 281, 310, 358]

162. N. T. Courtois, L. Goubin, and J. Patarin, "SFLASH: a fast asymmetric signature scheme - Statement issued by the authors." Technical report, NESSIE external documents, Oct. 2003. Available from `http://www.cryptonessie.org/tweaks.html`. [p. 669]

163. N. T. Courtois, L. Goubin, and J. Patarin, "SFLASHv3: a fast asymmetric signature scheme." Available at `http://eprint.iacr.org/2003/211/`, 2003. [p. 308, 669]

164. N. T. Courtois, A. Klimov, J. Patarin, and A. Shamir, "Efficient algorithms for solving overdefined systems of multivariate polynomial equations." in *Proceedings of Eurocrypt'00* (B. Preneel, ed.), no. 1807 in Lecture Notes in Computer Science, pp. 392–407, Springer-Verlag, 2000. [p. 70, 112, 115, 153, 308, 670]

165. N. T. Courtois and J. Pieprzyk, "Cryptanalysis of block ciphers with overdefined systems of equations." in *Proceedings of Asiacrypt'02* (Y. Zheng, ed.), no. 2501 in Lecture Notes in Computer Science, pp. 267–287, Springer-Verlag, 2002. Different version of the preprint [166]. [p. 70, 77, 85, 89, 107, 112, 115, 185, 528]

166. N. T. Courtois and J. Pieprzyk, "Cryptanalysis of block ciphers with overdefined systems of equations." Available at `http://eprint.iacr.org/2002/044/`, 2002. [p. 803]

167. C. Couvreur and J.-J. Quisquater, "Fast decipherment algorithm for RSA public-key cryptosystem." *Electronics Letters*, vol. 18, pp. 905–907, 1982. [p. 696, 697]

168. C. Couvreur and J.-J. Quisquater, "An introduction to fast generation of large prime numbers." *Philips Journal of Research*, vol. 37, pp. 231–264, 1982. [p. 695, 697]

169. R. Cramer and I. B. Damgård, "New generation of secure and practical RSA-based signatures." in *Proceedings of Crypto'96* (N. Koblitz, ed.), no. 1109 in Lecture Notes in Computer Science, pp. 173–185, Springer-Verlag, 1996. [p. 260]

170. R. Cramer and V. Shoup, "Signature schemes based on the strong RSA assumption." in *Proceedings of Conference on Computer and Communications Security – CCS'99*, pp. 46–52, ACM Press, 1999. [p. 300]

171. R. Cramer and V. Shoup, "Design and analysis of practical public-key encryption schemes secure against adaptive chosen ciphertext attack." 2001. Available at `http://www.shoup.net/papers/cca2.ps`. [p. 225, 231–233, 235, 639, 646]

172. P. Crowley and S. Lucks, "Bias in the LEVIATHAN stream cipher." in *Proceedings of Fast Software Encryption – FSE'01* (M. Matsui, ed.), no. 2355 in Lecture Notes in Computer Science, pp. 211–218, Springer-Verlag, 2001. [p. 168]

173. Cryptrec, "Cryptrec liaison report to ISO/IEC 18033-2 and 18033-3." Technical report, Cryptography Research and Evaluation Committees, Oct. 2002. [p. 385]

174. J. Daemen, *Cipher and Hash Function Design Strategies Based on Linear and Differential Cryptanalysis*. Doctoral dissertation, K. U. Leuven, Mar. 1995. [p. 85]

175. J. Daemen, R. Govaerts, and J. Vandewalle, "Cryptanalysis of 2.5 rounds of IDEA (extended abstract)." Technical report 93/1, Department of Electrical Engineering, ESAT–COSIC, Mar. 1993. [p. 84]

176. J. Daemen, R. Govaerts, and J. Vandewalle, "Weak keys for IDEA." in *Proceedings of Crypto'93* (D. R. Stinson, ed.), no. 773 in Lecture Notes in Computer Science, pp. 224–231, Springer-Verlag, 1994. [p. 82, 85]

177. J. Daemen, R. Govaerts, and J. Vandewalle, "Correlation matrices." in *Proceedings of Fast Software Encryption – FSE'94* (B. Preneel, ed.), no. 1008 in Lecture Notes in Computer Science, pp. 275–285, Springer-Verlag, 1995. [p. 756]

178. J. Daemen, L. R. Knudsen, and V. Rijmen, "The block cipher Square." in *Proceedings of Fast Software Encryption – FSE'97* (E. Biham, ed.), no. 1267 in Lecture Notes in Computer Science, pp. 149–165, Springer-Verlag, 1997. [p. 68, 69, 111, 112]

179. J. Daemen, L. R. Knudsen, and V. Rijmen, "Linear frameworks for block ciphers." *Designs, Codes, and Cryptography*, vol. 22, no. 1, pp. 65–87, 2001. [p. 68, 69]

180. J. Daemen, M. Peeters, G. van Assche, and V. Rijmen, "Noekeon." Primitive submitted to NESSIE, Sept. 2000. [p. 37, 52, 133, 134, 357]

181. J. Daemen and V. Rijmen, "AES proposal: Rijndael." Selected as the Advanced Encryption Standard. Available from `http://csrc.nist.gov/encryption/aes/`, Sept. 1999. [p. 15, 16, 22, 69, 111–113, 131, 144, 521, 763]

182. J. Daemen and V. Rijmen, "The wide trail design strategy." in *Proceedings of Cryptography and Coding – CC'01* (B. Honary, ed.), no. 2260 in Lecture Notes in Computer Science, pp. 222–238, Springer-Verlag, 2001. [p. 111]

183. J. Daemen and V. Rijmen, *The design of Rijndael: AES — The Advanced Encryption Standard*. Springer-Verlag, 2002. [p. 519, 520]

184. I. B. Damgård and M. Koprowski, "Generic lower bounds for root extraction and signature schemes in general groups." in *Proceedings of Eurocrypt'02* (L. R. Knudsen, ed.), no. 2332 in Lecture Notes in Computer Science, pp. 256–271, Springer-Verlag, 2002. Also available at `http://eprint.iacr.org/2002/013/`. [p. 300]

185. D. W. Davies and S. Murphy, "Pairs and triplets of DES S-Boxes." *Journal of Cryptology*, vol. 8, pp. 1–25, 1995. [p. 68]

186. E. Dawson, J. D. Golic, W. Millan, and L. Simpson, "Response to initial report on LILI-128." in *Proceedings of the Second NESSIE Workshop*, 2001. [p. 710]

187. E. Dawson, W. Millan, L. Penna, L. Simpson, and J. D. Golic, "LILI-128." Primitive submitted to NESSIE, Sept. 2000. [p. 38, 53, 358, 707, 709, 712]

188. C. De Cannière, "Guess and determine attack on SOBER." Public report, NESSIE, 2001. `NES/DOC/KUL/WP5/010`. [p. 166]

189. C. De Cannière, J. Lano, B. Preneel, and J. Vandewalle, "Distinguishing attacks on Sober-t32." in *Proceedings of the Third NESSIE Workshop*, 2002. [p. 165]

190. A. De Santis, Y. Desmedt, Y. Frankel, and M. Yung, "How to share a function securely." in *Proceedings of Symposium on Theory of Computing – STOC'94*, pp. 522–533, ACM Press, 1994. [p. 312]

191. H. Demirci, "Square-like attacks on reduced rounds of IDEA." in *Proceedings of Selected Areas in Cryptography – SAC'02* (K. Nyberg and H. M. Heys, eds.), no. 2595 in Lecture Notes in Computer Science, Springer-Verlag, 2002. [p. 84, 138]

192. A. W. Dent, "An evaluation of EPOC-2." Public report, NESSIE, 2001. `NES/DOC/RHU/WP5/017`. [p. 243, 343]

193. A. W. Dent, "ACE-KEM and the general KEM-DEM structure." Public report, NESSIE, 2002. `NES/DOC/RHU/WP5/023`. [p. 228, 229, 235, 248, 639]

194. A. W. Dent, "Adapting the weaknesses of the random oracle model to the generic group model." in *Proceedings of Asiacrypt'02* (Y. Zheng, ed.), no. 2501 in Lecture Notes in Computer Science, pp. 100–109, Springer-Verlag, 2002. `NES/DOC/RHU/WP5/021`. Also available at `http://eprint.iacr.org/2002/086/`. [p. 223, 272]

195. A. W. Dent, "A designer's guide to KEMs." Public report, NESSIE, 2002. NES/DOC/RHU/WP5/029. Also available at http://eprint.iacr.org/2002/174/. [p. 229, 240, 248, 634, 639]

196. A. W. Dent, "An implementation attack against the EPOC-2 public-key cryptosystem." *Electronics Letters*, vol. 38, no. 9, p. 412, 2002. [p. 244]

197. A. W. Dent and E. Dottax, "An overview of side-channel attacks on the NESSIE asymmetric encryption primitives." Public report, NESSIE, 2002. NES/DOC/RHU/WP5/020. [p. 224, 237, 241, 244, 247, 249, 345]

198. J.-F. Dhem, F. Koeune, P.-A. Leroux, P. Mestré, J.-J. Quisquater, and J.-L. Willems, "A practical implementation of the timing attack." in *Proceedings of CARDIS'98* (J.-J. Quisquater and B. Schneier, eds.), no. 1820 in Lecture Notes in Computer Science, pp. 167–182, Springer-Verlag, 2000. Also available at http://www.dice.ucl.ac.be/crypto/techreports.html. [p. 338]

199. M. Dichtl and M. Schafheutle, "Linearity properties of SOBER-t32 key loading." in *Proceedings of Fast Software Encryption – FSE'02* (J. Daemen and V. Rijmen, eds.), no. 2365 in Lecture Notes in Computer Science, pp. 225–230, Springer-Verlag, 2002. [p. 156, 166]

200. M. Dichtl and P. Serf, "About the NESSIE submission "Using the general next bit predictor like an evaluation criteria"." in *Proceedings of the Second NESSIE Workshop*, 2001. NES/DOC/SAG/WP3/019. [p. 53]

201. W. Diffie and M. E. Hellman, "New directions in cryptography." *IEEE Transactions on Information Theory*, vol. IT-22, pp. 644–654, 1976. [p. 213, 273, 633, 645]

202. C. Ding, D. Pei, and A. Salomaa, "Chinese remainder theorem." Word Scientific, 1996. [p. 696, 700]

203. distributed.net, "RC5-64 has been solved.". See http://distributed.net/pressroom/news-20020926.html. [p. 270]

204. H. Dobbertin, "RIPEMD with two-round compress function is not collision-free." *Journal of Cryptology*, vol. 10, no. 1, pp. 51–69, 1997. [p. 175]

205. H. Dobbertin, "Cryptanalysis of MD4." *Journal of Cryptology*, vol. 11, no. 4, pp. 253–271, 1998. [p. 175]

206. Y. Dodis and L. Reyzin, "On the power of claw-free permutations." in *Proceedings of SCN'02* (S. Cimato, C. Galdi, and G. Persiano, eds.), vol. 2576 of *Lecture Notes in Computer Science*, Springer-Verlag, 2002. Also available at http://eprint.iacr.org/2002/103/. [p. 276, 278, 279]

207. E. Dottax, "Fault and chosen modulus attacks on some NESSIE asymmetric primitives." Public report, NESSIE, 2002. NES/DOC/ENS/WP5/035. [p. 224, 249, 334, 346]

208. E. Dottax, "Fault attacks on NESSIE signature and identification schemes." Public report, NESSIE, 2002. NES/DOC/ENS/WP5/031. [p. 334, 345]

209. E. Dottax, "Three asymmetric encryption schemes based upon the factoring assumption: EPOC-2, HIME(R) and Rabin-SAEP." Public report, NESSIE, 2002. NES/DOC/ENS/WP5/028. [p. 244]

210. O. Dunkelman, "Comparing MISTY1 and KASUMI." Public report, NESSIE, Sept. 2002. NES/DOC/TEC/WP5/029. [p. 93]

211. O. Dunkelman and N. Keller, "Boomerang and rectangle attacks on SC2000." in *Proceedings of the Second NESSIE Workshop*, 2001. NES/DOC/TEC/WP3/014. [p. 136, 148, 722, 727]

212. C. Dwork and M. Naor, "An efficient existentially unforgeable signature scheme and its applications." in *Proceedings of Crypto'94* (Y. Desmedt, ed.), no. 839 in Lecture Notes in Computer Science, pp. 218–238, Springer-Verlag, 1994. [p. 260]

213. P. Ebinger and E. Teske, "Factoring $n = pq^2$ with the elliptic curve method." in *Proceedings of Algorithmic Number Theory Seminar – ANTS-V* (C. Fieker and D. R. Kohel, eds.), vol. 2369 of *Lecture Notes in Computer Science*, pp. 475–490, Springer-Verlag, 2002. [p. 221, 268]

214. P. Ekdahl and T. Johansson, “SNOW.” Primitive submitted to NESSIE, Sept. 2000. [p. 38, 53, 160, 358]

215. P. Ekdahl and T. Johansson, “Distinguishing attacks on SOBER-t16 and t32.” in *Proceedings of Fast Software Encryption – FSE’02* (J. Daemen and V. Rijmen, eds.), no. 2365 in Lecture Notes in Computer Science, pp. 210–224, Springer-Verlag, 2002. [p. 163, 165]

216. E. El Mahassni, P. Q. Nguyen, and I. Shparlinsky, “The insecurity of Nyberg-Rueppel and other DSA-like signature schemes with partially known nonces.” in *Cryptography and Lattices - Proceedings of CaLC’01* (J. H. Silverman, ed.), vol. 2146 of *Lecture Notes in Computer Science*, pp. 97–109, Springer-Verlag, 2001. [p. 304]

217. E. El Mahassni and I. Shparlinski, “On some uniformity of distribution properties of ESIGN.” in *Proc. Workshop on Coding and Cryptography*, pp. 189–196, INRIA, 2001. Also available at `http://www.cs.mq.edu.au/~igor/ESIGN.ps`. [p. 306]

218. A. J. Elbirt, W. Yip, B. Chetwynd, and C. Paar, “An FPGA implementation and performance evaluation of the AES block cipher candidate algorithm finalists.” in *Proceedings of the Third Advanced Encryption Standard Conference*, pp. 13–27, NIST, Apr. 2000. [p. 747, 748]

219. T. ElGamal, “A public key cryptosystem and a signature scheme based on discrete logarithms.” *IEEE Transactions on Information Theory*, vol. IT-31, no. 4, pp. 469–472, July 1985. [p. 273, 289, 806]

220. T. ElGamal, “A public-key cryptosystem and signature scheme based on discrete logarithms.” in *Proceedings of Crypto’84* (G. R. Blakley and D. Chaum, eds.), no. 196 in Lecture Notes in Computer Science, pp. 10–18, Springer-Verlag, 1985. Full version in [219]. [p. 227]

221. A. Enge and P. Gaudry, “A general framework for subexponential discrete logarithm algorithms.” *Acta Arithmetica*, vol. 102, no. 1, pp. 83–103, 2002. Also available at `http://www.lix.polytechnique.fr/Labo/Andreas.Enge/vorabdrucke/subexp.ps.gz`. [p. 221, 268]

