Schemes and Applications for Binding Hardware and Software in Computing Devices

Robert Philip Lee

Thesis submitted to Royal Holloway, University of London for the degree of Doctor of Philosophy in Information Security.

2018

Declaration

These doctoral studies were conducted under the supervision of Professor Konstantinos Markantonakis.

The work presented in this thesis is the result of original research I conducted, in collaboration with others, whilst enrolled in the School of Mathematics and Information Security as a candidate for the degree of Doctor of Philosophy. This work has not been submitted for any other degree or award in any other university or educational establishment.

Robert Lee

Date

Acknowledgements

Firstly, I would like to thank Prof. Kostas Markantonakis for his support and supervision throughout my PhD. Without his direction, support and ability (if not eagerness) to tell me his honest opinions then I'd have never produced this thesis. Similarly, I must also thank Raja Naeem Akram for his invaluable inputs to my research and for being so welcoming whenever I turned up at his office without warning.

Next, I must thank the Information Security Group, the Centre for Doctoral Training in Cyber Security and the Smart Card and IoT Security Centre. Thank you to the management committee who gave me the opportunity to pursue this PhD and thank you to my fellow students who made the experience so much fun. My thanks go especially to Andreas, Christian, Greg, Jonathan, Pip, Rachel, Sam, Thalia and Thyla who's friendship so blessed my time at RHUL.

To Adam, Rose, Steve, Steve, Steve, Mary, Claire, Sophie, Stuart, Reuben and all at St Liz - thank you for your friendship, company and for providing that essential tether to the "real world".

My family are also due their share of thanks for this thesis. Thank you to all of you but especially to my father, David, and my brother, Stuart, for your support, encouragement and advice.

And finally, a mountain of thanks are due to Ela, who during the course of this PhD has been the most amazing girlfriend, the most amazing fiancée and now the most amazing wife that anyone could ask for. Thank you for your love, for your support and for taking me away from work when I needed the breaks and for encouraging me back to my desk when I was convinced I needed more breaks from my corrections. I couldn't have done this without you, thank you.

Publications

A number of papers from this work have been presented and published at international, peer-reviewed conferences. The published papers include:

- R. P. Lee, K. Markantonakis, and R. N. Akram. Binding hardware and software to prevent firmware modification and device counterfeiting. In J. Zhou and J. López, editors, *Proceedings of the 2nd ACM International Workshop on Cyber-Physical System Security, CPSS@AsiaCCS, Xi'an, China, May 30, 2016*, pages 70–81. ACM, 2016

- C. Shepherd, G. Arfaoui, I. Gurulian, R. P. Lee, K. Markantonakis, R. N. Akram, D. Sauveron, and E. Conchon. Secure and trusted execution: Past, present, and future - A critical review in the context of the internet of things and cyber-physical systems. In *2016 IEEE Trustcom/BigDataSE/ISPA, Tianjin, China, August 23-26, 2016*, pages 168–177. IEEE, 2016

- R. P. Lee, K. Markantonakis, and R. N. Akram. Provisioning software with hardware-software binding. In *Proceedings of the 12th International Conference on Availability, Reliability and Security, Reggio Calabria, Italy, August 29 - September 01, 2017*, pages 49:1–49:9. ACM, 2017

- R. P. Lee, K. Markantonakis, and R. N. Akram. Ensuring secure application execution and platform-specific execution in embedded devices. *ACM Transactions on Embedded Computing Systems*, 18(3):26:1–26:21, Apr. 2019

Abstract

The Internet of Things (IoT) is considered one of the most significant computing trends of the 21st century so far, causing the number of deployed computing devices to increase dramatically. However, the frequency and severity of attacks against computer systems is increasing with attacks such as Stuxnet and Mirai. This thesis is concerned with securing computing devices by creating bonds between hardware and software to ensure deployed devices only execute the software they are intended to and vice versa. This prevents counterfeiting, device tampering and other attacks.

This thesis presents three studies into binding hardware and software. These concern creating secure dependencies between the hardware and software of computing devices. Specifically, this thesis considers mutual dependencies created by bidirectional bonds between hardware and software. The first study defines hardware-software bonds, the problem space and proposes the first bidirectional scheme for binding hardware and software together. The scheme is implemented using an FPGA and analysed for security and performance.

The second case considers the deployment of binding schemes. This is considered in a smart city environment by first studying and modelling the security architecture of the problem. Two security models are proposed with associated software provisioning protocols. All the proposals are formally analysed using Tamarin Prover and implemented for Raspberry Pi and laptop computer.

The third study combines hardware-software binding with another problem in cyber security - Secure Application Execution. The two problems are compared and set of requirements for simultaneously achieving both is defined. A binding scheme is proposed solving the expanded problem and a prototype Secure Execution Processor equipped with the scheme is presented and analysed.

Finally, the thesis concludes by summarising the research presented and highlighting potential areas for future work in binding hardware and software together.

Contents

1 Introduction **12**

1.1 The Computing Landscape 13

1.2 Contributions 14

1.3 Thesis Structure 15

2 Background **17**

2.1 Embedded Systems and Smart Cards 18

2.1.1 Embedded Systems 18

2.1.2 Smart Cards 19

2.2 Field Programmable Gate Arrays 21

2.3 Physically Unclonable Functions 22

2.3.1 PUF Examples 24

2.3.2 PUF Uses 26

2.4 Binding Hardware and Software 27

2.4.1 Application Binding 28

2.4.2 Algorithm Binding 29

2.4.3 Bond Directionality 30

2.5 Secure Application Execution 32

2.5.1 Proposals for Securing Application Execution 32

2.6 Spectre and Meltdown 35

2.7 Formal Analysis 36

3 Binding Hardware and Software **38**

3.1 Introduction 39

3.1.1 Contributions 40

3.1.2 Structure 40

3.2 Problem Description 41

3.2.1 Notation 41

3.2.2 Problem Scenarios . . . 41
3.2.3 Attacker Model . . . 42
3.2.4 Assumptions . . . 43
3.2.5 Design Requirements . . . 43
3.3 Related Work . . . 45
3.3.1 Anti-counterfeiting measures . . . 45
3.3.2 Hardware Intrinsic Security . . . 47
3.4 Proposed Solution . . . 50
3.4.1 Design Overview . . . 50
3.4.2 Securing Mutable Data . . . 52
3.4.3 Choice of Function F . . . 53
3.4.4 Implementation Considerations . . . 55
3.4.5 Solution Analysis . . . 56
3.4.6 Security Evaluation . . . 57
3.5 Implementation . . . 58
3.5.1 Development Platform . . . 59
3.5.2 Demonstration Application . . . 60
3.5.3 Unmasking Unit . . . 61
3.5.4 Performance Analysis . . . 65
3.5.5 Security Analysis . . . 66
3.6 Conclusion . . . 68

4 Installing and Updating Software with Hardware-Software Binding 69

4.1 Introduction . . . 70
4.1.1 Contributions . . . 71
4.1.2 Structure . . . 71
4.2 Problem Description . . . 72
4.2.1 Entities and Motivations . . . 72
4.2.2 Attacker Model . . . 73
4.2.3 Protocol Requirements . . . 74
4.3 Models for Software Provisioning . . . 75
4.3.1 Device-centric Software Provisioning . . . 76
4.3.2 Authority-centric Software Provisioning . . . 78
4.4 Related Work . . . 80
4.4.1 Binding Hardware and Software . . . 80

4.4.2 MULTOS . . . 81
4.4.3 Global Platform . . . 82
4.5 Proposed Solution . . . 84
4.5.1 Overview of Solution . . . 84
4.5.2 Device-centric Software Provisioning . . . 85
4.5.3 Authority-centric Software Provisioning . . . 86
4.6 Analysis . . . 87
4.6.1 Description of Protocol Models . . . 88
4.6.2 Scope of Mechanical Analysis . . . 89
4.6.3 Modelling Assumptions Required . . . 90
4.6.4 Results of Analysis . . . 90
4.6.5 Further Observations . . . 91
4.7 Conclusion . . . 92

5 Securing Application Execution and Binding Hardware and Software 93

5.1 Introduction . . . 94
5.1.1 Contributions . . . 94
5.1.2 Chapter Structure . . . 95
5.2 Problem Description . . . 95
5.2.1 Platform Specific Execution . . . 95
5.2.2 Secure Application Execution . . . 96
5.2.3 Attacker Model . . . 98
5.2.4 Requirements . . . 99
5.3 Related Work . . . 100
5.3.1 Platform Specific Execution . . . 100
5.3.2 Secure Application Execution . . . 101
5.4 Proposed Solution . . . 105
5.4.1 Protecting Sequential Transitions . . . 105
5.4.2 Protecting Instruction-Dependent Transitions . . . 106
5.4.3 Securing Instruction-Independent Transitions . . . 107
5.5 Implementation . . . 110
5.5.1 Design Decisions . . . 110
5.5.2 Assembler . . . 113
5.5.3 Hardware . . . 115
5.6 Analysis . . . 119

5.6.1 Requirements Analysis 119
5.6.2 Implementation Analysis 120
5.7 Conclusion 121

6 Conclusion 123
6.1 Summary and Conclusions 124
6.2 Future Work 125

A Simple Binding Scheme Test Application Code 126

B Tamarin Model for Device-Centric Provisioning 128

C Tamarin Model for Authority-Centric Provisioning 136

D Tamarin Model for Authority-Centric Provisioning with Pre-Shared Keys148

E Secure-Execution Processor Test Application Code 161

Bibliography 163

List of Figures

2.1 Embedded systems may be built using a combination of many components. 19

2.2 An FPGA consists of programmable logic elements and IO connections to other devices. 22

2.3 An arbiter Physically Uncloneable Function (PUF) may include a source, a number of multiplexer switch blocks and a D flip-flop acting as an arbiter. 25

2.4 Unidirectional binding only restricts one of hardware or software. When only the software is bound, it is restricted to only the hardware it is bonded to whereas the hardware is unrestricted and may execute other software. The same applies in reverse when only the hardware is bound, the hardware is restricted but the software chosen for the hardware is not. Edges between software and hardware indicate permitted installations. 31

2.5 Bidirectional binding of hardware and software ensures bound elements cannot be moved onto other devices or installed with alternative software. Edges indicate an allowed installation of software onto hardware. 31

2.6 Code designed to improve program security is exploited by Spectre. Example code copied from [64]. 36

3.1 Hardware block diagram of the typical embedded system considered in this work. The grey area indicates the "safe zone" impenetrable by attackers. . 44

3.2 The transformation of source memory content into a personalised application 51

3.3 Diagram of implemented system. 60

3.4 A flowchart of the instruction_in handling of the Unmasking Unit. 63

3.5 A flowchart of the calculation of the instruction mask values. 64

3.6 A flowchart of the operation of the PicoBlaze^{TM} pause process. 65

4.1 Device-centric application provisioning only requires connection between Operator (O) and Device (D), however this requires prior trust between D and Authority (A) and between A and O. 77

4.2 The Application-Relay model allows an Authority to monitor application provisioning from O to D. 79
4.3 Application-Broadcast model considers the situation in which O is loading a large number of Devices with a single application. 79
4.4 Modelling Device-centric application provisioning requires a state machine of four states with five connections. 89

5.1 Legitimate executions follow each instruction in a control path such as the ticked instructions, illegitimate executions skip instructions such as the path avoiding the `JUMP NZ, 6` that is marked by crosses. 97
5.2 The binding scheme from Chapter 3 uses a secret, S, and the previous, masked instruction, $M(Op)_{prev}$, to unmask $M(Op)$ and reveal Op for execution. 101
5.3 When protected by only simple hardware-software binding, the code snippet on the left can be modified to leak sensitive data by moving code into a location reached by a `JUMP` operation. 107
5.4 Changing return addresses can be used to skip security critical instructions. 108
5.5 The extra operations added to the instruction set secure the `RETURN` after execution of re-used function code. 109
5.6 The prototype included serial and LED outputs to demonstrate correct execution. 111
5.7 The Secure-Execution Processor consists of six main elements. 117
5.8 The Secure-Execution Processor updates each signal according to a strict schedule. 118

List of Tables

4.1 Protocol Notation . . . 85
4.2 Device-centric Software Provisioning . . . 86
4.3 Authority-centric Software Provisioning (with Public-Key Cryptography) . 87
4.4 Authority-centric Software Provisioning (with Pre-Shared Keys) . . . 87

5.1 Lightweight block ciphers implemented by Maene and Verbauwhede in [78]. 113
5.2 Only the instructions needed for the test application were implemented and thus some common instructions are missing. . . . 115
5.3 Sequential transitions require less memory accesses than unconditional `JUMP`s which require less than conditional `JUMP`s. . . . 118

Chapter 1

Introduction

Contents

1.1 The Computing Landscape **13**

1.2 Contributions **14**

1.3 Thesis Structure **15**

This chapter introduces the thesis by describing the context and need for the research undertaken. Firstly, the landscape of computer systems and recent trends in computing are described. Recent attacks are described as well as some consequences of security breaches to argue the need for better security in computer systems. The chapter finishes by listing the main research contributions of this work and by describing the structure of the document.