222. S. Even, O. Goldreich, and S. Micali, “On-line/off-line digital schemes.” in *Proceedings of Crypto’89* (G. Brassard, ed.), no. 435 in Lecture Notes in Computer Science, pp. 263–275, Springer-Verlag, 1989. [p. 777, 806]

223. S. Even, O. Goldreich, and S. Micali, “On-line/off-line digital signatures.” *Journal of Cryptology*, vol. 9, no. 1, pp. 35–67, 1996. Also available at `http://www.wisdom.weizmann.ac.il/~oded/PS/egm.ps`. Preliminary version in [222]. [p. 297]

224. J.-C. Faugère, “A new efficient algorithm for computing Gröbner bases without reduction to zero (F_5).” in *Proceedings of International Symposium on Symbolic and Algebraic Computation – ISSAC’02* (T. Mora, ed.), pp. 75–83, ACM Press, 2002. Also available at `http://www-calfor.lip6.fr/~jcf/Papers/@papers/f5/pdf`. [p. 308, 670]

225. J.-C. Faugère, “Report on a successful attack of HFE challenge 1 with Gröbner bases algorithm F5/2.” Announcement that appeared in `sci.crypt`, 19 Apr. 2002. [p. 268, 310]

226. H. Feistel, “Cryptography and computer privacy.” *Scientific American*, vol. 228, pp. 15–23, May 1973. [p. 750]

227. N. Ferguson, J. Kelsey, S. Lucks, B. Schneier, M. Stay, D. Wagner, and D. Whiting, “Improved cryptanalysis of Rijndael.” in *Proceedings of Fast Software Encryption – FSE’00* (B. Schneier, ed.), no. 1978 in Lecture Notes in Computer Science, pp. 213–230, Springer-Verlag, 2000. [p. 112–114, 144]

228. N. Ferguson, R. Schroeppel, and D. Whiting, “A simple algebraic representation of Rijndael.” in *Proceedings of Selected Areas in Cryptography – SAC’01* (S. Vaudenay and A. M. Youssef, eds.), no. 2259 in Lecture Notes in Computer Science, pp. 103–111, Springer-Verlag, 2001. [p. 112, 114]

229. A. Fiat and A. Shamir, “How to prove yourself: Practical solutions to identification and signature problems.” in *Proceedings of Crypto’86* (A. M. Odlyzko,

ed.), no. 263 in Lecture Notes in Computer Science, pp. 186–194, Springer-Verlag, 1987. [p. 272, 329, 780]

230. M. Fischlin, "A note on security proofs in the generic model." in *Proceedings of Asiacrypt'00* (T. Okamoto, ed.), no. 1976 in Lecture Notes in Computer Science, pp. 458–469, Springer-Verlag, 2000. [p. 223, 272]

231. M. Fischlin, "The Cramer-Shoup Strong-RSA signature scheme revisited." in *Proceedings of Public Key Cryptography – PKC'03* (Y. Desmedt, ed.), no. 2567 in Lecture Notes in Computer Science, pp. 116–129, Springer-Verlag, 2003. Also available at `http://eprint.iacr.org/2002/017/`. [p. 301]

232. S. R. Fluhrer, I. Mantin, and A. Shamir, "Weaknesses in the key scheduling algorithm of RC4." in *Proceedings of Selected Areas in Cryptography – SAC'01* (S. Vaudenay and A. M. Youssef, eds.), no. 2259 in Lecture Notes in Computer Science, pp. 1–24, Springer-Verlag, 2001. [p. 170]

233. S. R. Fluhrer and D. A. McGrew, "Statistical analysis of the alleged RC4 stream cipher." in *Proceedings of Fast Software Encryption – FSE'00* (B. Schneier, ed.), no. 1978 in Lecture Notes in Computer Science, pp. 19–30, Springer-Verlag, 2000. [p. 170]

234. P.-A. Fouque, J. Stern, and S. Vaudenay, "CS-cipher." Primitive submitted to NESSIE by CS Communication & Systèmes, Sept. 2000. [p. 37, 52, 125, 126, 147, 357]

235. G. Frey, M. Müller, and H.-G. Rück, "The Tate pairing and the discrete logarithm applied to elliptic curve cryptosystems." *IEEE Transactions on Information Theory*, vol. IT-45, no. 5, pp. 1717–1719, 1999. [p. 267]

236. G. Frey and H.-G. Rück, "A remark concerning m-divisibility and the discrete logarithm in the divisor class group of curves." *Mathematics of Computation*, vol. 62, no. 206, pp. 865–874, Apr. 1994. [p. 221, 268]

237. E. Fuijsaki, "Chosen ciphertext security of EPOC-2." Technical report, NTT Corporation, 2001. [p. 243]

238. E. Fujisaki, T. Kobayashi, H. Morita, H. Oguro, T. Okamoto, S. Okazaki, and D. Pointcheval, "PSEC: Provably secure elliptic curve encryption scheme." Primitive submitted to NESSIE by NTT Corp., Sept. 2000. See also `http://info.isl.ntt.co.jp/psec/`. [p. 39, 53, 253–255, 358, 389, 391, 392, 633]

239. E. Fujisaki, T. Kobayashi, H. Morita, H. Oguro, T. Okamoto, S. Okazaki, D. Pointcheval, and S. Uchiyama, "EPOC: Efficient probabilistic public-key encryption." Primitive submitted to NESSIE by NTT Corp., Sept. 2000. See also `http://info.isl.ntt.co.jp/epoc/`. [p. 39, 53, 249–252, 358]

240. E. Fujisaki and T. Okamoto, "A practical and provably secure scheme for publicly verifiable secret sharing and its applications." in *Proceedings of Eurocrypt'98* (K. Nyberg, ed.), no. 1403 in Lecture Notes in Computer Science, pp. 32–46, Springer-Verlag, 1998. [p. 788]

241. E. Fujisaki and T. Okamoto, "How to enhance the security of public-key encryption at minimum cost." in *Proceedings of Public Key Cryptography – PKC'99* (H. Imai and Y. Zheng, eds.), no. 1560 in Lecture Notes in Computer Science, pp. 53–68, Springer-Verlag, 1999. [p. 249, 254, 255]

242. E. Fujisaki and T. Okamoto, "Secure integration of asymmetric and symmetric encryption schemes." in *Proceedings of Crypto'99* (M. J. Wiener, ed.), no. 1666 in Lecture Notes in Computer Science, pp. 535–554, Springer-Verlag, 1999. [p. 243]

243. E. Fujisaki, T. Okamoto, D. Pointcheval, and J. Stern, "RSA-OAEP is secure under the RSA assumption." in *Proceedings of Crypto'01* (J. Kilian, ed.), no. 2139 in Lecture Notes in Computer Science, pp. 260–274, Springer-Verlag, 2001. `NES/DOC/ENS/WP3/004`. [p. 16, 257]

244. E. Fujiskai, T. Kobayashi, H. Morita, H. Oguro, T. Okamoto, and S. Okazaki, "ESIGN: Efficient digital signature scheme (submission to NESSIE)." Primitive submitted to NESSIE by NTT Corp., Sept. 2000. See also `http://info.isl.ntt.co.jp/esign/`. [p. 39, 53, 305, 306, 358]

245. J. Fuller and W. Millan, "On linear redundancy in S-boxes." in *Proceedings of Fast Software Encryption – FSE'03* (T. Johansson, ed.), Lecture Notes in Computer Science, Springer-Verlag, 2003. Earlier version available at `http://eprint.iacr.org/2002/111/`. [p. 70, 87, 107, 112, 114, 127, 528]
246. V. Furman, "Differential cryptanalysis of Nimbus." in *Proceedings of Fast Software Encryption – FSE'01* (M. Matsui, ed.), no. 2355 in Lecture Notes in Computer Science, pp. 187–195, Springer-Verlag, 2001. See also [72]. [p. 129]
247. S. Furuya and V. Rijmen, "Observations on Hierocrypt-3/l1 key-scheduling algorithms." in *Proceedings of the Second NESSIE Workshop*, 2001. [p. 127, 147]
248. K. Gaj and P. Chodowiec, "Comparison of the hardware performance of the AES candidates using reconfigurable hardware." in *Proceedings of the Third Advanced Encryption Standard Conference*, pp. 40–54, NIST, Apr. 2000.
249. S. D. Galbraith, J. Malone-Lee, and N. P. Smart, "Public key signatures in the multi-user setting." *Information Processing Letters*, vol. 83, no. 5, pp. 263–266, 2002. [p. 263]
250. R. P. Gallant, R. J. Lambert, and S. A. Vanstone, "Improving the parallelized Pollard lambda search on binary anomalous curves." *Mathematics of Computation*, vol. 69, pp. 1699–1705, 2000. [p. 221, 268]
251. K. Gandolfi, C. Mourtel, and F. Olivier, "Electromagnetic analysis: Concrete results." in *Proceedings of CHES'01* (Çetin Kaya Koç, D. Naccache, and C. Paar, eds.), no. 2162 in Lecture Notes in Computer Science, pp. 251–261, Springer-Verlag, 2001. [p. 338]
252. L. Gao, S. Shrivastava, and G. E. Sobelman, "Elliptic curve scalar multiplier design using FPGAs." in *Proceedings of CHES'99* (Çetin Kaya Koç and C. Paar, eds.), no. 1717 in Lecture Notes in Computer Science, pp. 257–268, Springer-Verlag, 2000. [p. 393]
253. W. Geiselmann, R. Steinwandt, and T. Beth, "Attacking the affine parts of SFLASH." in *Proceedings of Cryptography and Coding – CC'01* (B. Honary, ed.), no. 2260 in Lecture Notes in Computer Science, pp. 355–359, Springer-Verlag, 2001. Also in *Proceedings of the Second NESSIE Workshop*, 2001. Part of these results were presented in [255]. [p. 309]
254. W. Geiselmann, R. Steinwandt, and T. Beth, "Revealing 441 key bits of SFLASH^{v2}." in *Proceedings of the Third NESSIE Workshop*, 2002. Part of these results were presented in [255]. [p. 309]
255. W. Geiselmann, R. Steinwandt, and T. Beth, "Revealing the affine parts of SFLASH^{v1}, SFLASH^{v2}, and FLASH." in *Actas de la VII Reunión Española de Criptologia y Seguridad de la Información. Tomo I* (S. G. Jiménez and C. M. López, eds.), pp. 305–314, 2002. [p. 808]
256. R. Gennaro, S. Halevi, and T. Rabin, "Secure hash-and-sign signatures without the random oracle." in *Proceedings of Eurocrypt'99* (J. Stern, ed.), no. 1592 in Lecture Notes in Computer Science, pp. 123–139, Springer-Verlag, 1999. [p. 298]
257. H. Gilbert, H. Handschuh, A. Joux, and S. Vaudenay, "A statistical attack on RC6." in *Proceedings of Fast Software Encryption – FSE'00* (B. Schneier, ed.), no. 1978 in Lecture Notes in Computer Science, pp. 64–74, Springer-Verlag, 2000. [p. 111]
258. H. Gilbert and M. Minier, "A collision attack on seven rounds of Rijndael." in *Proceedings of the Third Advanced Encryption Standard Conference*, pp. 230–241, NIST, Apr. 2000. [p. 68, 112, 113, 119, 127, 144]
259. H. Gilbert and M. Minier, "Cryptanalysis of SFLASH." in *Proceedings of Eurocrypt'02* (L. R. Knudsen, ed.), no. 2332 in Lecture Notes in Computer Science, pp. 288–298, Springer-Verlag, 2002. [p. 315, 669]
260. M. Girault, "Self-certified public keys." in *Proceedings of Eurocrypt'91* (D. W. Davies, ed.), no. 547 in Lecture Notes in Computer Science, pp. 490–497, Springer-Verlag, 1991.

261. M. Girault and J.-C. Paillès, “On-line/off-line RSA-like.” in *Proc. Workshop on Coding and Cryptography*, pp. 223–232, INRIA, 2003. [p. 778, 784]
262. M. Girault, G. Poupard, and J. Stern, “Global Payment System (GPS): un protocole de signature à la volée.” in *Trusting Electronic Trade ’99*, 1999.
263. M. Girault, G. Poupard, and J. Stern, “Some modes of use of the GPS identification scheme.” in *Proceedings of the Third NESSIE Workshop*, 2002. [p. 290]
264. M. Girault and J.-J. Quisquater, “GQ+GPS.” Rump session talk at Eurocrypt’02, 2002. [p. 778, 782]
265. O. Goldreich, “Foundations of cryptography.” Available at `http://www.wisdom.weizmann.ac.il/~oded/foc-book.html`. Three volumes: basic tools (published [267]), basic applications (in preparation), and beyond the basics (in planning). [p. 51, 59, 160, 261, 297, 321, 329]
266. O. Goldreich, *Modern Cryptography, Probabilistic Proofs and Pseudorandomness*. No. 17 in Algorithms and Combinatorics, Springer-Verlag, 1999. [p. 160, 767]
267. O. Goldreich, *Foundations of Cryptography — Basic Tools*, vol. 1. Cambridge University Press, Aug. 2001. [p. 58, 809]
268. O. Goldreich, S. Goldwasser, and Silvio, “How to construct random functions.” *Journal of the ACM*, vol. 33, no. 4, pp. 792–807, 1986. [p. 764, 767]
269. O. Goldreich and L. A. Levin, “A hard core predicate for any one way function.” in *Proceedings of Symposium on Theory of Computing – STOC’89*, pp. 25–32, ACM Press, 1989. [p. 159, 160, 763]
270. S. Goldwasser and M. Bellare, “Lecture notes on cryptography.” Available at `http://www-cse.ucsd.edu/users/mihir/papers/gb.html`. [p. 51, 56, 59, 61]
271. S. Goldwasser and S. Micali, “Probabilistic encryption.” *Journal of Computer and System Sciences*, vol. 28, no. 2, pp. 270–299, 1984. [p. 216]
272. S. Goldwasser, S. Micali, and C. Rackoff, “The knowledge complexity of interactive proof systems.” in *Proceedings of Symposium on Theory of Computing – STOC’85*, pp. 291–304, ACM Press, 1985. [p. 321, 323]
273. S. Goldwasser, S. Micali, and R. L. Rivest, “A digital signature scheme secure against adaptive chosen message attacks.” *SIAM Journal on Computing*, vol. 17, no. 2, pp. 281–308, Apr. 1988. [p. 260–262, 296]
274. L. Granboulan, “How to repair ESIGN.” in *Proceedings of SCN’02* (S. Cimato, C. Galdi, and G. Persiano, eds.), vol. 2576 of *Lecture Notes in Computer Science*, Springer-Verlag, 2002. `NES/DOC/ENS/WP5/019`. Also available at `http://eprint.iacr.org/2002/074/` and also in *Proceedings of the Third NESSIE Workshop*, 2002. [p. 263, 266, 277, 306]
275. L. Granboulan, “RSA hybrid encryption schemes.” Public report, NESSIE, 2002. `NES/DOC/ENS/WP5/012`. [p. 232, 248, 249]
276. L. Granboulan, “Short signatures in the random oracle model.” in *Proceedings of Asiacrypt’02* (Y. Zheng, ed.), no. 2501 in Lecture Notes in Computer Science, pp. 364–378, Springer-Verlag, 2002. `NES/DOC/ENS/WP5/021`. [p. 272, 280, 281]
277. L. C. Guillou and J.-J. Quisquater, “A “paradoxical” identity-based signature scheme resulting from zero-knowledge.” in *Proceedings of Crypto’88* (S. Goldwasser, ed.), no. 403 in Lecture Notes in Computer Science, pp. 216–231, Springer-Verlag, 1988. [p. 336, 782]
278. L. C. Guillou and J.-J. Quisquater, “A practical Zero-Knowledge protocol fitted to security microprocessor minimizing both transmission and memory.” in *Proceedings of Eurocrypt’88* (C. Günther, ed.), vol. 330 of *Lecture Notes in Computer Science*, pp. 123–128, Springer-Verlag, 1988. [p. 334, 335]
279. S. Halevi, D. Coppersmith, and C. S. Jutla, “Scream: A software-efficient stream cipher.” in *Proceedings of Fast Software Encryption – FSE’02* (J. Daemen and V. Rijmen, eds.), no. 2365 in Lecture Notes in Computer Science, pp. 195–209, Springer-Verlag, 2002. [p. 157]

280. H. Handschuh, L. R. Knudsen, and M. J. B. Robshaw, “Analysis of SHA-1 in encryption mode.” in *Proceedings of CT-RSA'01* (D. Naccache, ed.), no. 2020 in Lecture Notes in Computer Science, pp. 70–83, Springer-Verlag, 2001. [p. 123, 555]
281. H. Handschuh and D. Naccache, “SHACAL.” Primitive submitted to NESSIE by Gemplus, Sept. 2000. [p. 38, 53, 121, 123, 188, 189, 357, 555, 556]
282. R. Harasawa, J. Shikata, J. Suzuki, and H. Imai, “Comparing the MOV and FR reductions in elliptic curve cryptography.” in *Proceedings of Eurocrypt'99* (J. Stern, ed.), no. 1592 in Lecture Notes in Computer Science, pp. 190–205, Springer-Verlag, 1999. [p. 268]
283. C. Harpes, G. G. Kramer, and J. L. Massey, “A generalization of linear cryptanalysis and the applicability of Matsui's piling-up lemma.” in *Proceedings of Eurocrypt'95* (L. C. Guillou and J.-J. Quisquater, eds.), no. 921 in Lecture Notes in Computer Science, pp. 24–38, Springer-Verlag, 1995. [p. 82]
284. J. Håstad, R. Impagliazzo, L. A. Levin, and M. Luby, “Pseudo random number generators from any one-way function.” *SIAM Journal on Computing*, vol. 28, pp. 1364–1396, 1999. [p. 159, 160]
285. J. Håstad and M. Nāslund, “BMGL: Synchronous key-stream generator with provable security.” Primitive submitted to NESSIE, Sept. 2000. [p. 38, 53, 158, 358, 763–769, 774, 775]
286. J. Håstad and M. Nāslund, “A generalized interface for the NESSIE submission BMGL.” Internal document, NESSIE, 2001. [p. 160]
287. Y. Hatano, H. Sekine, and T. Kaneko, “Higher order differential attack of Camellia (II).” in *Proceedings of Selected Areas in Cryptography – SAC'02* (K. Nyberg and H. M. Heys, eds.), no. 2595 in Lecture Notes in Computer Science, pp. 39–56, Springer-Verlag, 2002. [p. 143, 715, 729]
288. P. Hawkes, “Differential-linear weak key classes of IDEA.” in *Proceedings of Eurocrypt'98* (K. Nyberg, ed.), no. 1403 in Lecture Notes in Computer Science, pp. 112–126, Springer-Verlag, 1998. [p. 82, 84, 85, 138]
289. P. Hawkes and G. G. Rose, “SOBER.” Primitive submitted to NESSIE by Qualcomm International, Sept. 2000. [p. 38, 53, 161, 163–166, 358]
290. P. Hawkes and G. G. Rose, “Guess-and-determine attacks on SNOW.” in *Proceedings of Selected Areas in Cryptography – SAC'02* (K. Nyberg and H. M. Heys, eds.), no. 2595 in Lecture Notes in Computer Science, Springer-Verlag, 2002. [p. 161]
291. P. Hawkes and G. G. Rose, “On the applicability of distinguishing attacks against stream ciphers.” in *Proceedings of the Third NESSIE Workshop*, 2002. [p. 166]
292. Y. He and S. Qing, “Square attack on reduced Camellia cipher.” in *Proceedings of ICICS'01* (S. Qing, T. Okamoto, and J. Zhou, eds.), no. 2229 in Lecture Notes in Computer Science, pp. 238–245, Springer-Verlag, 2001. [p. 107, 143, 528, 715, 729]
293. Helion Technology Limited, The Granary, Home End, Fulbourn, Cambridge CB1 5BS, UK., *DATASHEET - Compact Dual Hash Processor Core for Xilinx FPGA*. URL: `www.heliontech.com`. [p. 392]
294. J. C. Hernández, J. M. Sierra, J. C. Mex-Perera, D. Borrajo, A. Ribagorda, and P. Isasi, “Using the general next bit predictor like an evaluation criteria.” Methodology submitted to NESSIE, Sept. 2000. [p. 39, 53]
295. A. Hevia and M. A. Kiwi, “Strength of two data encryption standard implementations under timing attacks.” *ACM Transactions on Information and System Security (TISSEC)*, vol. 2, pp. 416–437, 1999. [p. 338]
296. H. M. Heys, *The design of substitution-permutation network ciphers resistant to cryptanalysis*. Ph.d. thesis, Queen's University, Kingston, Canada, 1994. [p. 750]
297. S. Hong, S. Lee, J. Lim, J. Sung, D. Cheon, and I. Cho, “Provable security against differential and linear cryptanalysis for the SPN structure.” in *Proceedings of Fast Software Encryption – FSE'00* (B. Schneier, ed.), no. 1978 in Lecture Notes in Computer Science, pp. 273–283, Springer-Verlag, 2000. [p. 754]

298. P. Horster, M. Michels, and H. Petersen, "Meta-ElGamal signature schemes." in *Proceedings of Conference on Computer and Communications Security – CCS'94*, ACM Press, 1994. Full paper available at `http://www.geocities.com/CapeCanaveral/Lab/8967/TR-94-5.ps.gz`. [p. 287]
299. IEEE 1363-2000, "Standard Specifications for Public-Key Cryptography." Aug. 2000. Available from `http://standards.ieee.org/catalog/olis/busarch.html`. [p. 224, 302, 665]
300. IEEE P1363 working group, "Standard Specifications for Public-Key Cryptography.". Main web page at `http://grouper.ieee.org/groups/1363/`. [p. 224, 302, 666, 667]
301. International Organization for Standardization, "ISO/IEC 9796-2: Information Technology - Security Techniques - Digital signature schemes giving message recovery - Part 2: Integer factorization based mechanisms." 2002. [p. 302]
302. International Organization for Standardization, "ISO/IEC 9796-3: Information Technology - Security Techniques - Digital signature schemes giving message recovery - Part 3: Discrete logarithm based mechanisms." 2000. [p. 302]
303. International Organization for Standardization, "ISO/IEC 9797-1: Information Technology - Security Techniques - Message Authentication Codes (MACs) - Part 1: Mechanisms using a block cipher." 1999. [p. 199, 201, 366, 617]
304. International Organization for Standardization, "ISO/IEC 9797-2: Information Technology - Security Techniques - Message Authentication Codes (MACs) - Part 2: Mechanisms using a dedicated hash-function." 2002. [p. 200, 201, 623]
305. International Organization for Standardization, "ISO/IEC 10118-2: Information Technology - Security Techniques - Hash-functions - Part 2: Hash-functions using an n-bit block cipher." 2000. [p. 177, 180]
306. International Organization for Standardization, "ISO/IEC 10118-3: Information Technology - Security Techniques - Hash-functions - Part 3: Dedicated hash-functions." 1998. [p. 180, 605]
307. International Organization for Standardization, "ISO/IEC 10118-4: Information Technology - Security Techniques - Hash-functions - Part 4: Hash-functions using modular arithmetic." 1998. [p. 178, 180]
308. International Organization for Standardization, "ISO/IEC 14888-2: Information Technology - Security Techniques - Digital Signatures with Appendix - Part 2: Identity-based Mechanisms." 1998. [p. 302]
309. International Organization for Standardization, "ISO/IEC 14888-3: Information Technology - Security Techniques - Digital Signatures with Appendix - Part 3: Certificate-based Mechanisms." 1998. [p. 302, 665]
310. International Organization for Standardization, "ISO/IEC 15946-2: Information technology - Security techniques - Cryptographic techniques based on elliptic curves - Part 2: Digital signatures." 2002. [p. 302, 793]
311. International Organization for Standardization, "ISO/IEC 18032: Information technology - Prime number generation." 2002. Committee Draft. [p. 659]
312. International Organization for Standardization, "ISO/IEC 18033-2: Information technology - Security techniques - Encryption algorithms - Part 2: Asymmetric ciphers." 2002. Working Draft. [p. 224, 633, 639, 645]
313. G. Itkis and L. Reyzin, "Forward-secure signatures with optimal signing and verifying." in *Proceedings of Crypto'01* (J. Kilian, ed.), no. 2139 in Lecture Notes in Computer Science, pp. 332–354, Springer-Verlag, 2001. [p. 262]
314. T. Jakobsen, "Cryptanalysis of block ciphers with probabilistic non-linear relations of low degree." in *Proceedings of Crypto'98* (H. Krawczyk, ed.), no. 1462 in Lecture Notes in Computer Science, pp. 212–223, Springer-Verlag, 1998. [p. 67]
315. T. Jakobsen and L. R. Knudsen, "The interpolation attack on block ciphers." in *Proceedings of Fast Software Encryption – FSE'97* (E. Biham, ed.), no. 1267 in Lecture Notes in Computer Science, pp. 28–40, Springer-Verlag, 1997. [p. 67]

316. T. Jakobsen and L. R. Knudsen, "Attacks on block ciphers of low algebraic degree." *Journal of Cryptology*, vol. 14, pp. 197–210, 2001. [p. 67]

317. S. Janssens, J. Thomas, W. Borremans, P. Gijsels, I. Verbauwhede, F. Vercauteren, B. Preneel, and J. Vandewalle, "Hardware/software co-design of an elliptic curve public-key cryptosystem." in *Proceedings IEEE Workshop on of Signal Processing Systems*, pp. 209–216, 2001. [p. 393]

318. E. Jaulmes, A. Joux, and F. Valette, "On the security of randomized CBC-MAC beyond the birthday paradox limit: A new construction." in *Proceedings of Fast Software Encryption – FSE'02* (J. Daemen and V. Rijmen, eds.), no. 2365 in Lecture Notes in Computer Science, pp. 237–251, Springer-Verlag, 2002. [p. 208]

319. D. B. Johnson and S. Blake-Wilson, "ECDSA." Primitive submitted to NESSIE by Certicom, Sept. 2000. [p. 39, 53, 282, 289, 302, 358, 392]

320. D. B. Johnson and S. Blake-Wilson, "ECIES." Primitive submitted to NESSIE by Certicom, Sept. 2000. [p. 39, 53, 358]

321. F. Jönsson and T. Johansson, "A fast correlation attack on LILI-128." Technical report, Lund University, 2001. Available at `http://www.it.lth.se/thomas/papers/paper140.ps`. [p. 169]

322. J. Jonsson, "Security proof for the RSA-PSS signature scheme." in *Proceedings of the Second NESSIE Workshop*, 2001. [p. 310, 311, 657]

323. J. Jonsson, "An OAEP variant with a tight security proof." Available at `http://eprint.iacr.org/2002/034/`, Mar. 2002. [p. 272]

324. J. Jonsson and B. S. Kaliski, Jr, "RC6 block cipher." Primitive submitted to NESSIE by RSA, Sept. 2000. [p. 37, 52, 108, 110, 120, 143, 357]

325. J. Jonsson and B. S. Kaliski, Jr, "RSA-OAEP." Primitive submitted to NESSIE by RSA, Sept. 2000. [p. 39, 53, 247, 256, 358]

326. J. Jonsson and B. S. Kaliski, Jr, "RSA-PSS." Primitive submitted to NESSIE by RSA, Sept. 2000. [p. 39, 53, 310, 358, 657]

327. J. Jorge Nakahara, "An update to linear cryptanalysis of SAFER++." Internal document, NESSIE, 2003. [p. 99, 117, 144]

328. A. Joux and G. Martinet, "Weaknesses in Quartz signature scheme." Public report, NESSIE, 2002. `NES/DOC/ENS/WP5/026`. [p. 310]

329. A. Joux and K. Nguyen, "Separating decision Diffie-Hellman from Diffie-Hellman in cryptographic groups." Available at `http://eprint.iacr.org/2001/003/`, 2001. [p. 267]

330. M. Joye, A. K. Lenstra, and J.-J. Quisquater, "Chinese remaindering based cryptosystems in the presence of faults." *Journal of Cryptology*, vol. 12, no. 4, pp. 241–245, 1999. Also available from `http://www.geocities.com/CapeCanaveral/Launchpad/9160/publications.html`. [p. 313, 346]

331. M. Joye and P. Paillier, "Constructive methods for the generation of prime numbers." in *Proceedings of the Second NESSIE Workshop*, 2001. [p. 265]

332. M. Joye, P. Paillier, and S. Vaudenay, "Efficient generation of prime numbers." in *Proceedings of CHES'00* (Çetin Kaya Koç and C. Paar, eds.), no. 1965 in Lecture Notes in Computer Science, pp. 340–354, Springer-Verlag, 2000. [p. 695–697, 699, 700, 705, 706]

333. M. Joye, J.-J. Quisquater, S.-M. Yen, and M. Yung, "Observability analysis: Detecting when improved cryptosystems fail." in *Proceedings of CT-RSA'02* (B. Preneel, ed.), no. 2271 in Lecture Notes in Computer Science, pp. 17–29, Springer-Verlag, 2002. [p. 347]

334. B. S. Kaliski, Jr, "On hash function firewalls in signature scheme." in *Proceedings of CT-RSA'02* (B. Preneel, ed.), no. 2271 in Lecture Notes in Computer Science, pp. 1–16, Springer-Verlag, 2002. Earlier version available at `http://grouper.ieee.org/groups/1363/Research/contributions/HashFunctionFirewalls.pdf`. [p. 260, 311]