1.1 The Computing Landscape

The number of computing devices in the world is growing at an increasing rate and there are now more devices connected to the internet than people on the Earth [120, 18]. Computing power continues to increase, and the number of places it is used expands [102]. One area experiencing the biggest expansion is embedded computing [111]. The Internet of Things (IoT) concept is causing embedded systems to be included into more and more devices including (but not limited to) cars, aeroplanes, industrial control systems, fridges, light bulbs and utility meters [66, 87].

The IoT has many advantages; smart meters can help people to save money by monitoring their energy consumption [103], healthcare processes can be improved using IoT solutions [92] and connected cars may make the roads safer [50]. Furthermore, as well as practical value, the IoT has great monetary value with some estimating the market to be worth $457B or more in 2020 [25, 99].

However, as the deployment of computer systems increases, the number of attacks against them grows; 2017 saw many high profile attacks in which personal data or national infrastructure were targeted [95]. These attacks caused disruption, financial damage and reputation damage to businesses and governments [95]. Overall, the security software company Symantec observed a 600% increase in attacks against IoT systems in 2017 [114].

However, the consequences of insecure IoT devices can be much significant than damage to revenue or reputation. In 2014, Genesis Toys released “My Friend Cayla”, a doll containing an embedded system to allow it to talk to children [40]. In 2015, researchers from Pen Test Partners discovered the insecurity of the dolls developer options made it possible to change the dolls behaviour. This allowed the researchers to force the doll to swear at children (which should not have been possible) or to eavesdrop on conversations held around the doll [94]. The existence of the latter modification caused the toy to be declared as “illegal espionage apparatus” and to be banned in Germany [96].

Another recent attack on computing devices is the Stuxnet malware deployed to sabotage Iranian nuclear power stations. The malware would interfere with plant centrifuges to cause premature wear and damage the Industrial Control System. By disabling legitimate code, the malware was able to run the attack code needed to target the centrifuges [69].

Thirdly, Mirai is a malware that caused significant disruption to a number of internet services including Spotify, Twitter and PayPal [115]. It spread by searching for devices with default login credentials still accepted. Once the malware connected to an open device it would spread itself to the new device to add another bot to the botnet [26]. The final result was a botnet capable of performing 620 Gbps Distributed Denial of Service (DDoS) attacks against targets selected via the command and control system running the malware [115].

These attacks show that more efforts are needed to secure computing devices and systems. There are many approaches to securing computer systems, however one approach is to ensure devices are only running correct, legitimate and trusted software. If this goal were achieved, the attacks described previously would all be thwarted as malicious code would not be executable on the devices as it would be neither legitimate nor trusted and it's behaviour incorrect for the devices.

One way of ensuring that only the correct software can execute on a device is to bind the hardware and software of computing devices together. If an application must be bound to a device to execute and only trustworthy parties may bind software then many attacks, such as those described, are prevented [67, 110, 10]. This thesis is focussed on the development and use of schemes for binding hardware and software in computing devices.

1.2 Contributions

The main research contributions of this thesis are:

- Expanding the definition of binding hardware and software to include directionality (Chapter 3).
- Proposing and implementing the first bidirectional scheme to bind hardware and software together (Chapter 3).
- Analysing software provisioning in smart city environments and proposing three models to fit the setting (Chapter 4).
- Proposing, formally analysing and implementing three protocols to provisioning soft-

ware in a smart city environment (Chapter 4).

- Combining the problem of binding hardware and software with securing application execution to better model security threats facing embedded devices (Chapter 5).
- Proposing and implementing a scheme to guarantee secure and platform specific application execution in embedded devices (Chapter 5).

1.3 Thesis Structure

This thesis is organised in six chapters starting with an introduction to embedded systems and the need for their security.

Chapter 2 introduces the research topics relevant to this work. The background chapter does not exhaustively list all relevant work but describes each of the problems, technologies and techniques used in the later chapters of this document. As such, the chapter defines the problems as considered in this work and provides the information needed to understand the tools and technologies used.

Chapter 3 describes the first hardware-software binding scheme proposed in this work. We argue that previously developed schemes are insufficient and a bidirectional method of binding hardware and software is needed. A scheme is proposed to meet the elicited requirements. The viability of the scheme is demonstrated by a proof-of-concept implementation.

In Chapter 4, we consider how the scheme proposed in Chapter 3 could be used in a real-world environment. This study focusses on how to install and update software when hardware-software binding is used. Considering the case study of a smart city, several models were developed to consider the different software installation models that might apply. Protocols meeting the elicited requirements were proposed, implemented and formally analysed using Tamarin Prover.

Chapter 5 also builds on Chapter 3 by expanding the problem considered to include ensuring correct application execution. An analysis of securing application execution led to a new, transition-based approach to Secure Application Execution (SAE). By combining

the Chapter 3 scheme with ideas from the SAE literature a new binding scheme was proposed to ensure correct and platform specific application execution. The expanded scheme was implemented as a soft-core processor devloped and tested on a Field Programmable Gate Array (FPGA).

Finally, Chapter 6 concludes this thesis by summarising the work presented and describing potential directions for future work in the area.

Chapter 2

Background

Contents

2.1 **Embedded Systems and Smart Cards** 18
2.2 **Field Programmable Gate Arrays** 21
2.3 **Physically Unclonable Functions** 22
2.4 **Binding Hardware and Software** 27
2.5 **Secure Application Execution** 32
2.6 **Spectre and Meltdown** 35
2.7 **Formal Analysis** 36

This chapter introduces the relevant background material to this work. Early sections describe the main technologies associated with the research presented in later chapters of this thesis. Schemes are proposed for embedded system or smart card (Section 2.1) use, prototyped using FPGAs (Section 2.2) and including PUFs (Section 2.3). Background research presented concerns binding hardware and software (Section 2.4), the focus of this work. However, later chapters include other topics including secure application execution and formal analysis, described in Sections 2.5 and 2.7. Two recent, significant attacks on processor technologies, Spectre and Meltdown, are included for completeness in Section 2.6.

2.1 Embedded Systems and Smart Cards

Embedded systems and smart cards are the most prevalent types of computing devices in the world [120, 52]. Both have become a significant part of modern life; and are the subject of academic and industrial interest. This section defines both types of device and how they are considered in this document.

2.1.1 Embedded Systems

This thesis proposes multiple schemes for securing embedded systems. Embedded systems are defined by P. Marwedel as "information processing systems embedded into enclosing products" [88]. They are included in various settings including: cars, home electronics and Smart City infrastructure. In many of these, the presence of the computer is not obvious due to the difference of the form factor from "traditional" computers. This has led to embedded systems also being called *disappearing computers* [88]. Other titles given to the spreading of embedded systems include: *ubiquitous computing* [122], *pervasive computing* [105] and the Internet of Things (IoT) [11, 9].

Regardless of labels, embedded computing is concerned with the deployment of embedded systems. These are small, low power devices deployed for particular tasks. A general purpose processor or microcontroller may be included, but the device is deployed for a specific purpose. Some controllers operate washing machines, others network hardware or motor vehicles. The common feature is the devices are small, low-powered, and computing resources are often limited [88].

Embedded systems are complete systems deployed for a purpose, comprised of a number of elements [107]. Typically, an embedded system features a processor or microcontroller, this may include specialist hardware such as a cryptographic co-processor or Digital Signal Processor (DSP). Storage, such as flash memory, Read-Only Memory (ROM) or Electronically-Erasable Read-Only Memory (EEPROM) is included to store software and other data needed. Finally, some Input/Output (IO) devices may be attached to the system; these could be sensors to record information or actuators for interacting with the environment. In some cases, network capability is included allowing the device to communicate with other devices or servers. A diagram of an embedded system and the elements

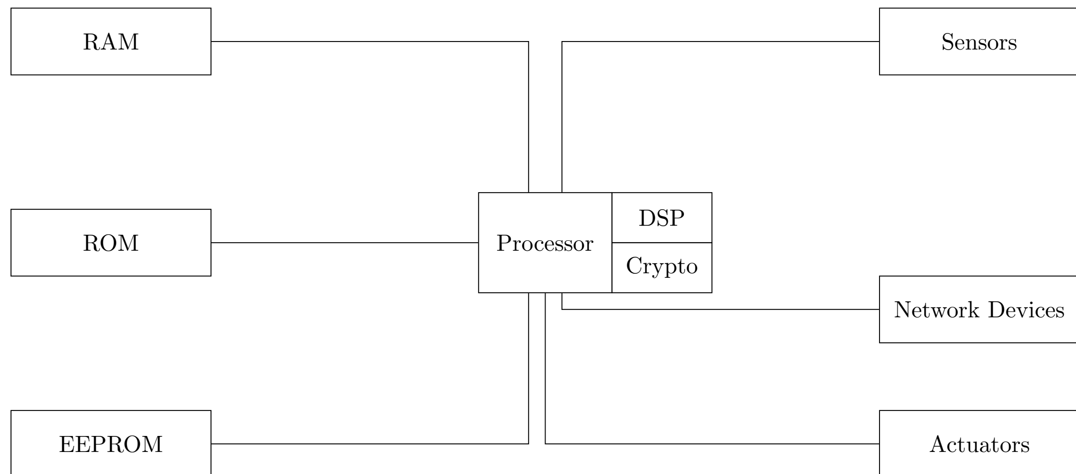


Figure 2.1: Embedded systems may be built using a combination of many components.

that may be included is included in Figure 2.1.

The diagram in Figure 2.1 demonstrates many of the possible components that may be included in an embedded system as considered in this thesis. We assume they will include a processor, some long-term storage and Input/Output devices such as sensors or actuators. Some may also include a cryptographic co-processor or digital signal processor. However, embedded systems in this work will always include a processor, storage and some means of interacting with the world.

2.1.2 Smart Cards

As well as applying to embedded systems, the schemes proposed in this thesis may apply to securing smart cards. Smart cards are small computing devices integrated into standardised forms used to add security to a transaction or communication. They are equipped with unique IDs, secure storage, security algorithms and tamper-resistance technologies to prevent forgery or duplication. Devices sometimes described as smart cards are magnetic stripe cards, these are cards without processors and equipped with magnetic stripes. However, these cards offer no security and are incapable of processing so are not considered in the definition of smart cards in this work. The most common forms of smart cards are the 86mm×54mm form factor of credit and debit cards and the smaller (25mm×15mm to 12.3mm×8.8mm) mobile phone SIM card forms. Like embedded systems (Section 2.1.1), smart cards are deployed as application specific devices. Common uses of smart cards include: payments, transport ticketing, mobile telephony, physical

access control and identification [90].