335. B. S. Kaliski, Jr and M. J. B. Robshaw, "Linear cryptanalysis using multiple approximations." in *Proceedings of Crypto'94* (Y. Desmedt, ed.), no. 839 in Lecture Notes in Computer Science, pp. 26–39, Springer-Verlag, 1994. [p. 65]
336. B. S. Kaliski, Jr and Y. L. Yin, "On differential and linear cryptanalsis of the RC5 encryption algorithm." in *Proceedings of Crypto'95* (D. Coppersmith, ed.), no. 963 in Lecture Notes in Computer Science, pp. 171–184, Springer-Verlag, 1995. [p. 108]
337. M. Kanda, "Practical security evaluation against differential and linear cryptanalysis for Feistel ciphers with SPN round function." in *Proceedings of Selected Areas in Cryptography – SAC'00* (D. R. Stinson and S. E. Tavares, eds.), no. 2012 in Lecture Notes in Computer Science, pp. 324–338, Springer-Verlag, 2001. [p. 720]
338. M. Kanda and T. Matsumoto, "Security of Camellia against truncated differential cryptanalysis." in *Proceedings of Fast Software Encryption – FSE'01* (M. Matsui, ed.), Lecture Notes in Computer Science, pp. 286–299, Springer-Verlag, 2001. [p. 715]
339. T. Kawabata and T. Kaneko, "A study on higher order differential attack of Camellia." in *Proceedings of the Second NESSIE Workshop*, 2001. [p. 107, 528, 715, 729]
340. KCDSA Task Force Team, "The Korean Certificate-based Digital Signature Algorithm." in *Proceedings of Asiacrypt'98* (K. Ohta and D. Pei, eds.), no. 1514 in Lecture Notes in Computer Science, pp. 175–186, Springer-Verlag, 1998. Also available at `http://grouper.ieee.org/groups/1363/P1363a/PSSigs.html` as an IEEE P1363a submission. [p. 264, 282, 289, 304]
341. L. Keliher, H. Meijer, and S. E. Tavares, "High probability linear hulls in Q." in *Proceedings of the Second NESSIE Workshop*, 2001. Available from `http://mathcs.mta.ca/faculty/lkeliher/publications.html`. [p. 134, 135, 148]
342. L. Keliher, H. Meijer, and S. E. Tavares, "Improving the upper bound on the maximum average linear hull probability for Rijndael." in *Proceedings of Selected Areas in Cryptography – SAC'01* (S. Vaudenay and A. M. Youssef, eds.), no. 2259 in Lecture Notes in Computer Science, pp. 112–128, Springer-Verlag, 2001. [p. 112, 113, 750, 754]
343. L. Keliher, H. Meijer, and S. E. Tavares, "New method for upper bounding the maximum average linear hull probability for SPNs." in *Proceedings of Eurocrypt'01* (B. Pfitzmann, ed.), no. 2045 in Lecture Notes in Computer Science, pp. 420–436, Springer-Verlag, 2001. [p. 750, 752, 754]
344. J. Kelsey, T. Kohno, and B. Schneier, "Amplified boomerang attacks against reduced-round MARS and Serpent." in *Proceedings of Fast Software Encryption – FSE'00* (B. Schneier, ed.), no. 1978 in Lecture Notes in Computer Science, pp. 75–93, Springer-Verlag, 2000. [p. 722, 727]
345. J. Kelsey, B. Schneier, and D. Wagner, "Key-schedule cryptanalysis of 3-WAY, IDEA, G-DES, RC4, SAFER, and Triple-DES." in *Proceedings of Crypto'96* (N. Koblitz, ed.), no. 1109 in Lecture Notes in Computer Science, pp. 237–251, Springer-Verlag, 1996. [p. 67, 102, 140, 710]
346. J. Kelsey, B. Schneier, and D. Wagner, "Mod n cryptanalysis, with applications against RC5P and M6." in *Proceedings of Fast Software Encryption – FSE'99* (L. R. Knudsen, ed.), no. 1636 in Lecture Notes in Computer Science, pp. 139–155, Springer-Verlag, 1999. [p. 66, 111]
347. J. Kim, D. Moon, W. Lee, S. Hong, S. Lee, and S. Jung, "Amplified boomerang attack against reduced-round SHACAL." in *Proceedings of Asiacrypt'02* (Y. Zheng, ed.), no. 2501 in Lecture Notes in Computer Science, pp. 243–253, Springer-Verlag, 2002. Also in *Proceedings of the Third NESSIE Workshop*, 2002. [p. 124, 146, 189]
348. A. Klapper and M. Goresky, "Feedback shift registers, 2-adic span, and combiners with memory." *Journal of Cryptology*, vol. 10, no. 2, pp. 111–147, 1997. [p. 73, 154]

349. V. Klíma and T. Rosa, "Further results and considerations on side channel attacks on RSA." in *Proceedings of CHES'02* (B. S. Kaliski, Çetin Kaya Koç, and C. Paar, eds.), no. 2535 in Lecture Notes in Computer Science, Springer-Verlag, 2002. Also available at `http://eprint.iacr.org/2002/071/`. [p. 249, 258, 313, 340, 345, 346]

350. L. R. Knudsen, "Cryptanalysis of LOKI'91." in *Proceedings of Auscrypt'92* (J. Seberry and Y. Zheng, eds.), no. 718 in Lecture Notes in Computer Science, pp. 196–208, Springer-Verlag, 1993. [p. 67, 68]

351. L. R. Knudsen, "A key-schedule weakness in SAFER K-64." in *Proceedings of Crypto'95* (D. Coppersmith, ed.), no. 963 in Lecture Notes in Computer Science, pp. 271–286, Springer-Verlag, 1995. [p. 67, 78, 98, 116, 117]

352. L. R. Knudsen, "Truncated and higher order differentials." in *Proceedings of Fast Software Encryption – FSE'94* (B. Preneel, ed.), no. 1008 in Lecture Notes in Computer Science, pp. 196–211, Springer-Verlag, 1995. [p. 64, 93, 502, 718]

353. L. R. Knudsen, "DEAL - a 128-bit block cipher." Technical report 151, Dept. of Informatics, University of Bergen, Norway, 1998. [p. 64, 94, 106]

354. L. R. Knudsen, "Contemporary block ciphers." in *Lectures on Data Security. Modern Cryptology in in Theory and Practice, LNCS Tutorial 1561* (I. B. Damgård, ed.), pp. 105–126, Springer-Verlag, 1999. Also available from `http://www.ii.uib.no/~larsr/papers/survey98.ps`. [p. 55, 62]

355. L. R. Knudsen, "Analysis of Camellia." 2000. Contribution for ISO/IEC JTC SC27. Available at `http://info.isl.ntt.co.jp/camellia/Publications/knudsen.ps`. [p. 715]

356. L. R. Knudsen, "A detailed analysis of SAFER K." *Journal of Cryptology*, vol. 13, pp. 417–436, 2000. [p. 78, 98, 99, 116, 117]

357. L. R. Knudsen, "The number of rounds in block ciphers." Public report, NESSIE, 2000. `NES/DOC/UIB/WP3/003`. [p. 76]

358. L. R. Knudsen, "Analysis of RMAC." Public report, NESSIE, 2002. `NES/DOC/UIB/WP5/024`. [p. 209, 212]

359. L. R. Knudsen, "Correlations in RC6 on 256-bit blocks." Public report, NESSIE, Sept. 2002. `NES/DOC/UIB/WP5/022`. [p. 69, 120, 146]

360. L. R. Knudsen, "Non-random properties of reduced-round Whirlpool." Public report, NESSIE, 2002. `NES/DOC/UIB/WP5/016`. [p. 184]

361. L. R. Knudsen, "Quadratic relations in Khazad and Whirlpool." Public report, NESSIE, 2002. `NES/DOC/UIB/WP5/017`. [p. 65, 185]

362. L. R. Knudsen and T. A. Berson, "Truncated differentials of SAFER." in *Proceedings of Fast Software Encryption – FSE'96* (D. Gollmann, ed.), no. 1039 in Lecture Notes in Computer Science, pp. 15–26, Springer-Verlag, 1996. [p. 98, 116]

363. L. R. Knudsen and T. Kohno, "Analysis of RMAC." in *Proceedings of Fast Software Encryption – FSE'03* (T. Johansson, ed.), Lecture Notes in Computer Science, Springer-Verlag, 2003. [p. 209, 212]

364. L. R. Knudsen, X. Lai, and B. Preneel, "Attacks on fast double block length hash functions." *Journal of Cryptology*, vol. 11, no. 1, pp. 59–72, 1998. [p. 177]

365. L. R. Knudsen and W. Meier, "Improved differential attacks on RC5." in *Proceedings of Crypto'96* (N. Koblitz, ed.), no. 1109 in Lecture Notes in Computer Science, pp. 216–228, Springer-Verlag, 1996. [p. 108]

366. L. R. Knudsen and W. Meier, "Correlations in RC6 with a reduced number of rounds." in *Proceedings of Fast Software Encryption – FSE'00* (B. Schneier, ed.), no. 1978 in Lecture Notes in Computer Science, pp. 94–108, Springer-Verlag, 2000. [p. 69, 110, 111, 143]

367. L. R. Knudsen and H. Raddum, "Recommendation to NIST for the AES." Public report, NESSIE, May 2000. `NES/DOC/UIB/WP3/005`. [p. 112, 115]

368. L. R. Knudsen and H. Raddum, "On Noekeon." in *Proceedings of the Second NESSIE Workshop*, 2001. `NES/DOC/UIB/WP3/009`. [p. 133, 134, 148]

369. L. R. Knudsen and V. Rijmen, "Truncated differentials of IDEA." Technical report 97/1, Department of Electrical Engineering, ESAT–COSIC, 1997. [p. 84]
370. L. R. Knudsen and M. J. B. Robshaw, "Non-linear approximations in linear cryptanalysis." in *Proceedings of Eurocrypt'96* (U. Maurer, ed.), no. 1070 in Lecture Notes in Computer Science, pp. 224–236, Springer-Verlag, 1996. [p. 65]
371. L. R. Knudsen and D. Wagner, "Integral cryptanalysis (extended abstract)." in *Proceedings of Fast Software Encryption – FSE'02* (J. Daemen and V. Rijmen, eds.), no. 2365 in Lecture Notes in Computer Science, pp. 112–127, Springer-Verlag, 2002. NES/DOC/UIB/WP5/015. [p. 68, 69, 94, 112, 139, 502]
372. D. E. Knuth, *Seminumerical algorithms, (2 ed.)*, vol. 2 of *The art of computer programming*. Addison-Wesley, 1982. [p. 697, 769]
373. N. Koblitz, "Elliptic curve cryptosystems." *Mathematics of Computation*, vol. 48, pp. 203–209, 1987. [p. 633, 645]
374. P. C. Kocher, "Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems." in *Proceedings of Crypto'96* (N. Koblitz, ed.), no. 1109 in Lecture Notes in Computer Science, pp. 104–113, Springer-Verlag, 1996. [p. 13, 337]
375. P. C. Kocher, J. Jaffe, and B. Jun, "Differential power analysis." in *Proceedings of Crypto'99* (M. J. Wiener, ed.), no. 1666 in Lecture Notes in Computer Science, pp. 388–397, Springer-Verlag, 1999. [p. 13, 338]
376. H. Krawczyk, "Simple forward-secure signatures from any signature scheme." in *Proceedings of Conference on Computer and Communications Security – CCS'00*, ACM Press, 2000. [p. 262]
377. H. Krawczyk and T. Rabin, "Chameleon signatures." in *Proceedings of Network and Distributed System Security Symposium – NDSS'00*, pp. 143–154, The Internet Society, 2000. Also available at http://www.research.ibm.com/security/chameleon.ps. [p. 297]
378. T. Krovetz, *Software-optimized universal hashing and message authentication*. Doctoral dissertation, University of California Davis, 2000. [p. 196]
379. T. Krovetz, J. Black, S. Halevi, A. Hevia, H. Krawczyk, and P. Rogaway, "UMAC." Primitive submitted to NESSIE, Sept. 2000. [p. 38, 53, 205, 208, 358, 583, 591, 597]
380. U. Kühn, "Cryptanalysis of reduced-round MISTY." in *Proceedings of Eurocrypt'01* (B. Pfitzmann, ed.), no. 2045 in Lecture Notes in Computer Science, pp. 325–339, Springer-Verlag, 2001. [p. 94, 139, 502]
381. U. Kühn, "Improved cryptanalysis of MISTY1." in *Proceedings of Fast Software Encryption – FSE'02* (J. Daemen and V. Rijmen, eds.), no. 2365 in Lecture Notes in Computer Science, pp. 61–75, Springer-Verlag, 2002. Also in *Proceedings of the Second NESSIE Workshop*, 2001. [p. 94, 139, 502]
382. K. Kurosawa, T. Iwata, and V. D. Quang, "Root finding interpolation attack." in *Proceedings of Selected Areas in Cryptography – SAC'00* (D. R. Stinson and S. E. Tavares, eds.), no. 2012 in Lecture Notes in Computer Science, pp. 303–314, Springer-Verlag, 2001. [p. 67]
383. X. Lai, *On the Design and Security of Block Ciphers*. Hartung-Gorre Verlag, Konstanz, 1992. [p. 64, 82]
384. X. Lai, *Communication and Cryptography, Two Sides of One Tapestry*. Kluwer Academic Publishers, 1994. [p. 64]
385. X. Lai, "Higher order derivatives and differential cryptanalysis." in *In Proceedings of "Symposium on Communication, Coding and Cryptography", in honor of James L. Massey on the occasion of his 60th birthday*, 1994. [p. 64, 93, 502]
386. X. Lai and J. L. Massey, "A proposal for a new block encryption standard." in *Proceedings of Eurocrypt'90* (I. B. Damgård, ed.), no. 473 in Lecture Notes in Computer Science, pp. 389–404, Springer-Verlag, 1990. [p. 815]
387. X. Lai and J. L. Massey, "IDEA." Primitive submitted to NESSIE by R. Straub, MediaCrypt AG, Sept. 2000. Based on [386] and [388]. [p. 37, 52, 80, 357]

388. X. Lai, J. L. Massey, and S. Murphy, “Markov ciphers and differential cryptanalysis.” in *Proceedings of Eurocrypt’91* (D. W. Davies, ed.), no. 547 in Lecture Notes in Computer Science, pp. 17–38, Springer-Verlag, 1991. [p. 63, 64, 749, 815]

389. S. K. Langford and M. E. Hellman, “Differential-linear cryptanalysis.” in *Proceedings of Crypto’94* (Y. Desmedt, ed.), no. 839 in Lecture Notes in Computer Science, pp. 17–25, Springer-Verlag, 1994. [p. 65, 140]

390. J. Lano and G. Peeters, “Cryptanalyse van NESSIE kandidaten.” Master’s thesis, ESAT-COSIC, KU-Leuven, 2002. [p. 343]

391. A. N. Lebedev and A. A. Volchkov, “NUSH.” Primitive submitted to NESSIE by LAN Crypto, Int., Sept. 2000. [p. 37, 52, 129, 134, 357]

392. S. Lee, S. Hong, S. Lee, J. Lim, and S. Yoon, “Truncated differential cryptanalysis of Camellia.” in *Proceedings of ICISC’01* (K. Kim, ed.), no. 2288 in Lecture Notes in Computer Science, pp. 32–38, Springer-Verlag, 2001. [p. 143, 715, 729]

393. A. K. Lenstra and H. W. Lenstra, Jr, eds., *The development of the number field sieve*. No. 1554 in Lecture Notes in Mathematics, Springer-Verlag, 1993. [p. 221, 267]

394. A. K. Lenstra, A. Shamir, J. Tomlinson, and E. Tromer, “Analysis of Bernstein’s factorization circuit.” in *Proceedings of Asiacrypt’02* (Y. Zheng, ed.), no. 2501 in Lecture Notes in Computer Science, pp. 1–26, Springer-Verlag, 2002. [p. 269]

395. A. K. Lenstra and E. R. Verheul, “Selecting cryptographic key sizes.” in *Proceedings of Public Key Cryptography – PKC’00* (H. Imai and Y. Zheng, eds.), no. 1751 in Lecture Notes in Computer Science, pp. 446–465, Springer-Verlag, 2000. [p. 816]

396. A. K. Lenstra and E. R. Verheul, “Selecting cryptographic key sizes.” *Journal of Cryptology*, vol. 14, no. 4, pp. 255–293, Autumn 2001. Full version of [395]. [p. 268, 269]

397. H. W. Lenstra, Jr, “Factoring integers with elliptic curves.” *Annals of Mathematics*, vol. 126, no. 3, pp. 649–673, Nov. 1987. Second series. [p. 221, 267]

398. K. H. Leung, S. L. Ma, Y. L. Wong, and P. H. W. Leong, “FPGA implementation of a microcoded elliptic curve cryptographic processor.” in *Proceedings of Field-Programmable Custom Computing Machines (FCCM’00)*, pp. 68–76, 2000. [p. 393]

399. C. H. Lim and P. J. Lee, “A key recovery attack on discrete log-based schemes using a prime order subgroup.” in *Proceedings of Crypto’97* (B. S. Kaliski, Jr, ed.), no. 1294 in Lecture Notes in Computer Science, pp. 249–263, Springer-Verlag, 1997. [p. 223, 634, 646]

400. H. Lipmaa, “IDEA — a cipher for multimedia architectures?.” in *Proceedings of Selected Areas in Cryptography – SAC’98* (S. E. Tavares and H. Meijer, eds.), no. 1556 in Lecture Notes in Computer Science, pp. 248–263, Springer-Verlag, 1999. [p. 385]

401. H. Lipmaa, “Fast software implementations of SC2000.” in *Proceedings of Information Security Conference – ISC’02* (A. H. Chan and V. D. Gligor, eds.), no. 2433 in Lecture Notes in Computer Science, pp. 63–74, Springer-Verlag, 2002. [p. 385]

402. M. Luby, *Pseudorandomness and Cryptographic Applications*. Princeton Computer Science Notes, Princeton University Press, 1996. [p. 160]

403. M. Luby and C. Rackoff, “How to construct pseudorandom permutations from pseudorandom functions.” in *Proceedings of Crypto’85* (H. C. Williams, ed.), no. 218 in Lecture Notes in Computer Science, p. 447, Springer-Verlag, 1986. Full version in [404]. [p. 816]

404. M. Luby and C. Rackoff, “How to construct pseudorandom permutations from pseudorandom functions.” *SIAM Journal on Computing*, vol. 17, no. 2, pp. 373–386, Apr. 1988. An abstract appeared in [403]. [p. 60, 816]

405. M. Luby and C. Rackoff, “A study of password security.” in *Proceedings of Crypto’87* (C. Pomerance, ed.), no. 293 in Lecture Notes in Computer Science, pp. 392–397, Springer-Verlag, 1988. [p. 60]

406. S. Lucks, "Attacking triple encryption." in *Proceedings of Fast Software Encryption – FSE'98* (S. Vaudenay, ed.), no. 1372 in Lecture Notes in Computer Science, pp. 239–253, Springer-Verlag, 1998. [p. 102, 140]

407. S. Lucks, "Attacking seven rounds of Rijndael under 192-bit and 256-bit keys." in *Proceedings of the Third Advanced Encryption Standard Conference*, pp. 215–229, NIST, Apr. 2000. [p. 112, 113, 144]

408. S. Lucks, "The saturation attack - a bait for Twofish." in *Proceedings of Fast Software Encryption – FSE'01* (M. Matsui, ed.), no. 2355 in Lecture Notes in Computer Science, pp. 1–15, Springer-Verlag, 2001. [p. 68]

409. S. Lucks, "A variant of the Cramer-Shoup cryptosystem for groups of unknown order." in *Proceedings of Asiacrypt'02* (Y. Zheng, ed.), no. 2501 in Lecture Notes in Computer Science, pp. 27–45, Springer-Verlag, 2002. [p. 237]

410. A. W. Machado, "Nimbus." Primitive submitted to NESSIE, Sept. 2000. [p. 37, 52, 128, 357]

411. T. Malkin, D. Micciancio, and S. Miner, "Efficient generic forward-secure signatures with an unbounded number of time periods." in *Proceedings of Eurocrypt'02* (L. R. Knudsen, ed.), no. 2332 in Lecture Notes in Computer Science, pp. 400–417, Springer-Verlag, 2002. [p. 262]

412. J. Malone-Lee and N. P. Smart, "Modifications of ECDSA." in *Proceedings of Selected Areas in Cryptography – SAC'02* (K. Nyberg and H. M. Heys, eds.), no. 2595 in Lecture Notes in Computer Science, Springer-Verlag, 2002. [p. 289]

413. S. Mangard, "A simple power-analysis (SPA) attack on implementations of the AES key expansion." in *Proceedings of ICISC'02* (K. Kim, ed.), no. 2587 in Lecture Notes in Computer Science, Springer-Verlag, 2002. [p. 339]

414. J. Manger, "A chosen ciphertext attack on RSA optimal asymmetric encryption padding (OAEP) as standardized in PKCS # 1 v2.0." in *Proceedings of Crypto'01* (J. Kilian, ed.), no. 2139 in Lecture Notes in Computer Science, pp. 230–238, Springer-Verlag, 2001. [p. 16, 258, 343]

415. I. Mantin and A. Shamir, "A practical attack on broadcast RC4." in *Proceedings of Fast Software Encryption – FSE'01* (M. Matsui, ed.), no. 2355 in Lecture Notes in Computer Science, pp. 152–164, Springer-Verlag, 2001. [p. 170]

416. G. Marsaglia, "The diehard statistical tests.". Available at `http://stat.fsu.edu/~geo/diehard.html`. [p. 769]

417. G. Martinet, "RSA-OAEP and RSA-PSS." Public report, NESSIE, 2001. `NES/DOC/ENS/WP3/007`. [p. 257]

418. G. Martinet, "The security assumptions." Public report, NESSIE, 2001. `NES/DOC/ENS/WP3/005`. [p. 218]

419. J. L. Massey, "Shift register synthesis and BCH decoding." *IEEE Transactions on Information Theory*, vol. IT-15, pp. 122–127, 1969. [p. 153]

420. J. L. Massey, G. Khachatrian, and M. K. Kuregian, "Nomination of SAFER++ as candidate algorithm for the New European Schemes for Signatures, Integrity, and Encryption (NESSIE)." Primitive submitted to NESSIE by Cylink Corp., Sept. 2000. [p. 38, 52, 95, 98, 116, 357]

421. M. Matsui, "Linear cryptanalysis method for DES cipher." in *Proceedings of Eurocrypt'93* (T. Helleseth, ed.), no. 765 in Lecture Notes in Computer Science, pp. 386–397, Springer-Verlag, 1993. [p. 64, 715, 754]

422. M. Matsui, "The first experimental cryptanalysis of the Data Encryption Standard." in *Proceedings of Crypto'94* (Y. Desmedt, ed.), no. 839 in Lecture Notes in Computer Science, pp. 1–11, Springer-Verlag, 1994. [p. 9]

423. M. Matsui, "On correlation between the order of S-boxes and the strength of DES." in *Proceedings of Eurocrypt'94* (A. De Santis, ed.), no. 950 in Lecture Notes in Computer Science, pp. 366–375, Springer-Verlag, 1995. [p. 755]

424. M. Matsui, "Block encryption MISTY." *ISEC96-11*, 1996. [p. 501]

425. M. Matsui, "Block encryption algorithm MISTY." in *Proceedings of Fast Software Encryption – FSE'97* (E. Biham, ed.), no. 1267 in Lecture Notes in Computer Science, pp. 64–74, Springer-Verlag, 1997. [p. 501, 740, 742]
426. M. Matsui, "Specification of MISTY1 - a 64-bit block cipher." Primitive submitted to NESSIE by E. Takeda, Mitsubishi, Sept. 2000. [p. 37, 52, 93, 357, 502, 740]
427. M. Matsui and A. Yamagishi, "A new method for known plaintext attack of FEAL cipher." in *Proceedings of Eurocrypt'92* (R. A. Rueppel, ed.), no. 658 in Lecture Notes in Computer Science, pp. 81–91, Springer-Verlag, 1992. [p. 64]
428. T. Matsumoto and H. Imai, "Public quadratic polynominal-tuples for efficient signature-verification and message-encryption." in *Proceedings of Eurocrypt'88* (C. Günther, ed.), vol. 330 of *Lecture Notes in Computer Science*, pp. 419–453, Springer-Verlag, 1988. [p. 669]
429. U. M. Maurer, "A universal statistical test for random bit generators." in *Proceedings of Crypto'90* (A. Menezes and S. A. Vanstone, eds.), no. 537 in Lecture Notes in Computer Science, pp. 409–420, Springer-Verlag, 1990. [p. 9]
430. U. M. Maurer, "Fast generation of prime numbers and secure public-key cryptographic parameters." *Journal of Cryptology*, vol. 8, no. 3, 1995. [p. 695, 697]
431. U. M. Maurer and Y. Yacobi, "Non-interactive public-key cryptography." in *Proceedings of Eurocrypt'91* (D. W. Davies, ed.), no. 547 in Lecture Notes in Computer Science, pp. 498–507, Springer-Verlag, 1991. [p. 818]
432. U. M. Maurer and Y. Yacobi, "A non-interactive public-key distribution system." *Designs, Codes, and Cryptography*, vol. 9, no. 3, pp. 305–316, 1996. Also available from `http://www.crypto.ethz.ch/~maurer/publications.html`, final version of [431]. [p. 296]
433. L. May, M. Henricksen, W. Millan, G. Carter, and E. Dawson, "Strengthening the key schedule of the AES." in *Proceedings of ACISP'02* (L. M. Batten and J. Seberry, eds.), no. 2384 in Lecture Notes in Computer Science, pp. 226–240, Springer-Verlag, 2002. [p. 112, 114]
434. L. McBride, "Q." Primitive submitted to NESSIE by Mack One Software, Sept. 2000. [p. 37, 52, 134, 135, 357, 749, 750, 752, 755, 760]
435. D. McGrew and S. R. Fluhrer, "LEVIATHAN." Primitive submitted to NESSIE by Cisco Systems, Inc., Sept. 2000. [p. 38, 53, 358]
436. M. McLoone and J. V. McCanny, "High performance single-chip FPGA rijndael algorithm implementations." in *Proceedings of CHES'01* (Çetin Kaya Koç, D. Naccache, and C. Paar, eds.), no. 2162 in Lecture Notes in Computer Science, pp. 65–76, Springer-Verlag, 2001.
437. W. Meier, "On the security of the IDEA block cipher." in *Proceedings of Eurocrypt'93* (T. Helleseth, ed.), no. 765 in Lecture Notes in Computer Science, pp. 371–385, Springer-Verlag, 1993. [p. 82, 84, 138]
438. W. Meier and O. Staffelbach, "Fast correlation attacks on certain stream ciphers." *Journal of Cryptology*, vol. 1, pp. 159–176, 1989. [p. 153]
439. A. J. Menezes, T. Okamoto, and S. A. Vanstone, "Reducing elliptic curve logarithms to logarithms in a finite field." *IEEE Transactions on Information Theory*, vol. IT-39, pp. 1639–1646, 1993. [p. 221, 268]
440. A. J. Menezes and N. P. Smart, "Security of signature schemes in a multi-user setting." Technical report CORR 2001-63, Department of C&O, University of Waterloo, 2001. Available from `http://www.cacr.math.uwaterloo.ca/~ajmeneze/research.html`. [p. 263]
441. A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography*. CRC Press, 1997. Available online at `http://www.cacr.math.uwaterloo.ca/hac/`. [p. 5, 51, 102, 140, 696, 697]
442. R. C. Merkle and M. E. Hellman, "On the security of multiple encryption." *Communications of the ACM*, vol. 24, 1981. [p. 102, 140]

443. T. S. Messerges, E. A. Dabbish, and R. H. Sloan, “Power analysis attacks of modular exponentiation in smartcards.” in *Proceedings of CHES’99* (Çetin Kaya Koç and C. Paar, eds.), no. 1717 in Lecture Notes in Computer Science, pp. 144–157, Springer-Verlag, 2000. [p. 338]
444. S. Micali and L. Reyzin, “Signing with partially adversarial hashing.” Technical report LCS-TM-575, MIT, 1998. Available at `http://www.cs.bu.edu/~reyzin/adv-hashing.html`. [p. 264, 304, 312]
445. M. Michels, D. Naccache, and H. Petersen, “GOST 34.10. A brief overview of Russia’s DSA.” *Computers and Security*, vol. 8, no. 15, pp. 725–732, 1996. Also available at `http://www.geocities.com/CapeCanaveral/Lab/8967/TR-95-14.zip`. [p. 282, 289]
446. V. S. Miller, “Uses of elliptic curves in cryptography.” in *Proceedings of Crypto’85* (H. C. Williams, ed.), no. 218 in Lecture Notes in Computer Science, pp. 417–426, Springer-Verlag, 1986. [p. 633, 645]
447. A. Miyaji, “A message recovery signature scheme equivalent to DSA over elliptic curves.” in *Proceedings of Asiacrypt’96* (K. Kim and T. Matsumoto, eds.), no. 1163 in Lecture Notes in Computer Science, pp. 1–14, Springer-Verlag, 1996. [p. 282]
448. V. Moen, “Integral cryptanalysis of block ciphers.” Master’s thesis, University of Bergen, May 2002. [p. 94, 502]
449. T. Moh, “On the Courtois-Pieprzyk’s attack on Rijndael.” University of San Diego Web-Site, Sept. 2002. Available from `http://www.usdsi.com/aes.html`. [p. 70, 112]
450. L. Monier, “Evaluation and comparison of two efficient probabilistic primality testing algorithms.” *Theoretical Computer Science*, vol. 12, pp. 97–108, 1980. [p. 697]
451. S. Murphy, “An analysis of SAFER.” *Journal of Cryptology*, vol. 11, no. 4, pp. 235–251, 1998. [p. 98, 116]
452. S. Murphy and M. J. B. Robshaw, “Essential algebraic structure within the AES.” in *Proceedings of Crypto’02* (M. Yung, ed.), no. 2442 in Lecture Notes in Computer Science, pp. 1–16, Springer-Verlag, 2002. `NES/DOC/RHU/WP5/022`. Also in *Proceedings of the Third NESSIE Workshop*, 2002. [p. 70, 85, 89, 107, 112, 115, 528, 802]
453. S. Murphy and M. J. B. Robshaw, “Key-dependent S-boxes and differential cryptanalysis.” *Designs, Codes, and Cryptography*, vol. 27, pp. 229–255, 2002. [p. 67, 77]
454. S. Murphy and M. J. B. Robshaw, “Comments on the security of the AES and the XSL technique.” *Electronics Letters*, vol. 39, no. 1, pp. 36–38, 2003. `NES/DOC/RHU/WP5/026`. [p. 70, 89, 107, 112, 115, 528]
455. D. Naccache and J. Stern, “Signing on a postcard.” in *Proceedings of Financial Cryptography – FC’00* (Y. Frankel, ed.), no. 1962 in Lecture Notes in Computer Science, pp. 121–135, Springer-Verlag, 2000. Also available at `http://grouper.ieee.org/groups/1363/Research/contributions/Postcard.ps`. [p. 282, 290]
456. J. Nakahara, Jr, “Extension of Knudsen’s attack on SAFER K-64: L.R.Knudsen, ”A detailed analysis of SAFER K”, J.of Cryptology, vol.13, no.4, 2000,pp.417-436.” Private Communication. [p. 99, 117–119, 144]
457. J. Nakahara, Jr, P. S. L. M. Barreto, B. Preneel, J. Vandewalle, and H. Y. Kim, “SQUARE attacks on reduced-round PES and IDEA block ciphers.” in *Proceedings of the Second NESSIE Workshop*, 2001. `NES/DOC/KUL/WP5/017`. [p. 83, 84, 136, 138]
458. J. Nakahara, Jr, B. Preneel, and J. Vandewalle, “Linear cryptanalysis of reduced-round versions of the SAFER block cipher family.” in *Proceedings of Fast Software Encryption – FSE’00* (B. Schneier, ed.), no. 1978 in Lecture Notes in Computer Science, pp. 244–261, Springer-Verlag, 2000. [p. 99, 117, 119]