Smart card communications, structures and security mechanisms are carefully standardised. While there are many details covered, the sections on application protection and secure application installation are the details significant to this work. There are several standards and standardisation bodies involved in smart cards. The main organisations and bodies are ISO, GlobalPlatform and MULTOS [90].

The International Organisation for Standardisation (ISO) is the main creator of smart card standards. Four main standards cover smart cards: ISO 7816, ISO 14443, ISO 15693 and ISO 7501 [57, 56, 53, 54]. ISO 7816 covers most details of smart cards including sizes, interfaces, file structures, executable commands and cryptographic services [57]. Close-proximity contactless cards are described by ISO 14443, whereas ISO 15693 describes longer range "vicinity cards" [56, 53]. Smart cards also feature in ISO 7501, although it primarily concerns machine-readable travel documents and smart cards are only covered in that context [54]. However, while these cover many important details, the most relevant standards to this thesis are those from GlobalPlatform and the MULTOS consortium covering secure mechanisms for installing and managing applications.

GlobalPlatform, like ISO, is another standardisation group producing standards for smart card behaviour [42]. GlobalPlatform standards are also applied to securing connected cars and IoT devices [43]. The GlobalPlatform card specification is more application and security focussed than ISO standards and does not describe form factors or interfaces. However, it does cover how applications may be securely installed and the architecture ensuring GlobalPlatform compliant card trustworthiness. Application management is included in the GlobalPlatform specification with cryptographic means for ensuring software installation and verification [42]. This is significant as secure installation and management of applications is critical to securing computing devices.

Finally, MULTOS is an operating system for multi-application smart cards developed by the MULTOS consortium [85]. Like GlobalPlatform, MULTOS is designed for security and enabling multiple applications to be securely installed to a single card. MULTOS has been included in various devices including smart cards, ID cards, passports and ticketing [83]. Security is a major focus of MULTOS and cryptographic methods are included to secure communication between MULTOS devices, Issuers and Application Providers. MULTOS

provides methods for secure software installation and updating [85]. These methods of securely distributing applications to remote devices can help secure cards against rogue applications. In MULTOS, security is maintained by certificates issued to each software before it may be installed [85].

2.2 Field Programmable Gate Arrays

FPGAs are reprogrammable hardware devices configurable for many different purposes including prototyping, computation and digital audioprocessing. A typical FPGA includes a two-dimensional array of configurable blocks, similar to a Programmable Array Logic (PAL) chip, but more complex and with a different development process. PALs required expert programming based on intimate knowledge of the architecture, FPGAs are programmed using powerful Computer Aided Design (CAD) software. Instead of a precise gate-level design, a designer uses Hardware Description Language (HDL) to describe required behaviour and the tools decide how to produce working circuits [49].

FPGAs are commonly used in prototyping circuit designs. Their re-programmable nature allows a design to be tested in real silicon without fabrication. However, FPGAs are also used in cases when a bespoke circuit is needed but a bespoke Integrated Circuit (IC) is not cost-effective. In these cases an FPGA provides the required functionality while minimising production cost [49].

Rapid prototyping is the main use of FPGAs in this thesis. In later chapters, two FPGA implementations are described demonstrating schemes proposed in this work. This approach allows a scheme to be demonstrated with real circuits rather than in theoretical models or simulations [49].

Many FPGAs, such as those produced by Xilinx Inc. and used in this work, consist of an array of programmable blocks programmed to create a circuit. Blocks include two components: a logical component and a routing component. The logical component receives block inputs and performs a function on some, all or none of the inputs. In Xilinx devices, the logical component is a Lookup Table (LUT) indexed by the inputs, allowing any boolean function. The result of the logic component is then routed to nearby blocks, as determined by the CAD tool. This logical structure allows for any possible boolean

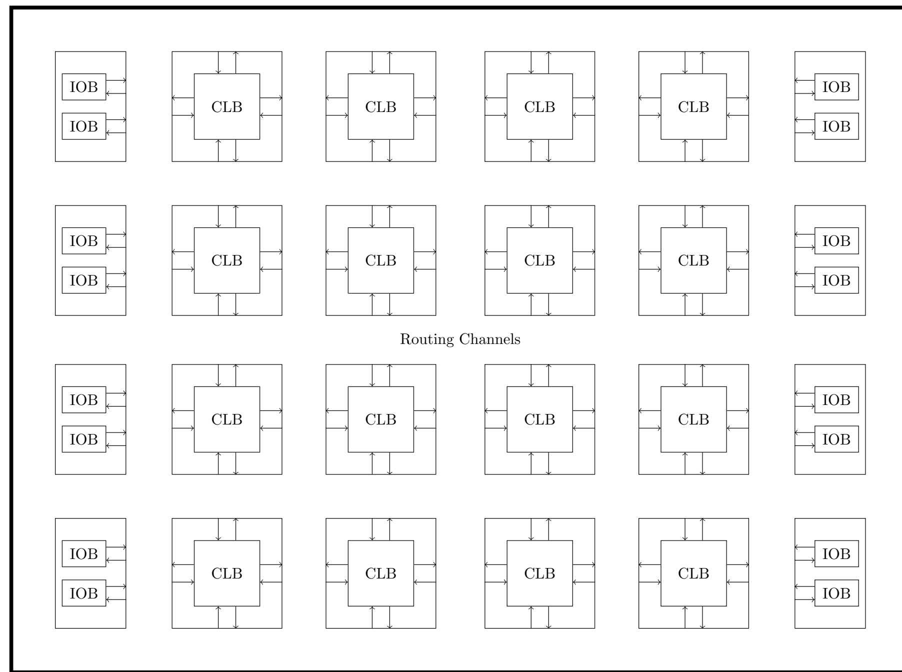


Figure 2.2: An FPGA consists of programmable logic elements and IO connections to other devices.

functions of the block inputs to be programmed into a block. A complicated equation or a simple one will each require the same number of resources and give the same performance [49].

In addition to programmable logic blocks, FPGAs contain IO blocks to allow the FPGA to interface with other components and devices [49]. For example, the Xilinx Spartan-6 FPGA evaluation board used in this thesis includes UART, ethernet and headers for other I/O devices [124].

A diagram of this typical FPGA structure can be found in Figure 2.2. For clarity, the wiring between blocks is omitted, however, in many FPGAs, each block has the ability to connect to any other block in the FPGA [108].

2.3 Physically Unclonable Functions

Physically Uncloneable Functions (PUFs) as considered in recent literature were introduced in 2002 by Gassend et al. as Silicon Physical Random Functions [39]. A PUF can be considered a hardware biometric or as a fingerprint of an object [79]. Attempts had

been made to exploit unclonable characteristics of physical devices before 2002 [14]. However, Gassend et al. were the first to propose silicon circuits with unclonable properties [39]. Since the work of Gassend et al. however, there have been many PUF designs proposed including: SRAM PUFs [45], Ring Oscillator PUFs [112], Arbiter PUFs [70] and FPGA specific designs [6, 68, 82]. This work considers PUFs as designs implemented in silicon integrated circuits.

We consider a PUF to be a circuit where two implementations do not exhibit precisely the same behaviour. PUFs provide device-specific output based on inputs received, giving similar (up to a small error rate) responses to repeated challenges and different responses to different challenges [79]. In 2010, Maes and Verbauwhede defined PUFs as *physical challenge-response procedures* [81]. They denoted PUF functionality by: $\Pi : \mathcal{X} \rightarrow \mathcal{Y} : \Pi(x) = y$, where Π is the PUF instance, $\mathcal{X}$ and $\mathcal{Y}$ the input and output spaces in which the input x and output y are defined. Maes and Verbauwhede also listed seven properties defining PUFs. These properties state a PUF must be: Evaluatable, Unique, Reproducible, Unclonable, Unpredictable, One-way and Tamper evident [81]. These properties from Maes and Verbauwhede in [81] will now be described.

- Evaluatable: with the PUF, Π and an input, x, it should be easy to calculate $y = \Pi(x)$. This requires that the PUF must be usable in polynomial time [81].
- Unique: each PUF output, $\Pi(x)$, should provide some information about the physical entity that contains Π. Each PUF should produce outputs so that given a number of them, they should uniquely identify the entity containing Π [81].
- Reproducible: every time a PUF output, $y = \Pi(x)$ is computed it should be approximately unchanged, subject to a small error rate. PUFs exploit random characteristics that are often susceptible to noise. However, outputs should be approximately repeatable for the PUF to be useful [81].
- Unclonable: a critical property, it declares that given a PUF, Π, it should be hard to construct a procedure, $\Gamma \neq \Pi$ s.t. $\forall x : \Gamma(x) \approx \Pi(x)$ within a small error rate [81]. Maes and Verbauwhede explicitly emphasise that Γ is a procedure and need not be a hardware function. This is drawn from their defining two types of cloning: mathematical cloning and physical cloning. Mathematical cloning requires an attacker to create a function, f, copying the behaviour of Π subject to error. Whereas, physical

cloning involves creating an alternative physical entity containing a PUF, Π' where $\Pi' \neq \Pi$, mimicking Π for all input challenges, subject to error [81].

- Unpredicatable: this requires that given a set $Q = (x_i, y_i = \Pi(x_i))$ and an x_c, s.t. $x_c \notin Q$, the attacker is unable to predict y_c s.t. $y_c = \Pi(x_c)$. Maes and Verbauwhede connect this property with mathematical uncloneability as follows. If a PUF is predictable, it is possible to create a mathematical clone of the PUF and violate the Unclonablility property of the PUF [81].

- One-way: this property is inherited from traditional cryptographic definitions and requires that given a y and a Π, it is hard to find an x s.t. $\Pi(x) = y$ [81]. This property adds little to the others listed; Maes and Verbauwhede admit to including it only because early PUF literature considered PUFs as physical one-way functions.

- Tamper evident: finally, this property requires that if the physical entity containing a PUF, Π, is interfered with, it will transform Π into a Π' s.t., with a high probability, $\exists x \in \mathcal{X} : \Pi(x) \neq \Pi'(x)$ excluding error, where (X) is the set of all possible inputs, x [81]. This property is used in many schemes proposed that use PUFs and it relied upon for security by ensuring any attackers attempting to tamper with the device will cause a lasting, and noticeable, impact.

In this work, we assume PUFs behave perfectly and have an error rate of 0%. This means that for a given PUF instance, the same input will always yield the same output. We also assume outputs of PUF instances are high entropy, perfectly unpredictable and unclonable. We assume the PUF is one-way and tamper evident and that we are able to access it efficiently.

2.3.1 PUF Examples

One common PUF design is the SRAM PUF; a simple design using an unmodified SRAM to provide device-unique outputs. A PUF provides an output, in the case of SRAM PUFs the device-specific output is the initial state of the chip. This is usable as a PUF because when an SRAM powers up, the memory cells initialise as either 0s or 1s with bias. The initial value is not set by design but by variations in the cells from the manufacturing process. The SRAM PUF is queried by performing a memory lookup and receiving the

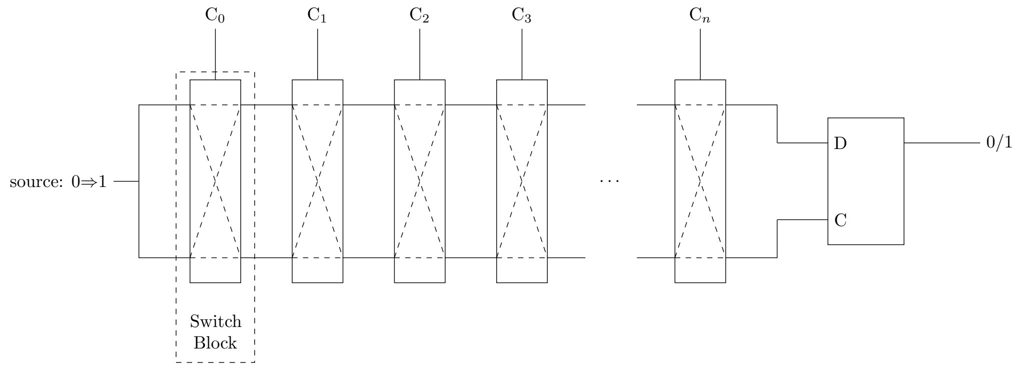


Figure 2.3: An arbiter PUF may include a source, a number of multiplexer switch blocks and a D flip-flop acting as an arbiter.

result of that lookup. This type of PUF has a limited number of challenge/response pairs, so it is classified as a "weak PUF". Note: the term weak does not refer to the security of the design but the number of distinct responses it can provide [45].