459. J. Nakahara, Jr, B. Preneel, and J. Vandewalle, “Linear cryptanalysis of reduced-round SAFER++.” in *Proceedings of the Second NESSIE Workshop*, 2001. [p. 99, 100, 117–119, 140, 144]
460. J. Nakahara, Jr, B. Preneel, and J. Vandewalle, “Impossible differential attacks on reduced-round SAFER ciphers.” Public report, NESSIE, 2002. `NES/DOC/KUL/WP5/030`. [p. 118, 119, 144]
461. J. Nakajima and M. Matsui, “Performance analysis and parallel implementation of dedicated hash functions.” in *Proceedings of Eurocrypt’02* (L. R. Knudsen, ed.), no. 2332 in Lecture Notes in Computer Science, pp. 165–180, Springer-Verlag, 2002. [p. 385]
462. National Institute of Standards and Technology, “FIPS-113: Computer Data Authentication.” May 1985. Available at `http://csrc.nist.gov/publications/fips/`. [p. 201, 208]
463. National Institute of Standards and Technology, “FIPS-180: Secure Hash Standard (SHS).” 1993. [p. 573]
464. National Institute of Standards and Technology, “FIPS-186: Digital Signature Standard (DSS).” Nov. 1994. Available at `http://csrc.nist.gov/publications/fips/`. [p. 282, 289, 302]
465. National Institute of Standards and Technology, “FIPS-180-1: Secure Hash Standard (SHS).” Apr. 1995. Available at `http://csrc.nist.gov/publications/fips/`. [p. 15, 573]
466. National Institute of Standards and Technology, “FIPS-186-1: Digital Signature Standard (DSS).” Dec. 1998. Available at `http://csrc.nist.gov/publications/fips/`. [p. 302]
467. National Institute of Standards and Technology, “FIPS-186-2: Digital Signature Standard (DSS).” Jan. 2000. Available at `http://csrc.nist.gov/publications/fips/`. [p. 302]
468. National Institute of Standards and Technology, “New secure hash algorithms.” 2000. [p. 124]
469. National Institute of Standards and Technology, “A statistical test suite for random and pseudorandom number generators for cryptographic applications.” NIST Special Publication 800-22, Dec. 2000. [p. 9]
470. National Institute of Standards and Technology, “FIPS-197: Advanced Encryption Standard.” Nov. 2001. Available at `http://csrc.nist.gov/publications/fips/`, see also [630]. [p. 15, 22, 79, 112, 519, 520, 523, 525, 749, 750]
471. National Institute of Standards and Technology, “DRAFT Recommendation for Block Cipher Modes of Operation: the RMAC Authentication Mode..” NIST Special Publication 800-38B, 4 Nov. 2002. [p. 208, 209]
472. National Institute of Standards and Technology, “FIPS-180-2: Secure Hash Standard (SHS).” Aug. 2002. Available at `http://csrc.nist.gov/publications/fips/`. [p. 15, 16, 22, 124, 180, 185, 189, 191, 192, 555, 556, 573]
473. National Institute of Standards and Technology, “FIPS-198: The Keyed-Hash Message Authentication Code (HMAC).” Mar. 2002. Available at `http://csrc.nist.gov/publications/fips/`, based on [44]. [p. 201, 209, 623]
474. V. I. Nechaev, “Complexity of a determinate algorithm for the discrete logarithm.” *Mathematical Notes*, vol. 55, no. 2, pp. 165–172, 1994. [p. 222, 268, 272]
475. NESSIE consortium, “Call for cryptographic primitives.” NESSIE, Feb. 2000. `NES/DOC/KUL/WP1/001`. [p. 49, 50, 214, 269, 469]
476. NESSIE consortium, “NESSIE Phase I: Selection of primitives.” Public report, NESSIE, 2001. `NES/DOC/RHU/WP3/017`. [p. 15, 49]
477. NESSIE consortium, “Performance evaluation of NESSIE First Phase.” Deliverable report D14, NESSIE, 2001. `NES/DOC/UCL/WP4/D14`. [p. 14, 49, 361, 364, 366, 368–370, 470]
478. NESSIE consortium, “Security evaluation of NESSIE First Phase.” Deliverable report D13, NESSIE, 2001. `NES/DOC/RHU/WP3/D13`. [p. 11, 49, 134, 470]

479. NESSIE consortium, "Toolbox version 1." Deliverable report D9, NESSIE, 2001. NES/DOC/SAG/WP2/0D9. [p. 72, 73, 154]

480. NESSIE consortium, "NESSIE Performance report." Deliverable report D21, NESSIE, 2002. NES/DOC/TEC/WP6/D21. Available from http://www.cryptonessie.org/. [p. 475]

481. NESSIE consortium, "NESSIE Security report." Deliverable report D20, NESSIE, 2002. NES/DOC/ENS/WP5/D20. Available from http://www.cryptonessie.org/. [p. 475, 497]

482. P. Q. Nguyen and I. Shparlinsky, "The insecurity of the digital signature algorithm with partially known nonces." *Journal of Cryptology*, vol. 15, pp. 151–176, 2002. Also available at ftp://ftp.ens.fr/pub/dmi/users/pnguyen/PubDSA.ps.gz. [p. 304]

483. P. Q. Nguyen and I. Shparlinsky, "The insecurity of the elliptic curve digital signature algorithm with partially known nonces." *Design, Codes and Cryptography*, 2002. Also available at ftp://ftp.ens.fr/pub/dmi/users/pnguyen/PubECDSA.ps.gz. [p. 304]

484. M. Nishioka, H. Satoh, and K. Sakurai, "Design and analysis of fast provably secure public-key cryptosystems based on modular squaring." in *Proceedings of ICISC'01* (K. Kim, ed.), no. 2288 in Lecture Notes in Computer Science, pp. 81–102, Springer-Verlag, 2001. [p. 244]

485. NTT Information Sharing Platform Laboratories, "EPOC-2 specifications." Technical report, NTT Corporation, 2001. [p. 241]

486. NTT Information Sharing Platform Laboratories, "PSEC-KEM specifications." Technical report, NTT Corporation, 2001. [p. 244]

487. K. Nyberg, "Linear approximations of block ciphers." in *Proceedings of Eurocrypt'94* (A. De Santis, ed.), no. 950 in Lecture Notes in Computer Science, pp. 439–444, Springer-Verlag, 1995. [p. 65, 749, 755]

488. K. Nyberg and L. R. Knudsen, "Provable security against a differential attack." *Journal of Cryptology*, vol. 8, no. 1, pp. 27–38, 1995. [p. 64]

489. K. Nyberg and R. A. Rueppel, "A new signature scheme based on the dsa giving message recovery." in *Proceedings of Conference on Computer and Communications Security – CCS'93*, pp. 58–61, ACM Press, Nov. 1993. [p. 282, 290]

490. K. Nyberg and R. A. Rueppel, "Message recovery for signature schemes based on the discrete logarithm problem." in *Proceedings of Eurocrypt'94* (A. De Santis, ed.), no. 950 in Lecture Notes in Computer Science, pp. 182–193, Springer-Verlag, 1995. [p. 282, 290]

491. A. M. Odlyzko, "The future of integer factorization." *Cryptobytes*, vol. 1, no. 2, Summer 1995. Available at http://www.rsasecurity.com/rsalabs/cryptobytes/. [p. 268]

492. A. M. Odlyzko, "Discrete logarithms: The past and the future." *Designs, Codes, and Cryptography*, vol. 19, pp. 129–145, 2000. Also available at http://www.research.att.com/~amo/doc/discrete.logs.future.ps. [p. 221, 268]

493. K. Ohkuma, F. Sano, H. Muratani, M. Motoyama, and S. Kawamura, "Hierocrypt." Primitive submitted to NESSIE by Toshiba Corp., Sept. 2000. See also [494]. [p. 37, 52, 126, 127, 132, 133, 147, 148, 357, 390]

494. K. Ohkuma, H. Shimizu, F. Sano, and S. Kawamura, "The block cipher Hierocrypt." in *Proceedings of Selected Areas in Cryptography – SAC'00* (D. R. Stinson and S. E. Tavares, eds.), no. 2012 in Lecture Notes in Computer Science, pp. 72–88, Springer-Verlag, 2001. [p. 821]

495. K. Ohkuma, H. Shimizu, F. Sano, and S. Kawamura, "Security assessment of Hierocrypt and Rijndael against the differential and linear cryptanalysis (extended abstract)." in *Proceedings of the Second NESSIE Workshop*, 2001. [p. 112–114, 127, 133]

496. T. Okamoto, "ESIGN-D specification." Technical report, NESSIE external documents, Aug. 2002. Available from `http://www.cryptonessie.org/tweaks.html`. [p. 305]
497. T. Okamoto, "Security of ESIGN-D signature scheme." Technical report, NESSIE external documents, Aug. 2002. Available from `http://www.cryptonessie.org/tweaks.html`. [p. 306]
498. T. Okamoto and D. Pointcheval, "The gap problems: A new class of problems for the security of cryptographic schemes." in *Proceedings of Public Key Cryptography – PKC'01* (K. Kim, ed.), no. 1992 in Lecture Notes in Computer Science, pp. 104–118, Springer-Verlag, 2001. [p. 220]
499. T. Okamoto and D. Pointcheval, "REACT: Rapid Enhanced-security Asymmetric Cryptosystem Transform." in *Proceedings of CT-RSA'01* (D. Naccache, ed.), no. 2020 in Lecture Notes in Computer Science, pp. 159–175, Springer-Verlag, 2001. [p. 249, 251]
500. T. Okamoto and D. Pointcheval, "RSA-REACT: An alternative to RSA-OAEP." in *Proceedings of the Second NESSIE Workshop*, 2001. Available at `http://www.di.ens.fr/~pointche/pub.php?reference=OkPo01c`. [p. 473]
501. T. Okamoto and S. Uchiyama, "A new public-key cryptosystem as secure as factoring." in *Proceedings of Eurocrypt'98* (K. Nyberg, ed.), no. 1403 in Lecture Notes in Computer Science, pp. 308–318, Springer-Verlag, 1998. [p. 243, 249, 251]
502. K. Okeya, H. Kurumatani, and K. Sakurai, "OK-ECDSA.". Submission to CRYPTREC, available at `http://www.sdl.hitachi.co.jp/crypto/ok-ecdsa/`. [p. 303]
503. K. Okeya, H. Kurumatani, and K. Sakurai, "Elliptic curves with the Montgomery-form and their cryptographic applications." in *Proceedings of Public Key Cryptography – PKC'00* (H. Imai and Y. Zheng, eds.), no. 1751 in Lecture Notes in Computer Science, pp. 238–257, Springer-Verlag, 2000. Also available at `http://www.sdl.hitachi.co.jp/crypto/ok-ecdsa/`. [p. 303]
504. G. Orlando and C. Paar, "A high-performance reconfigurable elliptic curve processor for $GF(2^m)$." in *Proceedings of CHES'00* (Çetin Kaya Koç and C. Paar, eds.), no. 1965 in Lecture Notes in Computer Science, pp. 41–56, Springer-Verlag, 2000. [p. 392, 393]
505. G. Orlando and C. Paar, "A scalable $GF(p)$ elliptic curve processor architecture for programmable hardware." in *Proceedings of CHES'01* (Çetin Kaya Koç, D. Naccache, and C. Paar, eds.), no. 2162 in Lecture Notes in Computer Science, pp. 356–371, Springer-Verlag, 2001. [p. 393]
506. S. B. Örs, L. Batina, and B. Preneel, "Hardware implementation of elliptic curve processor over $GF(p)$." Public report, NESSIE, 2002. `NES/DOC/KUL/WP5/023`. [p. 389, 392]
507. E. Oswald, "Enhancing simple power-analysis attacks on elliptic curve cryptosystems." in *Proceedings of CHES'02* (B. S. Kaliski, Çetin Kaya Koç, and C. Paar, eds.), no. 2535 in Lecture Notes in Computer Science, Springer-Verlag, 2002. [p. 340]
508. E. Oswald and B. Preneel, "A survey on passive side-channel attacks and their countermeasures for the NESSIE public-key cryptosystems." Public report, NESSIE, 2002. `NES/DOC/KUL/WP5/027`. [p. 224, 305, 340, 342, 666]
509. E. Oswald and B. Preneel, "A theoretical evaluation of some NESSIE candidates regarding their susceptibility towards power analysis attacks." Public report, NESSIE, 2002. `NES/DOC/KUL/WP5/022`. [p. 71, 340–342]
510. P. Pailler, "Public-key cryptosystems based on composite degree residuosity classes." in *Proceedings of Eurocrypt'99* (J. Stern, ed.), no. 1592 in Lecture Notes in Computer Science, pp. 223–238, Springer-Verlag, 1999. [p. 788]
511. M. G. Parker, "Generalised S-Box nonlinearity." Public report, NESSIE, June 2002. `NES/DOC/UIB/WP5/020`. [p. 87, 94]