Alternatively, there are "strong PUFs" providing a large (exponential) number of challenge-response pairs. One strong PUF is the arbiter PUF, these require dedicated circuits to provide a PUF output by comparing signal transmission delay between wires of equal length. Theoretically, the wires should exhibit equal delay, however, due to production variations this is not true in reality. In practice, a single source connected to two wires is changed from low to high, the signal travels down the wires to the arbiter which returns a 0 or 1 depending on which signal arrived first. One potential arbiter is a D-type flip-flop with the two wires connected to the data and clock lines. Whichever is set first will determine if a 0 or 1 is output from the flip-flop. A simple arbiter PUF consists of $2N$ multiplexers connected in two lines from source to arbiter. The path used from source to arbiter is determined by the N-bit input challenge passed bitwise to each pair of multiplexers, when the bit is 1, the wires are crossed over, otherwise they are unchanged. A diagram of an arbiter PUF is shown in Figure 2.3. Due to the input provided to the circuit, an arbiter PUF has 2^N challenge-response pairs making it a strong PUF, even though different challenges are correlated [70, 59].

2.3.1.1 PUF Security

The previous subsection described two different PUF designs to demonstrate examples of how authors have attempted to implement the theoretical model described at the start of

this section. One common theme in PUF research however, is the difficulty in creating designs that successfully meet the Unclonablility and Unpredictability properties. A main work studying the security of different PUF designs was published in 2012 by Katzenbeisser et al. [59].

Some recent attacks on PUF designs have made use of machine learning to model the PUF and violate the Unpredictability and Unclonability of designs. Recent works include attacks against Bistable Ring PUFs by Ganji et al. in 2016 [38] and Becker's attack against XOR Arbiter PUFs in 2015 [15].

Other attacks have used less conventional methods to compromise the properties of PUF designs. These include the characterisation attack on Arbiter PUFs demonstrated by Tajik et al. in 2014 that modelled instances using photonic emission analysis [116] and Helfmeier et al. who used Focussed Ion Beam circuit editing to clone an SRAM PUF [48].

2.3.2 PUF Uses

Due to providing device-specific, random outputs, PUFs are an attractive security primitive to designers. Three main applications of PUFs arising in academic literature are: authentication, key generation and preventing counterfeiting.

Strong PUFs offer a set of challenge-response pairs, leading several authors to include them in authentication systems. Like biometrics for humans, a PUF is something a device is and not merely information it knows. Therefore, due to the Uniqueness property identified by Maes and Verbauwhede, a PUF can be used to identify devices uniquely. The authentication system would be intialised by the PUF-equipped device registering a number of challenge-response pairs with the identifier. When the device wishes to identify itself, the identifier would provide it one or more challenges. The device would respond with the PUF response and if these match, the identifier can confirm the identity of the device. If a PUF is Unique, it provides information about the device with each PUF-response. Furthermore, if the PUF is Unpredictable, Unclonable and one-way the identifier can be confident no other PUF, function, or entity would be able to correctly respond to the provided challenges [81]. Examples of PUF-based authentication schemes are found in [104, 89].

PUFs can also provide a device-specific sources of randomness for generating key material. This application mainly relies on the Uniqueness and Reproducibility properties. If a PUF is Unique, the outputs it produces contain information about the identity of the device containing the PUF. This ensures keys generated by the device will exist uniquely for that device. Furthermore, if the outputs are Reproducible, the device need not store key material - the PUF will reproduce keys when required [81]. Many PUF-based key generation schemes exist such as those of Paral and Devadas [97] and Liu et al. [75].

However, PUFs are significant in this thesis due to the intrinsic, device-specific randomness they offer by design. As such, schemes to prevent counterfeiting have been proposed by authors such as Simpson and Schaumont [110] and Guajardo et al. [45]. In their schemes, the authors proposed mechanisms by which hardware and software developers can combine products without exposing their Intellectual Property (IP) to the other party. PUF provided keys ensure IP elements can be decrypted by the target device only and no others devices [110, 45]. In these cases, the properties needed of the PUF are it being Unique, Reproducible, Unclonable and Unpredictable. These ensure the only devices with access to the protected IP are the devices for which use is authorised.

2.4 Binding Hardware and Software

Binding hardware and software is a little studied area of information security with three works comprising most of the literature in the area [67, 110, 10]. The major work in the area is a 2003 patent by Krasinski and Rosner proposing a scheme for binding copy protection software to devices to prevent copyright infringement [67]. In academia the most significant papers are by Simpson and Schaumont in 2006 [110] and a later work by Atallah et al. in 2008 [10].

In this thesis, binding hardware and software concerns combining the two elements together. The bond ensures a secure dependency between hardware and software that prevents correct operation if the entities are separated. Software cannot be moved, or copied, from one hardware to another as the removal of the bound hardware will prevent correct execution. For the bound system to correctly function it requires the presence of both correct the hardware and the correct software for the hardware [67, 110, 10].

The bond between hardware and software proposed in previous literature can be divided into two categories: application binding schemes and algorithm binding schemes. This section includes algorithm binding for completeness however the research in this thesis only explores application binding and "Hardware-software binding" and related terms are used to refer to application binding schemes only.

2.4.1 Application Binding

As stated, binding hardware and software concerns securely combining hardware and software together. Application binding creates this bond by attaching an application, or IP, to a specific hardware instance. The schemes proposed by Krasinski and Rosner and also by Simpson and Schaumont are both application binding schemes [67, 110]. In both works, the authors use application binding to provide copy protection by ensuring the bound software cannot be removed from the hardware [67, 110].

Krasinski and Rosner proposed using device specific information to bind copy protection software, required for accessing secured media, to a particular device. Therefore, instead of binding the (much larger) media to the device, the authors could bind just the decoding application to the device. Efforts to move the protected content and thwart the copy protection are prevented because the decoding software is bound to the device using a device-specific key [67].

Similarly, Simpson and Schaumont use a PUF (Section 2.3) to bind third-party IP to the particular FPGA (Section 2.2) that will be hosting it [110]. Their scheme allows system developers to authenticate IP from developers and for IP developers to restrict their technology to authenticated parties. Simpson-Schaumont binding relies on encrypting hardware blocks using device-specific keys derived from PUF outputs. As the key is device-specific, it is not possible for a malicious party to attack the scheme and discover the IP. Simpson and Schaumont use a Trusted Third Party (TTP) to share key information between system and IP developers. Devices enroll keys with the TTP and these are shared securely with the IP developer who encrypts their product knowing that only the intended device may decrypt it. One issue with this scheme is it relies on all developers being honest and not leaking protected IP after decryption. However, a more serious flaw is that, as the scheme is only designed to protect the IP, it offers no protection for the FPGA system

against malicious IP or IP added by attackers [110].

2.4.2 Algorithm Binding

Algorithm binding is an alternative method of binding hardware and software. We define it as hardware-software binding using device specific information to bind the software at an algorithm level. This differs from the schemes described in the previous subsection; whereas they take the application and attach it to the device entirely, an algorithm binding scheme ties correct output of the computation to the particular hardware device. The first, published, algorithm binding scheme was proposed in 2008 by Atallah et al. [10].

Atallah et al. also proposed a scheme binding software to a hardware instance to prevent it being transplanted to a different device. The attacks they considered were of software being ran in a virtual environment replicating the behaviour of correct hardware. The authors proposed using hardware-software binding for copy protection and also for securing battlefield applications where rehosting by enemies is a national security threat. Like the Simpson-Schaumont scheme, the Atallah et al. scheme includes a PUF. However, where Simpson and Schaumont proposed binding through device specific encryption, Atallah et al. integrate a PUF result into correct application execution by including a device specific value into the computation. This ensures applications only produce correct answers if they have access to the hardware they are intended to be executed on. Due to the properties of PUFs (Section 2.3), attacks virtualising or emulating hardware are prevented because the virtual PUF will be unable to produce the correct value for the algorithm [10].

The example application Atallah et al. described binding was an implementation of RSA encryption[1]. However, where the encryption was split into two encryptions: one under a modified key and a second using a PUF output, R_i. The modified key is calculated by multiplying the original key, e, by the inverse of the PUF output[2], R_i^{-1} to calculate R_i'. The modified encryption algorithm is then computed in two stages [10]:

[1]However, it would apply to RSA decryption too.
[2]Given p and q, calculating the inverse of R_i is straightforward.

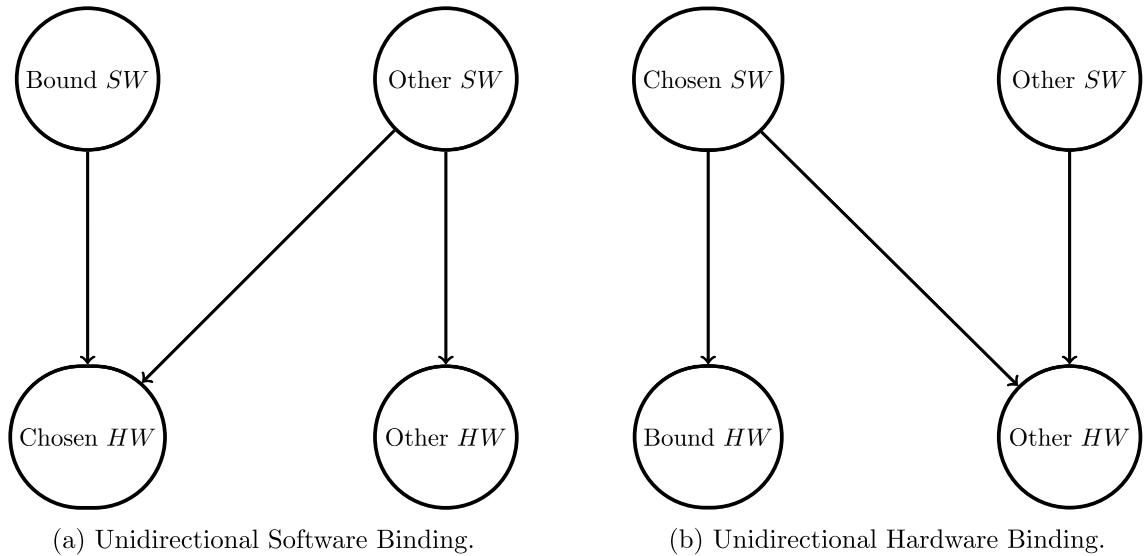
$$\begin{aligned}
\mathsf{Enc}_{R_i'}(m) &= m^{R_i'} \mod n \\
\mathsf{Enc}_e(m) &= \mathsf{Enc}_{R_i'}(m)^{R_i} \mod n = m^{eR_i^{-1}R_i} \mod n = m^e \mod n \\
&= c.
\end{aligned}$$

This method ensures that only hardware in possession of the correct PUF can successfully compute the RSA encryption. Therefore, the Atallah et al. scheme creates a secure bond attaching software to a hardware instance at the algorithm level [10]. However, as with the application binding schemes, the Atallah et al. scheme only binds software to hardware. Therefore, it provides no protection to devices from malicious software or from software modification.

2.4.3 Bond Directionality

A significant concept in binding hardware and software together is "directionality". In this work, the term "directionality" defines a goal of a hardware-software bond. There are two different directionalities of hardware-software bonds: unidirectionality and bidirectionality. Unidirectional bonds tie one element (hardware or software) to the other element (software or hardware). The previous schemes by Krasinski-Rosner [67], Simpson and Schaumont [110] and Atallah et al. [10], described in this section create unidirectional bonds; the software (or algorithm or IP) is bound to the hardware it is intended for. Unidirectional binding is illustrated in Figure 2.4 where an arrow indicates where a software installation is permitted.

The alternative to unidirectional bonds is bidirectional hardware-software binding where bonds securely connect both elements together. With a bidirectional scheme, a piece of hardware only executes software bound to it *and* software only executes on the hardware it is bound to. The permitted installation combinations of software and hardware when bidirectional-software binding is used are shown in Figure 2.5.



(a) Unidirectional Software Binding.

(b) Unidirectional Hardware Binding.

Figure 2.4: Unidirectional binding only restricts one of hardware or software. When only the software is bound, it is restricted to only the hardware it is bonded to whereas the hardware is unrestricted and may execute other software. The same applies in reverse when only the hardware is bound, the hardware is restricted but the software chosen for the hardware is not. Edges between software and hardware indicate permitted installations.

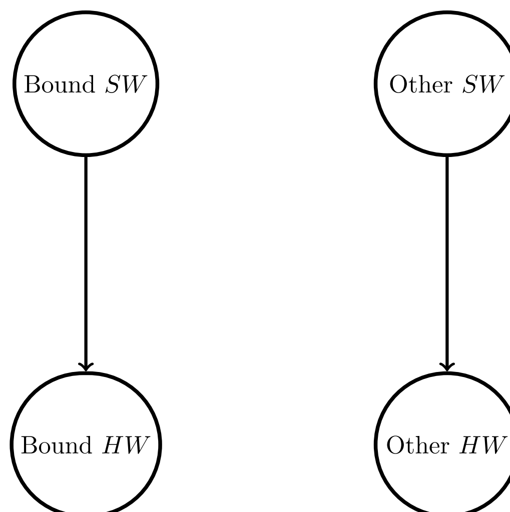


Figure 2.5: Bidirectional binding of hardware and software ensures bound elements cannot be moved onto other devices or installed with alternative software. Edges indicate an allowed installation of software onto hardware.

2.5 Secure Application Execution

Secure Application Execution (SAE) concerns ensuring software runs correctly; following a legitimate path through the application without illegitimately avoiding any instructions or functions. A major work in this area is the 2002 paper by Kiriansky et al. who sought to prevent attacks by only allowing legitimate behaviour. Previous works prevented attacks exploiting particular vulnerabilities whereas Kiriansky et al. sought to prevent unauthorised behaviour rather than attacks [62]. Kiriansky et al. did not formally define SAE, instead considering broadly defined exploits comprising three features: a vulnerability exploited, a program address overwritten and some malicious code executed. Their scheme prevents any attacks from executing malicious code after exploiting any vulnerability to divert execution [62]. This is the definition of a Secure Application Execution scheme used in this work.

Similar to Secure Application Execution is Control-Flow Integrity (CFI). While SAE is a goal, CFI is a security property defined by Abadi et al. in 2005 [1]. Ensuring CFI guarantees program execution has followed a legitimate control-flow, or path, through the application. A legitimate execution path starts at the beginning of the application and runs every instruction in the path in sequence through the application until reaching the end. A legitimate sequence may not run all instructions or run instructions in series. Control-flow instructions may be in the sequence diverting execution to reused code or avoiding code not in the current path. Control-flow instructions are those transfering execution to remote instructions and include: `JUMP`, `CALL` and `RETURN` instructions and their conditional variants [1, 2].

This work uses the Kiriansky et al. and Abadi et al. definitions of Secure Application Execution and Control-Flow Integrity as securing the execution of an application to legitimate paths through the application, preventing all others [62, 1, 2].

2.5.1 Proposals for Securing Application Execution

Many schemes have been proposed to securely ensure program execution. This section describes the two main types of SAE schemes: software-based and hardware-based schemes. The schemes described are designed to prevent unauthorised behaviour by ensuring SAE.

2.5.1.1 Software-Based Control-Flow Integrity Enforcement

The first software-based scheme is program shepherding, published in 2002, by Kiriansky et al. Program shepherding protects against threats with three techniques: restricting code origins, restricting control transfers and un-circumventable sandboxing. These properties are connected; sandboxing is used to defend the first two techniques securing execution [62].

Program shepherding is a security framework allowing different security levels to be set according to a security policy. In their paper, Kiriansky et al. provided a table detailing different program features and security options for each sorted from least to most restrictive. For example, a non-restrictive policy may allow any indirect call, a maximally restrictive policy may allow none [62].

Kiriansky et al. implemented their scheme by extending an interpreter to allow real-time execution monitoring. When tested, their implementation protected against several known vulnerabilities [62].

Another software-based scheme is inlined CFI enforcement proposed by Abadi et al. in 2005, it is an attack-agnostic method of protecting execution. CFI enforcement limits execution to the Control-Flow Graph (CFG) determined by the software developer [1, 2]. A CFG is a map showing all possible correct paths when executing the program. CFGs can be generated from application binaries, several methods have been proposed (e.g. [23, 117, 61]) but the precise method used is out of the scope of this work.

The CFG considers a program as a set of nodes and transitions. Each transition is the result of a control-flow instruction and each node represents a basic block. Basic blocks are groups of one or more instructions, executed in series, in which only the final instruction may be a control-flow instruction.

Abadi et al. implemented inlined CFI enforcement by labelling each control-flow instruction and their destinations. A control-flow instruction is only permitted if the attached label matches the label of the destination. This prevents attacks transferring control flow to arbitrary locations. The authors implemented their proposal as a tool to rewrite x86 binaries to include the proposed checks. The implementation prevented "jump-to-`libc`"

attacks on tested binaries, but not those manipulating the program to improperly launch other software [1, 2].

2.5.1.2 Hardware-Based CFI Enforcement

Several authors have proposed methods of securing application execution using hardware rather than software extensions. These provide the same level of CFI but with a reduced performance penalty as protections are included in device architecture. One significant hardware-based scheme was proposed by Arora et al. in 2005; using a hardware monitor to ensure execution security. Like Kiriansky et al., Arora et al. proposed a flexible primitive tailorable to the granularity of execution security available. They proposed three security measures of increasing resolution: inter-procedural control-flow, intra-procedural control-flow and integrity of the instructions executed [8].

Inter-procedural control-flow secures flow between different functions of the application. Intra-procedural control-flow checks every control-flow transfer in the application, between basic blocks and function calls. Finally, a running hash computation ensures integrity of instructions executed. The hash values calculated on groups of instructions are compared against stored, precomputed values to ensure correct instructions have been executed [8].

Arora et al. simulated an implementation of their of their proposal to test its performance using SimpleScalar and the PISA architectural simulation toolset [8]. Inter-procedural control-flow checks are facilitated by tables storing function start and return addresses and a finite state machine checking call and return validity. A basic block table was added to check intra-procedural control-flow by tracking control flow within functions. Finally, an instruction buffer and hash engine verify instruction integrity by computing hash values of basic blocks during execution. These values are compared with a column in the basic block table containing expected values. These measures ensure secure application execution at the three levels of granularity proposed [8].

Another hardware-based control flow integrity architecture is SOFIA, proposed by Clercq et al. in 2016 [32], publishing an extended paper in 2017 [31]. Like Arora et al., the authors combined CFI and Software Integrity (SI) protections to ensure correct, unmodified instruction execution.

SOFIA includes two main components, encryption and Message Authentication Codes (MACs). To ensure SAE, each instruction is encrypted using a random value, the Program Counter (PC) value of the previous instruction and the PC of the current instruction. This ensures each instruction is encrypted according to its position in the application control-flow, preventing instruction tampering. To ensure SI, as instructions execute they are combined in a runtime MAC compared to a precalculated value to ensure SI [32, 31].

2.6 Spectre and Meltdown

Spectre and Meltdown are two significant attacks on modern processors, published in 2018, exploiting optimisation techniques used to improve performance. Both attacks are based on one class of vulnerability and Meltdown is considered a variant of Spectre [64, 74]. The difference between the two is Spectre is effective against processors from Intel, AMD and ARM, whereas Meltdown only effects some Intel devices [64].

To increase performance, modern processors use techniques including branch prediction and speculative execution. These are used when a branch decision is dependent on a memory access. For example, in Figure 2.6, `x` is compared to the `array_size` stored in memory. In this case, while waiting for the result of the memory access, the processor predicts the boolean statement will be true and proceeds to execute the branch, this is a speculative execution. In this speculative execution the processor continues executing and when the memory access completes, it determines if the current state should be kept (the branch prediction was correct) or rolled back to the state at the branch (if the prediction was wrong). In the latter case execution follows the branch previously not taken requiring any changes from the speculative execution to be undone [64].

If a speculative execution is abandoned, the processor should undo all changes made to ensure correct execution of the other branch. However, in practice, the authors discovered processors detectably affecting the cache during speculative execution. This cache effect is a side channel that authors were able to develop code to exploit and use to leak information about memory contents [64].

In the example given in Figure 2.6, the attacker first trains the CPU to assume the conditional statement will most likely return true. Second, the cache is manipulated

```
if (x < array_size)
  y = array2[array1[x] * 256];
```

Figure 2.6: Code designed to improve program security is exploited by Spectre. Example code copied from [64].

to exclude `array_size`. Therefore, when executed, the code will start to perform the comparison and while waiting for the memory access to complete, it speculatively executes the if-true condition. The value `x` and the address of `array1` are added and the content of that location read to discover the secret byte k. This k is used to form a memory access in `array2`. When `array_size` is read from memory, the speculative execution is declared erroneous and the register states are rewound. However, the speculated read of `array2` affects the cache by moving the value read into cache space. The attack is completed by testing the cache to determine the address accessed by the speculative read, the index accessed with the lowest latency is k [64].

These attacks have prompted the discovery of similar attacks on Intel SGX [21], virtual memory [47] and cryptographic key generation [4]. However, these cache-based side-channel attacks are considered outside of the scope of this work and have been included for completeness only.

2.7 Formal Analysis

Formal analysis, also known as symbolic, automatic or mechanical analysis involves using software tools to analyse protocols. In formally analysing a protocol, a model of the protocol is used by the tool to reason about the scheme. From the model, analysis tools perform detailed analysis of the protocol and can find attacks missed by human analysis [28].

One difference between human and formal analysis is analysis tools usually consider all cryptographic primitives to be completely secure, this is called the Dolev-Yao model. In this model, the attacker can intercept all messages but not break any cryptographic functions. Analysis tools search for protocol logic flaws allowing attackers to violate security properties. This search is carried out differently in different tools. Some tools test common security features of protocols, others allow the user to define properties [28].

Many tools exist for protocol analysis including Casper FDR [76], Scyther [27] and Tamarin [91]. Different tools have different strengths and weaknesses studied by multiple authors [20, 28, 98]. However, the merits of different tools is not the focus of this research and will not be discussed in detail.

In this work, Tamarin prover ([91]) is used to analyse protocols. Like Scyther, Tamarin uses backwards search to explore the protocol space. Protocols are specified with multi-set rewriting rules and properties by guarded first-order logic. This tool was chosen due to its accurate modelling of Diffie-Hellman Key Exchange, a featured used in all protocols proposed. Furthermore, proving properties is greatly aided by the Tamrin interactive proof mode [91].