512. J. Patarin, "Cryptoanalysis of the Matsumoto and Imai public key scheme of eurocrypt'88." in *Proceedings of Crypto'95* (D. Coppersmith, ed.), no. 963 in Lecture Notes in Computer Science, pp. 248–261, Springer-Verlag, 1995. [p. 669]

513. J. Patarin, N. T. Courtois, and L. Goubin, "FLASH: a fast multivariate signature algorithm." in *Proceedings of CT-RSA'01* (D. Naccache, ed.), no. 2020 in Lecture Notes in Computer Science, pp. 298–307, Springer-Verlag, 2001. [p. 314]

514. J. Patarin *et al.*, "Flash and Sflash." Primitives submitted to NESSIE, Sept. 2000. [p. 39, 53, 277, 358, 669]

515. J. Patarin, L. Goubin, and N. T. Courtois, "C^{*+-} and HM: variations around two schemes of T. Matsumoto and H. Imai." in *Proceedings of Asiacrypt'98* (K. Ohta and D. Pei, eds.), no. 1514 in Lecture Notes in Computer Science, pp. 35–49, Springer-Verlag, 1998. [p. 307, 308, 669, 670]

516. J. Patarin, L. Goubin, and N. T. Courtois, "Improved algorithms for isomorphisms of polynomials." in *Proceedings of Eurocrypt'98* (K. Nyberg, ed.), no. 1403 in Lecture Notes in Computer Science, pp. 184–200, Springer-Verlag, 1998. [p. 71]

517. R. Peralta and E. Okamoto, "Faster factoring of integers of a special form." *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Science*, vol. 4, no. E79-A, 1996. [p. 221, 268]

518. E. Petrank and C. Rackoff, "CBC MAC for real-time data sources." *Journal of Cryptology*, vol. 13, no. 3, pp. 315–338, 2000. [p. 199, 208, 209, 617, 618]

519. L. A. Pintsov and S. A. Vanstone, "Postal revenue collection in the digital age." in *Proceedings of Financial Cryptography – FC'00* (Y. Frankel, ed.), no. 1962 in Lecture Notes in Computer Science, pp. 105–120, Springer-Verlag, 2000. Also available at `http://www.cacr.math.uwaterloo.ca/techreports/2000/corr2000-43.ps`. [p. 282, 290]

520. G. Piret and J.-J. Quisquater, "Integral cryptanalysis on reduced-round SAFER++." Public report, NESSIE, 2003. `NES/DOC/UCL/WP5/002`. [p. 118, 144]

521. H. C. Pocklington, "The determination of the prime or composite nature of large numbers by fermat's theorem." *Proceedings of the Cambridge Philosophical Society*, vol. 18, pp. 29–30, 1914. [p. 697]

522. D. Pointcheval, "The composite discrete logarithm and secure authentication." in *Proceedings of Public Key Cryptography – PKC'00* (H. Imai and Y. Zheng, eds.), no. 1751 in Lecture Notes in Computer Science, pp. 113–128, Springer-Verlag, 2000. [p. 333]

523. D. Pointcheval and J. Stern, "Security arguments for digital signatures and blind signatures." *Journal of Cryptology*, vol. 13, no. 3, pp. 361–396, 2000. Also available at `http://www.di.ens.fr/~pointche/pub.php?reference=PoSt00`. [p. 271, 290, 291]

524. J. M. Pollard, "Monte Carlo methods for index computation mod p." *Mathematics of Computation*, vol. 32, pp. 918–924, 1978. [p. 221, 268]

525. G. Poupard *et al.*, "GPS." Primitive submitted to NESSIE, Sept. 2000. [p. 39, 53, 358]

526. G. Poupard and J. Stern, "Security analysis of a practical "on the fly" authentication and signature generation." in *Proceedings of Eurocrypt'98* (K. Nyberg, ed.), no. 1403 in Lecture Notes in Computer Science, pp. 422–436, Springer-Verlag, 1998. [p. 332, 777, 778, 780]

527. G. Poupard and J. Stern, "On the fly signatures based on factoring." in *Proceedings of Conference on Computer and Communications Security – CCS'99*, pp. 37–45, ACM Press, 1999. [p. 333, 788]

528. G. Poupard and J. Stern, "Fair encryption of rsa keys." in *Proceedings of Eurocrypt'00* (B. Preneel, ed.), no. 1807 in Lecture Notes in Computer Science, pp. 172–189, Springer-Verlag, 2000. [p. 788]

529. G. Poupard and J. Stern, "Short proofs of knowledge for factoring." in *Proceedings of Public Key Cryptography – PKC'00* (H. Imai and Y. Zheng, eds.),

no. 1751 in Lecture Notes in Computer Science, pp. 147–166, Springer-Verlag, 2000. [p. 784, 788]

530. B. Preneel, “Cryptographic primitives for information authentication — state of the art.” in *State of the Art in Applied Cryptography* (B. Preneel and V. Rijmen, eds.), no. 1528 in Lecture Notes in Computer Science, pp. 50–105, Springer-Verlag, 1998. [p. 171, 172, 174, 195, 197]
531. B. Preneel, A. Bosselaers, and H. Dobbertin, “The cryptographic hash function ripemd-160.” *Cryptobytes*, vol. 3, no. 2, pp. 9–14, 1997. [p. 204, 605]
532. B. Preneel and P. C. van Oorschot, “MDx-MAC and building fast MACs from hash functions.” in *Proceedings of Crypto’95* (D. Coppersmith, ed.), no. 963 in Lecture Notes in Computer Science, pp. 1–14, Springer-Verlag, 1995. [p. 197, 198, 200]
533. B. Preneel and P. C. van Oorschot, “On the security of two MAC algorithms.” in *Proceedings of Eurocrypt’96* (U. Maurer, ed.), no. 1070 in Lecture Notes in Computer Science, pp. 19–32, Springer-Verlag, 1996. [p. 198]
534. S. Pyka, “Status report of SOBER-t16 cryptanalysis.”. Private communication. [p. 163]
535. J.-J. Quisquater and D. Samyde, “ElectroMagnetic Analysis (EMA): Measures and counter-measures for smart cards.” in *Proceedings of E-smart 2001* (I. Attali and T. P. Jensen, eds.), no. 2140 in Lecture Notes in Computer Science, pp. 200–210, Springer-Verlag, 2001. [p. 338]
536. C. Rackoff and D. R. Simon, “Noninteractive zero-knowledge proof of knowledge and chosen ciphertext attack.” in *Proceedings of Crypto’91* (J. Feigenbaum, ed.), no. 576 in Lecture Notes in Computer Science, pp. 433–444, Springer-Verlag, 1991. [p. 216]
537. H. Raddum, “The statistical evaluation of the NESSIE submission IDEA.” Public report, NESSIE, Sept. 2001. `NES/DOC/UIB/WP3/012`. [p. 84]
538. H. Raddum, “Cryptanalysis of IDEA-X/2.” in *Proceedings of Fast Software Encryption – FSE’03* (T. Johansson, ed.), Lecture Notes in Computer Science, Springer-Verlag, 2003. `NES/DOC/UIB/WP5/023`. [p. 83]
539. H. Raddum and L. R. Knudsen, “A differential attack on reduced-round SC2000.” in *Proceedings of the Second NESSIE Workshop*, 2001. `NES/DOC/UIB/WP3/008`. [p. 136, 148]
540. H. Riesel., *Prime Numbers and Computer Methods for Factorization*. Birkhäuser, 1985. [p. 697]
541. V. Rijmen, B. Preneel, and E. De Win, “On weaknesses of non-surjective round functions.” *Designs, Codes, and Cryptography*, vol. 12, no. 3, pp. 253–266, 1997. [p. 68]
542. R. L. Rivest, “The RC4 enycryption algorithm.” RSA Security Inc., Mar. 1992. [p. 157, 170]
543. R. L. Rivest, A. Shamir, and L. M. Adleman, “A method for obtaining digital signatures and public-key cryptosystems.” *Communications of the ACM*, vol. 21, no. 2, pp. 120–126, Feb. 1978. [p. 213, 216, 248, 257, 273, 639, 657, 696]
544. P. Rogaway and D. Coppersmith, “A software-oriented encryption algorithm.” in *Proceedings of Fast Software Encryption – FSE’94* (B. Preneel, ed.), no. 1008 in Lecture Notes in Computer Science, pp. 56–63, Springer-Verlag, 1995. [p. 157]
545. M. Rogawski, “Analysis of implementation of Hierocrypt-3 algorithm (and its comparison to Camellia algorithm) using ALTERA devices.” Available at `http://eprint.iacr.org/2003/258/`, June 2003. [p. 390]
546. T. Rosa, “On key-collisions in (EC)DSA schemes.” Available at `http://eprint.iacr.org/2002/129/`, 2002. [p. 263]
547. G. Rouvroy, F.-X. Standaert, J.-J. Quisquater, and J.-D. Legat, “Efficient FPGA implementation of block cipher MISTY1.” Internal report, NESSIE, Sept. 2002. `NES/DOC/UCL/WP6/004`. Included in [592]. also presented in the poster session at RAW 2003. [p. 389]

548. RSA Laboratories, "Public-Key Cryptography Standards.". Available from `http://www.rsasecurity.com/rsalabs/pkcs/`. [p. 224]

549. RSA Labs, "PKCS #1 - RSA Cryptography Standard." June 2002. Available at `http://www.rsasecurity.com/rsalabs/pkcs/pkcs-1/`. [p. 230]

550. H.-G. Rück, "On the discrete logarithm in the divisor class group of curve." *Mathematics of Computation*, vol. 68, no. 226, pp. 805–806, 1999. [p. 221, 268]

551. R. A. Rueppel, *Analysis and Design of stream ciphers*. Springer-Verlag, 1986. [p. 151]

552. M.-J. O. Saarinen, "A time-memory trade-off attack against LILI-128." in *Proceedings of Fast Software Encryption – FSE'02* (J. Daemen and V. Rijmen, eds.), no. 2365 in Lecture Notes in Computer Science, pp. 231–236, Springer-Verlag, 2002. [p. 169]

553. M.-J. O. Saarinen, "Cryptanalysis of block ciphers based on SHA-1 and MD5." in *Proceedings of Fast Software Encryption – FSE'03* (T. Johansson, ed.), Lecture Notes in Computer Science, Springer-Verlag, 2003. Also available at `http://www.tcs.hut.fi/~mjos/shaan.ps`. [p. 124, 125, 188, 189, 555, 573]

554. T. Satoh and K. Araki, "Fermat quotients and the polynomial time discrete algorithm for anomalous elliptic curves." *Commentarii Math. Univ. Sancti Pauli*, vol. 47, no. 1, pp. 81–92, 1998. Errata in [555]. [p. 221, 268]

555. T. Satoh and K. Araki, "Errata to the paper: Fermat quotients and the polynomial time discrete algorithm for anomalous elliptic curves." *Commentarii Math. Univ. Sancti Pauli*, vol. 48, pp. 211–213, 1999. Summary available at `http://www.rimath.saitama-u.ac.jp/lab.en/TkkzSatoh/A1998.html`. [p. 825]

556. W. Schindler, "A combined timing and power attack." in *Proceedings of Public Key Cryptography – PKC'02* (D. Naccache and P. Paillier, eds.), no. 2274 in Lecture Notes in Computer Science, pp. 263–279, Springer-Verlag, 2002. [p. 339]

557. B. Schneier, "Bernstein's factoring breakthrough?." Crypto-Gram, Mar. 2002. Available from `http://www.counterpane.com/crypto-gram-0203.html`. [p. 269]

558. B. Schneier, J. Kelsey, D. Whiting, DavidWagner, C. Hall, and N. Ferguson, "Performance comparision of the AES submissions." in *Proceedings of the Second Advanced Encryption Standard Conference*, pp. 15–34, NIST, 1999. [p. 385]

559. C. P. Schnorr, "Efficient identification and signatures for smart cards." in *Proceedings of Crypto'89* (G. Brassard, ed.), no. 435 in Lecture Notes in Computer Science, pp. 239–252, Springer-Verlag, 1990. [p. 282, 289]

560. C. P. Schnorr, "Efficient signature generation by smart cards." *Journal of Cryptology*, vol. 4, no. 3, pp. 161–174, 1991. [p. 282, 330, 781, 785]

561. C. P. Schnorr and S. Vaudenay, "Black box cryptanalysis of hash networks based on multipermutations." in *Proceedings of Eurocrypt'94* (A. De Santis, ed.), no. 950 in Lecture Notes in Computer Science, pp. 47–57, Springer-Verlag, 1995. [p. 126]

562. K. Schramm, "DES sidechannel collision attacks on smartcard implementations." Master's thesis, Ruhr Universität Bochum, Aug. 2002. [p. 339]

563. T. Schweinberger and V. Shoup, "ACE: The advanced cryptographic engine." Primitive submitted to NESSIE, Sept. 2000. [p. 38, 39, 53, 358, 645]

564. I. A. Semaev, "Evaluation of discrete logarithms in a group of p-torsion points of an elliptic curve in characteristic p." *Mathematics of Computation*, vol. 67, no. 221, pp. 353–356, 1998. [p. 221, 268]

565. P. Serf, "The degrees of completeness, of avalanche effect, and of strict avalanche criterion for Mars, RC6, Rijndael, Serpent, and Twofish with reduced number of rounds." Public report, NESSIE, 2000. `NES/DOC/SAG/WP3/003`. [p. 73]

566. A. Shamir and A. Kipnis, "Cryptanalysis of the HFE public key cryptosystem." in *Proceedings of Crypto'99* (M. J. Wiener, ed.), no. 1666 in Lecture Notes in Computer Science, pp. 19–30, Springer-Verlag, 1999. [p. 70, 268]

567. C. E. Shannon, "Communication theory of secrecy systems." *Bell System Technical Journal*, vol. 28, 1949. [p. 55, 57]

568. T. Shimoyama, M. Takenaka, and T. Koshiba, "Multiple linear cryptanalysis of a reduced round RC6." in *Proceedings of Fast Software Encryption – FSE'02* (J. Daemen and V. Rijmen, eds.), no. 2365 in Lecture Notes in Computer Science, pp. 76–88, Springer-Verlag, 2002. [p. 108, 110, 143]

569. T. Shimoyama, H. Yanami, K. Yokoyama, M. Takenaka, K. Itoh, J. Yajima, N. Torii, and H. Tanaka, "SC2000." Primitive submitted to NESSIE by N. Torii, Fujitsu Laboratories LTD, Sept. 2000. Based on [570]. [p. 38, 53, 135, 136, 148, 357]

570. T. Shimoyama, H. Yanami, K. Yokoyama, M. Takenaka, K. Itoh, J. Yajima, N. Torii, and H. Tanaka, "The block cipher SC2000." in *Proceedings of Fast Software Encryption – FSE'01* (M. Matsui, ed.), no. 2355 in Lecture Notes in Computer Science, pp. 312–327, Springer-Verlag, 2001. [p. 136, 826]