Chapter 3

Binding Hardware and Software

Contents

3.1 Introduction 39
3.2 Problem Description 41
3.3 Related Work 45
3.4 Proposed Solution 50
3.5 Implementation 58
3.6 Conclusion 68

Binding hardware and software can guarantee software executes only on the device it was intended for. Similarly, hardware-software binding can ensure hardware only executes software bound to it to prevent execution of unauthorised software. This two-way bond is an extension on previous work which focusses on unidirectional bonds attaching software to hardware. This chapter presents a new scheme that creates bidirectional hardware-software bonds. First, the problem area is examined and requirements for a hardware-software binding scheme are defined. Later in the chapter, a scheme is proposed and a proof-of-concept implementation described that demonstrates the scheme on a Xilinx Spartan-6 FPGA.

3.1 Introduction

This chapter considers securely binding software to individual hardware devices and vice versa. If a piece of software has been bound to a hardware device, the software will only run on the hardware it has been loaded onto. Binding software and hardware can prevent unauthorised firmware tampering, platform counterfeiting or be used to protect intellectual property.

Intuitively, unauthorised firmware alteration and counterfeiting can be prevented in the following way. Only the legitimate manufacturer may install or provision software for execution because only they are able to bind software to their hardware. Conversely, attackers are unable to use alternative firmware or create counterfeit products using legitimate products because software extracted from one device cannot be copied onto another because it will only operate correctly on hardware it is bound to.

We propose the following setting as a real-world case study for our problem. A manufacturer has developed a controller for use in an Industrial Control System (ICS). The ICS may be a production line or a major part of the functioning of the organisation owning and operating the system. As the system is of great importance, security of the devices and their applications is critical. The manufacturing or operational process is the result of a large investment and so it is very important the precise process remain only in the possession of the organisation. Furthermore, due to the importance of the process, its integrity is of great importance. Any device firmware modification will be the result of significant testing and not done lightly. Therefore, the device firmware must only be modified with authorisation of the organisation.

Two threats to the security of the ICS components are attackers attempting to counterfeit the devices or sabotaging the system. Competitors of the organisation may try to copy the system to use themselves. Developing the ICS required significant investment, a competitor able to counterfeit the system may be able to produce similar results without the same investment cost. Another threat from attackers is the situation in which they attempt to sabotage the ICS by modifying device firmwares. This may be to downgrade the software or to use software from an alternative part of the line to cause problems. In these cases the attacker may be attempting to damage the organisation by reducing product quality or system performance.

However, if the firmware were securely bound to the devices these attacks would be prevented. Firmware taken from one device in the control system would fail to execute on any other device it was loaded on. Furthermore, and modified software or software created by an attacker would also fail to execute correctly. Therefore, the ICS integrity would be ensured while preventing any competitors from duplicating the system with counterfeit hardware.

3.1.1 Contributions

This chapter makes the following contributions:

- Identifying a new problem setting for binding software with hardware devices considering a powerful attacker who can access all device storage as well as duplicate hardware created from the same specification as the original (Section 3.2).
- Proposing a flexible technique to allow an application to be bound to a hardware instance and thus prevent unauthorised software from executing on the hardware device (Section 3.4).
- Describing several different security primitives which could be used in the proposed scheme in order to securely bind a software program to a hardware instance (Section 3.4.3).
- Presenting a prototype implementation of the proposed scheme that demonstrates how it may be used and at what performance cost (Section 3.5).

3.1.2 Structure

This chapter is structured as follows: Section 3.2 contains a more detailed examination of the problem considered in this paper. Section 3.3 contains a survey of previous work related to this paper. Section 3.4 contains the solution we are proposing to the problem considered in this paper. Section 3.5 describes the proof-of-concept implementation of the proposed solution that has been developed. Section 3.6 concludes the paper and describes the future work still required in this area.

3.2 Problem Description

This section expands on the previously described settings to formalise the problem considered.

3.2.1 Notation

The following notation will be used when describing the problem.

A	The attacker attempting to create counterfeit platforms.
x, y	Platforms comprised of a hardware and software element.
$SW(x)$	The software personalised to a platform x.
$HW(x)$	The hardware of the platform x.
$MEM(x)$	The memory of platform x. This includes both *persistent* and *non-persistent* storage.

3.2.2 Problem Scenarios

The scenarios considered are listed here.

1. The manufacturer of a platform has a hardware device $HW(x)$ and software which will be run upon it. The manufacturer wishes to personalise the software for use on the hardware device $HW(x)$, creating a device-personalised software $SW(x)$, which is loaded onto the device $HW(x)$. This process creates the platform x, the amalgamation of $HW(x)$ and $SW(x)$.

2. The platform manufacturer wishes to ensure only they are able to install software onto their products. The manufacturer is seeking to prevent a legitimate device $HW(x)$ from executing alternative software to the $SW(x)$ originally loaded onto x.

3. The manufacturer of a platform wants to ensure there does not exist an attacker A who can create counterfeit platforms. The attacker will seek to create a counterfeit platform y by purchasing a legitimate platform x, extracting the contents of $MEM(x)$ and loading them onto $HW(y)$, where $HW(y)$ has been built to the same specification as $HW(x)$. How A accesses hardware such as $HW(y)$ is outside of the scope of this work, however $HW(y)$ may be a device in the same product line as x but a different level of product as in the case study described in Section 3.1.

3.2.3 Attacker Model

The attacker A has the power to:

1. Read and copy the entire memory, $MEM(x)$, of a device.
2. Make use of software, $SW(y)$ taken from legitimate devices. The software A can access is software extracted from legitimate devices bound to the device it is taken from.
3. Make use of hardware built to the same specification as the device it wishes to counterfeit. The attacker can create a hardware for a counterfeit platform y such that $HW(y)$ is created from the identical specification as $HW(x)$ for a target platform x. Note, A does not have the ability to exactly copy intrinsic properties of $HW(x)$, only to create hardware according to the same design. Hardware intrinsic properties are described in Section 3.3.2.
4. Read and copy any data loaded onto any buses or other channels of x external to the CPU.

A has achieved its aim if it achieves either of the following:

1. The attacker creates a functioning, duplicate platform y which cannot be differentiated from a legitimate platform x.
2. The attacker installs a software $SW'(x)$ which can successfully execute on a hardware $HW(x)$, where $SW' \neq SW$. Where SW is the software loaded onto the device by the manufacturer.

3.2.4 Assumptions

It is assumed the application is suitably complex to render recreation non-trivial for an attacker. If the application was simple, the most practical approach for an attacker would be to write their own version. It follows that if developing a product equivalent to the original requires a significant investment from the attacker, this would make an attempted counterfeiting venture significantly less profitable.

We assume the attacker is unable to read the contents of any registers which are part of the CPU. The attacker is also unable to directly query any individual parts of the CPU. The attacker is able to access any of the memory of the device and observe data sent via buses, however the attacker is not able to see within the processor itself. This assumption is made because while bus analysis equipment can be purchased for less than $40 [3]; processor decapsulation and analysis requires more expensive, complicated and dangerous equipment [58].

A diagram of the embedded system being considered, and the elements of it accessible by the attacker, can be found in Figure 3.1. The grey portion of the figure indicates the "safe zone" unseen and inaccessible to A. As the grey area only includes the CPU, any security-sensitive operations or functions will need to be limited to this part of the system. This model of relying only on the security of the CPU matches that of Clercq et al. who also argue the need for security schemes to be implemented in hardware only and as a processor extension [32].

We also assume manufacturing of platforms is carried out in a secure location. The attacker is not able to choose, access or interfere with the software before it is installed. The attacker is only able to access software bound to a platform by the manufacturer.

3.2.5 Design Requirements

To prevent the attacker from achieving its goals, a solution to the problem must meet the following requirements.

1. **The proposed technique must grant the ability to securely personalise**

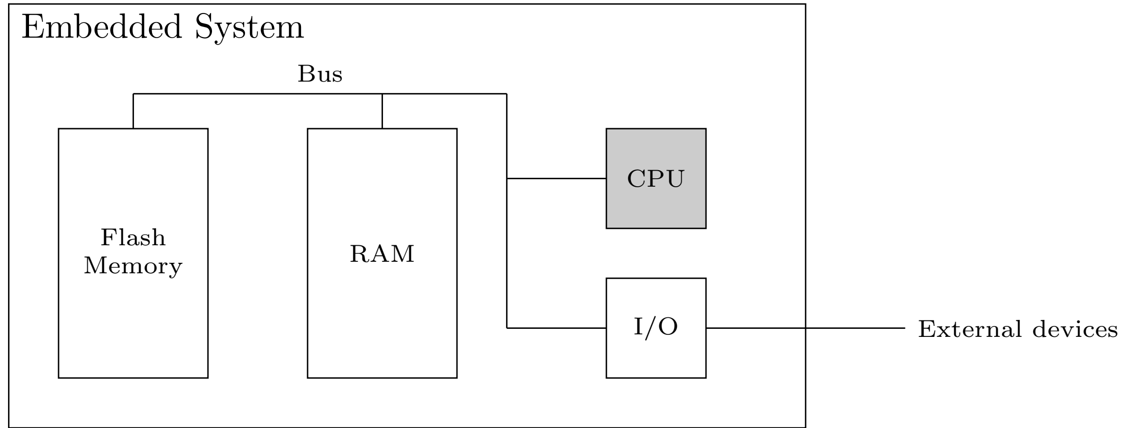


Figure 3.1: Hardware block diagram of the typical embedded system considered in this work. The grey area indicates the "safe zone" impenetrable by attackers.

software to a hardware device. The security of the personalisation shall ensure only the legitimate manufacturer is able to personalise their software. If only the manufacturer is able to personalise their software, this technique ensures only they may provision software to their products. The technique proposed must account for the fact that tamper resistance storage is not available.

2. **The software personalisation must make use of a hardware fingerprinting technique.** The proposed technique must allow the software running on the hardware device to be bound to the specific hardware instance it is personalised for. Personalising the hardware to a property intrinsic to the device would provide confidence that software on one device cannot be copied onto another. A property intrinsic to the hardware instance is required because tamper-resistant storage is not available.

3. **The solution must protect all data that is to be stored in any of the storage areas present on the device.** This requirement is necessary because the attacker is able to access any of the memory contents. If the data is ever unprotected in memory, an attacker could gain data unbound to the platform. Because the attacker can access all device storage, the only "safe zone" it cannot view or access is within the processor of the device.

3.3 Related Work

This section describes related work in the areas of preventing counterfeiting, unauthorised device modification and hardware intrinsic security. The first half considers different anti-counterfeiting and device integrity protection techniques in the context of the problem described in Section 3.2. The second half considers the use of physically unclonable functions for hardware fingerprinting.

3.3.1 Anti-counterfeiting measures

The piracy and protection of IP has received a large amount of interest as discussed below.

One technique for preventing device counterfeiting is to use a Trusted Platform Module (TPM) to ensure application integrity, securely store cryptographic keys or provide a secure boot process [93]. The cryptographic operations and storage of a TPM are attractive because they exist in a tamper resistant environment protecting data and preventing counterfeiting [119]. However, TPMs are not suitable for solving the addressed problem as they are included into systems as companion chips or co-processors. In the model considered in this chapter, this is unsuitable because the attacker (Section 3.2.3) could eavesdrop on communications between the TPM and CPU, preventing the TPM from adding the security required.

A software-based method for protecting IP is to use code obfuscation. Simply put, code obfuscation is the act of “jumbling up” a program in order to render a program very difficult to analyse/modify while retaining performance [24, 118, 5].

Code obfuscation could be applied to solving the problem posed in this paper as it would prevent the application data from being understood by an attacker. However, preventing attempts to analyse the program is not sufficient for our requirements. The attacker can copy all data from one platform and load it onto another. An obfuscated program copied from a legitimate device would operate on any device it was loaded onto as on the device it was taken from. Code obfuscation protects IP by preventing reverse engineering or third party modification of the firmware rather than unauthorised replacement of firmware. The difference between the goal of obfuscation techniques and the requirements in Section 3.2.5

leads to the conclusion that obfuscation is not suitable. Another concern with using obfuscation are the claims that secure obfuscation is not possible [12, 7].

Unlike the solutions described, our problem is one of controlling platform manufacturing to prevent unauthorised firmware modification and product counterfeiting. This problem has been considered by multiple authors including Kean who, in 2002, introduced the problem of a company providing licenses for a limited number of uses of a Field Programmable Gate Array (FPGA) IP core to a customer [60]. Kean's techniques allow for a different licensing model by moving from a large up-front fee to licensing per FPGA IP core. The protocol contained two main steps: providing the bitstream to the customer and loading the IP core onto the FPGA. The bitstream is encrypted by the company providing the IP core and then sent to the customer. A trusted programming software would be used by the customer to decrypt the bitstream and customise it using secrets known only by itself and the FPGA. The customised bitstream is then loaded to the FPGA [60].

However, this solution is not suitable because while Kean's scenario is related to that considered here, it is different. Firstly, Kean considers an IP producer selling uses of an FPGA bitstream to customers whereas, in this chapter, the entity programming the platform is the IP developer, not a separate entity. Secondly, this chapter is not merely interested in how a legitimate IP may be used, a solution is required to also prevent unauthorised modification to device firmware.

Another method for securing deployed software against attack is to use code signing. Code signing involves developers deploying applications with signatures to ensure their authenticity and prevent man-in-the-middle modification of executables. These schemes leverage the guarantees provided by asymmetric cryptography to secure applications from unauthorised modification. One parameter which can be determined by system integrators would be how regularly the signature would be verified against the application [29].

In the setting considered in this chapter, we could choose to check the signature before execution to minimise attack window. However, this would still allow attacks made against the system after the signature was verified. More regular checking could minimise these attack windows, but at a considerable cost due to the effort required to perform asymmetric cryptographic operations. A more serious subset of attacks that would not be prevented are those using attacker controlled hardware for execution as attackers are not motivated

to check signatures and could thus avoid the security provided. We are looking for an efficient solution that retains security by minimising the available attack windows and that is not easily circumvented. Therefore code signing is not a suitable solution for the problem considered in this chapter.

3.3.2 Hardware Intrinsic Security

Hardware intrinsic security encompasses several different areas concerned with trying to use or protecting against some of the physical properties of hardware devices. Topics include side channel leakage and random variations in hardware fabrication, however, only PUFs are described here for brevity.

3.3.2.1 Physically Unclonable Functions

Physically Uncloneable Functions (PUFs) were first introduced in 2002 by Gassend et al. [39]. A PUF can be considered as a fingerprint for an electronic device.

As described in Section 2.3, a PUF is a circuit which if two copies are constructed, then they will not have precisely the same behaviour. PUFs have been the subject of a large amount of study and many different designs of PUFs have been proposed [70, 112, 45, 80]. The security (or lack thereof) of PUFs is a question without a clear answer however some circuits have been proved to be more effective than others [59].

The potential for using PUFs for preventing counterfeiting was first described by Simpson and Schaumont in 2006. In their paper they described a technique by which hardware and software developers could combine their products with confidence that their IP is secured. PUF challenges and responses for the hardware device are stored with a trusted third party (TTP); using these and encryption, the hardware and software can authenticate with each other offline [110]. The work of Simpson and Schaumont was extended in 2007 by Guajardo et al., who simplified original protocol and removed the need for secure channels, preventing even the TTP from seeing the IP of the software developer [45].

The solutions produced by Simpson and Schaumont, and the later improvement by Gua-

jardo et al., controls the ability for an IP to be used on a piece of hardware. However their problem scenario is significantly different from that considered here which results in their solution being inapplicable to the problem. The authors are solving a problem wherein a system developer (SYS) wishes to use the IP of an IP Provider (IPP). These parties use a TTP for authentication to ensure genuine IP is loaded onto a genuine SYS's product. However, in our problem the SYS and IPP are the same entity so will not require a TTP for communication.

PUFs have also been applied to similar IP protecting problems, as by Guajardo et al. in 2008 [46], or by controlling the ability to manufacture a platform, as by Gora et al. in 2009 [44], with later work by van der Leest et al. in 2012 [121]. The paper of Gora et al. introduced the problem of binding a piece of Software Intellectual Property (SWIP) to a particular FPGA ensure the SWIP would only function on the intended FPGA [44]. Their technique consists of two main components: an enrolment phase completed in a trusted location, before the FPGA is delivered to the end-user and the operational phase when the device, deployed in the field, makes use of the protected SWIP. Firstly, enrolment consists of extracting the encryption key from the Ring-Oscillator PUF and using it to encrypt the SWIP via a custom encryption tool. This encryption tool operates by encrypting the application using AES and adding to it a security kernel. This security kernel is an unencrypted piece of software used to decrypt the application at execution time. It is necessary for the security kernel to be equipped with the challenge which is input to the PUF via a software interface to produce the decryption key for unlocking the SWIP. The operational phase is where the device has been deployed and is used, beginning with a secure boot process, decrypting the SWIP for later use [44].

The proposal by Gora et al. is a solution which effectively meets several of the requirements defined in Section 3.2.5. A SWIP is successfully bound to a HW device and the technique takes steps to protect the SWIP while it is stored on the device. However, there are some areas for improvement on their scheme. Firstly, the attacker described in Section 3.2.3 can access any part of device storage without restrictions on when the attacker can use its access. Against the scheme of Gora et al. the attacker could wait until the secure boot has decrypted the SWIP before copying it from the storage.

Another option for securely binding hardware and software was proposed in 2006 by Simpson and Schaumont as described in Section 2.4. Their scheme is similar to that of Kean,

also using a PUF to bind IP to a particular hardware instance. Binding the software to the hardware instance guarantees it cannot be moved onto other instances and execute as designed. In the Simpson and Schaumont solution, each software is registered with a Trusted Third Party who facilitates the exchange of key material between IP providers and hardware developers. However, like the Kean solution, it is designed to operate with more entities than we consider, and would therefore not provide an efficient solution. More seriously however, the scheme provides security guarantees only that IP is not moved from one device to another. It fails to protect hardware from unauthorised software installed by attackers and so is not suitable [110].

An alternative hardware-focussed solution was published by Zhang et al. in 2015 [125]. In their paper, the authors also proposed a mechanism for binding hardware IP to a particular FPGA instance. This allows sellers of IP blocks to protect themselves against counterfeiting while expanding their business model from expensive, unlimited licenses to offering single use licenses. Like the Kean proposal, the scheme uses a PUF to identify individual FPGAs and underpin the binding of IP to device [125]. Operationally, the scheme is similar to that of Kean [60]. Firstly, hardware devices are enrolled by registering Challenge-Response pairs (CRPs) and when developers wish to include the protected IP block in their design, they use registered CRPs to enable IP provisioning [125]. However, as with the Kean scheme, the Zhang et al. scheme does not provide an effective solution to the problem described in Section 3.2. In our setting, the parties producing devices and IP cores are the same, making it an inefficient solution [125]. Furthermore, the security of the Zhang et al. scheme is also questionable as highlighted by Bossuet and Colombier in 2016 [17].

One significantly different application of PUFs to binding hardware and software was proposed by Atallah et al. in 2008 [10]. They proposed a means of binding the output of an application to a single hardware instance and demonstrated their scheme with an implementation of RSA which could be personalised to produce correct outputs only when executed on a particular hardware instance [10]. This work is interesting but too specialised for the needs of this chapter. The Atallah et al. scheme successfully binds RSA, but required specialist knowledge of the algorithm to enable binding. We would prefer a more general purpose, algorithm agnostic solution and so Atallah et al. binding is not suitable for the problem considered in this chapter.

Finally, another approach to including PUFs in securing execution was published by Kleber et al. in 2015 [63]. Their paper presents a Secure Execution PUF-based Processor (SEPP) using a PUF generated key and AES counter-mode to encrypt instructions stored in device memory. The SEPP meets many of the requirements of the problem considered in this chapter, however due to it also offering secure execution guarantees it also incurs a significant performance overhead. We require a more specialised solution to the problem to preserve performance while protecting against attackers.

3.4 Proposed Solution

This section provides a description of the scheme solving the problem described in Section 3.2.

3.4.1 Design Overview

We consider the software ran on a platform x to comprise a set of instructions I_i, where i is the memory location storing the instruction I_i. The instructions comprising the program are assumed to be stored sequentially in memory without gaps. Different portions of code may be navigated to via control flow instructions (such as `jump` or `call` instructions). However, the code is stored in one unbroken portion of memory.

As stated in Section 3.2.5, we wish to personalise an application for a device. This requirement will be met by using a masking technique to conceal the instructions/data wherever they are stored on the device.

To meet the listed requirements (Section 3.2.5) we require the masking to be specific to the device. Each instruction will be masked by using a challenge-response function, F, yielding different masks for instructions and data based on the input challenge. F will be defined more precisely in Section 3.4.3. The masking function must also use a property or properties intrinsic to the device and will be as follows:

$$I'_i = I_i \oplus F(C_i).$$

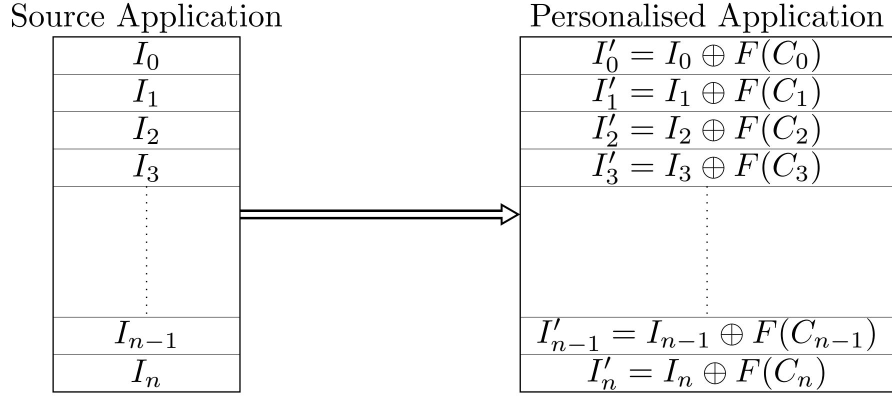


Figure 3.2: The transformation of source memory content into a personalised application

Where I'_i is the masked version of the instruction I_i and C_i is the challenge used in calculating the mask value. However, requiring challenges for each instruction adds two additional problems: how much storage will be required for the challenges and how will the challenges be chosen. To address these problems the following method will be used to assign challenge values:

$$C_i = I'_{i-1}.$$

This reduces the storage requirement for the masking challenges to one single challenge. The one challenge still required is C_0, the challenge value used to protect the instruction in the first location (location 0) in memory. There is no storage before the beginning of the memory and so a value needs to be assigned to C_0. C_0 could be generated by Random Number Generator (RNG) or Pseudo-Random Number Generator (PRNG), however, as it will be stored publicly, it does not need to be securely generated and may be chosen arbitrarily. A diagram of the transformation applied to the contents of memory when personalising an application is included in Figure 3.2.

An alternative approach would be to use the memory address as the challenge value, however this would introduce a security flaw in a multi-application setting. In a multi-application setting, each application will be loaded from virtual memory address 0 onwards. If applications are produced by different developers then each application could be vulnerable to attacks from the developers of the other applications. The attack would involve a platform x containing applications from two developers A and B. Developer A

would buy a legitimate product with both applications, A would then copy their own application from the platform. Using the source code for its application, A could recover the mask values concealing their application, the responses $F(0)$–$F(n)$ (where n is the length of As application as stored on x. Developer A could then recover the first n instructions of the application developed by B as it would have used the same challenges and responses. However, the proposed choice of challenge avoids the possibility of this type of attack.