571. R. Shipsey, "ECIES." Public report, NESSIE, 2001. `NES/DOC/RHU/WP3/007`. [p. 238]

572. R. Shipsey, "GPS." Public report, NESSIE, 2001. `NES/DOC/RHU/WP3/004`. [p. 333]

573. R. Shipsey, "How long...?." Public report, NESSIE, 2001. `NES/DOC/RHU/WP3/015`. [p. 225, 268]

574. R. Shipsey, "PSEC-1-2-3." Public report, NESSIE, 2001. `NES/DOC/RHU/WP3/008`. [p. 246, 254, 255]

575. R. Shipsey, "PSEC-KEM." Public report, NESSIE, 2001. `NES/DOC/RHU/WP5/016`. [p. 246]

576. R. Shipsey, "Selecting a version of PSEC." Public report, NESSIE, 2001. `NES/DOC/RHU/WP3/012`. [p. 243, 246, 250, 252, 254, 255]

577. T. Shirai, "Differential, linear, boomerang and rectangle cryptanalysis of reduced-round Camellia." in *Proceedings of the Third NESSIE Workshop*, 2002. [p. 106, 143, 528]

578. T. Shirai, S. Kanamaru, and G. Abe, "Improved upper bounds of differential and linear characteristic probability for Camellia." in *Proceedings of Fast Software Encryption – FSE'02* (J. Daemen and V. Rijmen, eds.), no. 2365 in Lecture Notes in Computer Science, pp. 128–142, Springer-Verlag, 2002. [p. 106, 528, 715, 716, 718, 720, 723]

579. T. Shirai and K. Shibutani, "On the diffusion matrix employed in the Whirlpool hashing function." Technical report, NESSIE external documents, Mar. 2003. [p. 185, 569]

580. P. W. Shor, "Algorithms for quantum computation: Discrete logarithms and factoring." in *Proceedings of FOCS'94*, pp. 124–134, IEEE, 1994. [p. 222, 268]

581. V. Shoup, "Lower bounds for discrete logarithms and related problems." in *Proceedings of Eurocrypt'97* (W. Fumy, ed.), no. 1233 in Lecture Notes in Computer Science, pp. 256–266, Springer-Verlag, 1997. Revised version available at `http://shoup.net/papers/dlbounds1.pdf`. [p. 220, 222, 268, 272, 293]

582. V. Shoup, "Using hash functions as a hedge against chosen ciphertext attack." in *Proceedings of Eurocrypt'00* (B. Preneel, ed.), no. 1807 in Lecture Notes in Computer Science, pp. 275–288, Springer-Verlag, 2000. Also available at `http://www.shoup.net/papers/hedge.ps`. [p. 236]

583. V. Shoup, "OAEP reconsidered." in *Proceedings of Crypto'01* (J. Kilian, ed.), no. 2139 in Lecture Notes in Computer Science, pp. 239–259, Springer-Verlag, 2001. Also available at `http://www.shoup.net/papers/oaep.pdf`. [p. 16, 257, 473]

584. V. Shoup, "A proposal for an ISO standard for public key encryption (version 2.0)." Available at `http://eprint.iacr.org/2001/112/`, 2001. [p. 16, 17, 39, 53, 224, 225, 230, 233, 234, 238, 239, 246, 247, 283, 284, 358, 473, 633, 634, 639, 645]

585. T. Siegenthaler, "Decrypting a class of stream ciphers using ciphertext only." *IEEE Transactions on Computers*, vol. 34, no. 1, pp. 81–85, 1985. [p. 153]

586. R. D. Silverman, "A cost based security analysis of symmetric and asymmetric key lengths." RSA Bulletin 13, RSA Laboratories, Apr. 2000. Available at `http://www.rsasecurity.com/rsalabs/bulletins/`. [p. 268, 269]
587. N. P. Smart, "The discrete logarithm problem on elliptic curve of trace one." *Journal of Cryptology*, vol. 12, no. 3, pp. 193–196, 1999. [p. 221, 268]
588. N. P. Smart, "The exact security of ECIES in the generic group model." in *Proceedings of Cryptography and Coding – CC'01* (B. Honary, ed.), no. 2260 in Lecture Notes in Computer Science, pp. 73–84, Springer-Verlag, 2001. [p. 239]
589. N. P. Smart, "The hessian form of an elliptic curve." in *Proceedings of CHES'01* (Çetin Kaya Koç, D. Naccache, and C. Paar, eds.), no. 2162 in Lecture Notes in Computer Science, pp. 121–128, Springer-Verlag, 2001. [p. 393]
590. R. Solovay and V. Strassen, "A fast Monte-Carlo test for primality." *SIAM Journal on Computing*, vol. 6, pp. 84–85, 1977. [p. 697]
591. F.-X. Standaert, G. Rouvroy, J.-J. Quisquater, and J.-D. Legat, "Design strategies and modified descriptions to optimize cipher FPGA implementations: Fast and compact results for DES and Triple-DES." Technical report, DICE-UCL, 2002. [p. 389]
592. F.-X. Standaert, G. Rouvroy, J.-J. Quisquater, and J.-D. Legat, "Efficient FPGA implementations of block ciphers KHAZAD and MISTY1." in *Proceedings of the Third NESSIE Workshop*, 2002. [p. 389, 824, 827]
593. F.-X. Standaert, G. Rouvroy, J.-J. Quisquater, and J.-D. Legat, "Optimized FPGA implementation of block cipher KHAZAD." Internal report, NESSIE, Sept. 2002. `NES/DOC/UCL/WP6/004`. Included in [592]. [p. 389]
594. F.-X. Standaert, G. Rouvroy, J.-J. Quisquater, and J.-D. Legat, "A methodology to implement block ciphers in reconfigurable hardware and its application to fast and compact AES rijndael." in *Proceedings of FPGA 2003* (R. Tessier, ed.), 2003. [p. 389]
595. R. Steinwandt, W. Geiselmann, and T. Beth, "A theoretical DPA-based cryptanalysis of the NESSIE candidates FLASH and SFLASH." in *Proceedings of the Second NESSIE Workshop*, 2001. [p. 309, 677]
596. J. Stern, "Almost uniform density of e-th powers on large intervals." 2002. [p. 306]
597. J. Stern, D. Pointcheval, J. Malone-Lee, and N. P. Smart, "Flaws in applying proof methodologies to signature schemes." in *Proceedings of Crypto'02* (M. Yung, ed.), no. 2442 in Lecture Notes in Computer Science, pp. 93–110, Springer-Verlag, 2002. Also available at `http://www.di.ens.fr/~pointche/pub.php?reference=MaPoSmSt02`. [p. 261, 262, 278, 306]
598. D. R. Stinson, *Cryptography: Theory and Practice*. CRC Press, 1995. [p. 51]
599. V. Strassen, "Gaussian elimination is not optimal." *Numerische Mathematik*, vol. 13, pp. 354–356, 1969. [p. 709]
600. M. Sudan, "Decoding of Reed Solomon codes beyond the error-correction bound." *Journal of Complexity*, vol. 13, no. 1, pp. 180–193, 1997. Also available at `http://theory.lcs.mit.edu/~madhu/papers/reeds-journ.ps`. [p. 67]
601. M. Sugita, "Higher order differential attack on block cipher MISTY1, 2." Technical report ISEC98-4, IEICE, May 1998. [p. 139]
602. M. Sugita, K. Kobara, and H. Imai, "Security of reduced version of the block cipher Camellia against truncated and impossible differential cryptanalysis." in *Proceedings of Asiacrypt'01* (C. Boyd, ed.), no. 2248 in Lecture Notes in Computer Science, pp. 193–207, Springer-Verlag, 2001. [p. 106, 143, 528, 715]
603. S. Sutikno, R. Effendi, and A. Surya, "Design and implementation of arithmetic processor $gf(2^{155})$ for elliptic curve cryptosystems." in *Proceedings of the 1998 IEEE Asia-Pacific Conference on Circuits and Systems (APCCAS'98)*, pp. 647–650, 1998. [p. 393]

604. H. Tanaka, K. Hisamatsu, and T. Kaneko, “Strength of MISTY1 without FL function for higher order differential attack.” in *Proceedings of AAECC’99* (M. Fossorier, H. Imai, S. Lin, and A. Poli, eds.), no. 1719 in Lecture Notes in Computer Science, pp. 221–230, Springer-Verlag, 1999. [p. 93, 139, 502]

605. H. Tanaka, C. Ishii, and T. Kaneko, “On the strength of KASUMI without FL functions against higher order differential attack.” in *Proceedings of ICISC’00* (D. Won, ed.), no. 2015 in Lecture Notes in Computer Science, pp. 14–21, Springer-Verlag, 2000. [p. 139]

606. E. van den Bogaert and V. Rijmen, “Differential analysis of SHACAL.” Internal report, NESSIE, May 2001. `NES/DOC/KUL/WP3/009`. [p. 123, 188]

607. P. C. van Oorschot and M. J. Wiener, “Parallel collision search with applications to hash functions and discrete logarithms.” in *Proceedings of Conference on Computer and Communications Security – CCS’94*, pp. 210–218, ACM Press, 1994. [p. 221, 268]

608. P. C. van Oorschot and M. J. Wiener, “Improving implementable meet-in-the-middle attacks by orders of magnitude.” in *Proceedings of Crypto’96* (N. Koblitz, ed.), no. 1109 in Lecture Notes in Computer Science, pp. 229–236, Springer-Verlag, 1996. [p. 102]

609. B. Van Rompay and B. den Boer, “TTMAC.” Primitive submitted to NESSIE, Sept. 2000. [p. 38, 53, 201, 358, 605]

610. S. A. Vanstone, “Responses to NIST’s proposal.” *Communications of the ACM*, vol. 35, pp. 50–52, July 1992. [p. 302, 665]

611. S. Vaudenay, “An experiment on DES - statistical cryptanalysis.” in *Proceedings of Conference on Computer and Communications Security – CCS’95*, pp. 139–147, ACM Press, 1995. [p. 69]

612. S. Vaudenay, “Hidden collisions on DSS.” in *Proceedings of Crypto’96* (N. Koblitz, ed.), no. 1109 in Lecture Notes in Computer Science, pp. 83–88, Springer-Verlag, 1996. [p. 264, 304]

613. S. Vaudenay, “On the security of CS-Cipher.” in *Proceedings of Fast Software Encryption – FSE’99* (L. R. Knudsen, ed.), no. 1636 in Lecture Notes in Computer Science, pp. 260–274, Springer-Verlag, 1999. [p. 126, 754]

614. S. Vaudenay, “Security flaws induced by CBC padding — applications to SSL, IPSEC, WTLS.” in *Proceedings of Eurocrypt’02* (L. R. Knudsen, ed.), no. 2332 in Lecture Notes in Computer Science, pp. 534–546, Springer-Verlag, 2002. [p. 343]

615. S. Vaudenay, “The security of DSA and ECDSA.” in *Proceedings of Public Key Cryptography – PKC’03* (Y. Desmedt, ed.), no. 2567 in Lecture Notes in Computer Science, pp. 309–323, Springer-Verlag, 2003. Also available from `http://lasecwww.epfl.ch/php_code/publications/search.php?ref=Vau03a`. [p. 264, 304]

616. D. Wagner, “The boomerang attack.” in *Proceedings of Fast Software Encryption – FSE’99* (L. R. Knudsen, ed.), no. 1636 in Lecture Notes in Computer Science, pp. 156–170, Springer-Verlag, 1999. [p. 64, 65, 67, 83, 722, 727]

617. C. D. Walter and S. Thompson, “Distinguishing exponent digits by observing modular subtractions.” in *Proceedings of CT-RSA’01* (D. Naccache, ed.), no. 2020 in Lecture Notes in Computer Science, pp. 192–207, Springer-Verlag, 2001. [p. 339]

618. W. Wenling and F. Dengguo, “Linear cryptanalysis of NUSH block cipher.” *Science in China (Series F)*, vol. 45, no. 1, pp. 59–67, 2002. [p. 129, 147, 148]

619. R. Wernsdorf, “IDEA, SAFER++ and their permutation groups.” in *Proceedings of the Second NESSIE Workshop, and update (private communication, 2003)*, 2001. [p. 82, 98, 117]

620. R. Wernsdorf, “The round functions of Rijndael generate the alternating group.” in *Proceedings of Fast Software Encryption – FSE’02* (J. Daemen and V. Rijmen, eds.), no. 2365 in Lecture Notes in Computer Science, pp. 143–148, Springer-Verlag, 2002. [p. 112, 114]

621. J. White, "Initial report on the EPOC asymmetric encryption scheme." Public report, NESSIE, 2001. NES/DOC/RHU/WP3/011. [p. 243, 250, 252]

622. M. J. Wiener, "Cryptanalysis of short RSA secret exponents." in *Proceedings of Eurocrypt'89* (J.-J. Quisquater and J. Vandewalle, eds.), vol. 434 of *Lecture Notes in Computer Science*, p. 372, Springer-Verlag, 1990. [p. 336]

623. M. J. Wiener, "Performance comparison of public key cryptosystems." *Cryptobytes*, vol. 4, no. 1, Summer 1998. Available at http://www.rsasecurity.com/rsalabs/cryptobytes/. [p. 268]

624. M. J. Wiener and R. J. Zuccherato, "Fast attacks on elliptic curve cryptosystems." in *Proceedings of Selected Areas in Cryptography – SAC'98* (S. E. Tavares and H. Meijer, eds.), no. 1556 in Lecture Notes in Computer Science, pp. 190–200, Springer-Verlag, 1999. [p. 221, 268]

625. Xilinx, *Virtex 2.5V Field Programmable Gate Arrays Data Sheet*. URL: http://www.xilinx.com.

626. A. C.-C. Yao, "Theory and applications of trapdoor functions." in *Proceedings of FOCS'82*, pp. 80–91, IEEE, 1982. [p. 770]

627. Y. Yeom, S. Park, and I. Kim, "On the security of Camellia against the square attack." in *Proceedings of Fast Software Encryption – FSE'02* (J. Daemen and V. Rijmen, eds.), no. 2365 in Lecture Notes in Computer Science, pp. 89–99, Springer-Verlag, 2002. [p. 106, 143, 528, 715, 729]

628. A. M. Youssef and G. Gong, "On the interpolation attacks on block ciphers." in *Proceedings of Fast Software Encryption – FSE'00* (B. Schneier, ed.), no. 1978 in Lecture Notes in Computer Science, pp. 109–120, Springer-Verlag, 2000. [p. 67]

629. A. M. Youssef and S. E. Tavares, "On some algebraic structures in the AES round function." Available at http://eprint.iacr.org/2002/144/, 2002. [p. 70, 114]

630. "Advanced encryption algorithm (AES) development effort." 1997–2000. http://csrc.nist.gov/CryptoToolkit/aes/. [p. 5, 820]

631. "CRYPTREC project." 2000–2002. http://www.ipa.go.jp/security/enc/CRYPTREC/. [p. 5, 715]

632. "Modes of operation for symmetric key block ciphers." 2000–2002. http://csrc.nist.gov/CryptoToolkit/modes/. [p. 56]

633. "New European Schemes for Signatures, Integrity, and Encryption." 2000–2003. http://www.cryptonessie.org/. [p. 7, 15, 19, 715, 763]