3.4.2 Securing Mutable Data

The ability to secure data which must be stored on the platform, but may change, is a challenge for solutions to our problem.

Unlike when protecting application instructions, cascading previous masked values as challenges is not practical for data that may change. If the contents of one memory location were changed, the subsequent value would have to be updated to reflect the change in its challenge value. After the immediately following value was updated, the next value would have to be updated to reflect the different challenge stored. Updating values would continue until the end of the memory used by the application was reached. The large number of updates would add considerable overhead to storing data in memory.

One potential solution to this problem is to store a challenge value for each mutable value. This would result in an increased storage overhead for platforms using the proposed scheme. However, it would prevent changing a stored value from requiring updates to all following stored values, as previously described. With challenges stored, as well as the data, it is likely they will require regular updating to prevent data leakage. This approach would incur a large storage overhead as the amount of data to be stored would be doubled. Alternatively, mutable data could be split into smaller groups of values each allocated an extra challenge value. This technique would limit the amount of extra challenges required although reintroducing a limited version of the chained update problem described previously. This approach would allow a tradeoff between performance and memory overheads.

One remaining open question surrounding memory is how to handle dynamic memory requests/allocations at runtime. A similar technique to protecting other mutable data

could protect data in allocated memory. Extra challenges would be required for each dynamically allocated memory location. Any extra challenges required for dynamically allocated memory would need to be generated and used securely to avoid giving away information to attackers. Care will be needed to prevent applications accessing memory locations used to store the extra challenge values.

3.4.3 Choice of Function F

The central element in the technique proposed is the function creating masking values based on inputted challenges. The security of this element is critical to the security of the entire technique so it must be chosen carefully. The function used for F must be able to provide an output value based on an input challenge.

The output of F is required to be unpredictable because if output of F can be predicted from the input, attackers can recover the data stored. More precisely, the function F must be both collision and pre-image resistant. This will ensure attackers are unable to find alternative challenges yielding the same mask values. This must be prevented as any attacker who could find challenges producing the same mask values as challenges used could successfully modify the data or program instructions protected by the scheme violating Requirement 3. As the challenges are known to the attacker, if the function used is public, it must also use a secret value, not known by the attacker, when calculating the instruction mask value. This secret ensures only legitimate devices, possessing the secret, can unmask application instructions and data. If a secret function were chosen for F, the unpredictability would stem from the unknown nature of the function. However, this would be inadvisable as it would result in the platform being protected only by "security through obscurity".

There are several different candidates for F. For example, hash functions, block ciphers or PUFs could all perform as required for F. However, as each of these different options have significantly different properties they would need to be used in different ways to ensure the security of the proposed binding technique. Due to the different properties of the candidates for F, different amounts of extra information will be needed for each. The changes in required information will impact what secrets are required for each candidate function. Note that we also assume only public functions will be used for F and that

attackers will know the choice of F.

The remainder of this subsections will consider how the suggested functions may be used for F. The first type of function which could be used for F is a hash functions. Using a hash function does not address all requirements as it does not incorporate properties intrinsic to the hardware. If a hash function were to be used for F then an extra element would need to be added to use hardware intrinsicness in protecting the application. This extra element could be a value representing a hardware fingerprint of the device which could be mixed with each challenge before hashing. A PUF could be used to provide the secret, hardware fingerprint value which would be mixed with the input or output of the hash function. PUFs, as defined in Section 2.3, are Unique and Uncloneable, making them effective methods of key generation while removing the need for key storage (see Section 2.3.2).

Block ciphers could also be used for the function F. A secure block cipher would ensure only an entity knowing the secret key would be able to calculate the mask values used to conceal the application. However, similarly to using a hash function simply, a block cipher alone will not meet the requirements as there is no included hardware intrinsic property/ies. A simple method for incorporating hardware intrinsicness could be to use a PUF to provide the key used by the block cipher. This would ensure duplicate hardware constructed from the same specification would not possess the binding key of a different device.

A drawback to using a block cipher or hash function for F is it would result in a large number of encryptions/hashes being performed. If each instruction were masked separately then the cost may be severe. Grouping instructions into basic blocks may be considered to reduce the number of encryption/hash computations required.

A third type of function which could be used for F is a PUF. PUFs have many properties which make them useful in solving the problem posed. Firstly they are an element of hardware which, by definition (see Section 2.3), will be Unique per device. If a PUF is included in the platform, it will ensure that *exactly identical* copies of hardware cannot be made due to the Uncloneable property defined by Maes and Verbauwhede [81]. By binding the application to the PUF, the software found on a platform will only function on that particular platform. One challenge of using PUFs is many are noisy and have to

be combined with fuzzy extractors (or similar) for their output to be usable, significantly impacting the rate at which outputs of the PUF may be accessed [45, 79].

Of the suggested candidates, only a PUF would incorporate hardware intrinsic behaviour into the function F directly, however the performance would not be sufficient for an instruction level masking. When considering other choices of F there is a need for some information to remain secret, e.g. encryption keys. If a PUF were chosen, their unclonable nature ensures neither F nor any other information must remain secret. The definition of a PUF states that simply knowing the design of the circuit and the input used is not enough to predict the output. The PUF would be incorporated into the design of the CPU and so, by our assumptions, it cannot be queried directly by the attacker. Therefore, the most efficient solution would be to use the PUF to provide device specific information for a cryptographic function included in the CPU. This would provide the function, F, needed to securely bind the application to the device according to the requirements in Section 3.2.5.

3.4.4 Implementation Considerations

We have proposed a technique to securely bind a piece of software to an individual hardware instance. However, there are several factors which would need to be considered when implementing this technique on a platform as well as the choice of function F.

Firstly, the masking and unmasking of operations/data would have to only exist in the CPU to prevent the scheme from being bypassed by an attacker. The unbinding must be a step in the execution process; the operation/data is unmasked as it is loaded from memory, into the CPU, before it is executed/used. If the masking is specific to a device and unmasking is a step in execution then programs copied from one hardware to another would fail to execute correctly. An attacker will have to remove the binding from a piece of copied software before it can execute correctly on a different piece of hardware.

Secondly, when choosing the function F it will be important to consider how to use the function securely and efficiently. For example, if F is chosen to be a block cipher with a 128 bit block size. If the architecture protected uses 32 bit instructions then it will likely be more efficient to protect four instructions per challenge instead of one. Combining

instructions may be required for the security of the scheme too. If 32 bit instructions were protected individually then each output of F is only one of 2^{32} possibilities. However, the number of possible outputs of the function increases rapidly if more instructions are protected simultaneously making the output one of 2^{64}, 2^{128} or more possibilities.

A further consideration implementing the scheme is when and how the masking would be applied to the software. One approach would be for software to be loaded to the device already masked. However, this would require the manufacturer to interrogate the device to learn about the hardware intrinsic properties of that hardware instance. If this involved using an extra, personalisation, circuit it might introduce a new attack surface, it might also add a significant delay in the manufacturing process. Alternatively, the device could add the masking when the software is loaded to it. This technique would prevent the manufacturer ever knowing the secret information of any particular hardware device which may prevent insider attacks. However, for the device to personalise itself it may require additional hardware increasing design complexity and the cost of producing each device.

3.4.5 Solution Analysis

A technique for preventing the counterfeiting of devices has been proposed in preceding subsections. This section examines how it meets the three requirements in Section 3.2.5.

1. **The proposed technique must grant the ability to securely personalise software to a hardware device.** The proposed technique provides the ability for a manufacturer to create a device-specific masking to bind software to hardware devices. F is used to provide device-specific mask values which are mixed with instructions and data stored in device memory. Several candidate functions were proposed allowing for different, secure personalisations for different devices. The suggested functions for F were described in Section 3.4.3 and suggestions for how they would be used were given. The security of the scheme was considered against two different attackers and an argument for its security was included in Section 3.4.6.

2. **The software personalisation must make use of a hardware fingerprinting technique.** Of the three candidate functions proposed for F, only a PUF provides an

adequate inclusion of hardware fingerprinting by definition. However, the alternative functions could include the use of a PUF to provide a hardware fingerprint. This may be preferred if a PUF cannot be found to satisfy the unclonability requirement. A key programmed into the silicon of the CPU could be used to provide a device specific secret, however this would not provide the hardware intrinsicness required. Methods for including hardware intrinsic features when using block ciphers or hash functions were suggested when describing the candidates for F in Section 3.4.3.

3. **The solution must protect all data that is to be stored in any of the storage areas present on the device.** The proposed technique would be implemented as an extension to the pipeline of instructions and data loaded into the processor. Adding the technique as a pipeline stage allows instructions and data to be masked/demasked as they enter/exit the processor. In this setting there would be no data stored in memory in an unprotected manner.

3.4.6 Security Evaluation

This subsection will evaluate the security of the solution presented.

The proposed scheme uses an XOR operation to combine instructions with generated mask values. Due to the use of XOR, this scheme will remain secure only while two of the three elements involved (I_n, $F(C_n)$ and I'_n) remain secret. The values of I'_n are stored in memory and will be available to the attacker. The values of I_n are the values sought by the attacker; these are only calculated within the CPU where the attacker cannot view them. To recover the values of the instructions I_n, the attacker must find or deduce the values of $F(C_n)$. The results of the function F on the values C_n are only calculated within the CPU, therefore, by assumption, the attacker is unable to read them. Therefore, the security of the scheme is based on the strength of the function F.

The attacker described in Section 3.2.3 has the power to access all of the storage, $MEM(x)$, of a platform, x. Most of the challenges used with the function, F, are the masked instructions known to the attacker. The first challenge, C_0, is also known to the attacker. It is therefore required that knowing only the input to F does not allow its output to be predicted. If the output of F is predictable then, using challenges, C_n, the attacker can recover instructions, I_n. If the attacker is unable to calculate the values $F(C_n)$, it is

unable to either create a correct mask for its own application or unmask the instructions of an existing application.

An attacker who is the developer of one of multiple applications loaded onto a device x is now considered. This attacker has access to all the masked instructions I'_n and their corresponding challenges C_n as well as a subset of the values for I_n which are the instructions of its own application. The developer-attacker uses its application and its masked version recovered from x to recover a set of corresponding C_n and $F(C_n)$ pairs. This poses a risk of the developer-attacker unmasking a portion of any other applications found on the device x using the $C_n/F(C_n)$ pairs. Similarly the attacker might be able to use its knowledge to create the correct masking for any application of its choice. If the CPU were a 32 bit processor then the maximum application size would be 2^{32} addresses; however it is unlikely for an embedded application to be so large. If the applications were much smaller than 2^{32}, such as in the embedded setting, the chance of challenges for instructions of one application overlapping with instructions of another application significantly decreases. If this scheme were used in the multi-application setting, a maximum application size would need to be carefully set to minimise the change of significant challenge-space overlap. A countermeasure against challenge-space overlap would be to mask multiple instructions at the same time. If two operations were masked at once, the challenge space for a 32 bit processor would increase to 2^{64}, lowering the chance of challenges from applications overlapping one another. Overlapping of mask values used for binding software to hardware can also be decreased by ensuring no two applications on a platform use the same C_0.

3.5 Implementation

A prototype implementation has been developed to demonstrate the viability of the scheme proposed in Section 3.4. The prototype also allows the performance impact of the scheme to be analysed.

The first half of this section describes the implementation including a system diagram of how the scheme was included into an example embedded system. Also described are the three different transitions possible between operations and how they are addressed by the prototype.

The second half of this section analyses the prototype implementation, describes the overheads of using the proposed scheme and the costs associated with different choices for the function, F.

3.5.1 Development Platform

The prototype implementation was designed and implemented on a Xilinx Spartan-6 FPGA SP601 evaluation board. VHDL was used to describe the circuits for the Unmasking Unit and other components in the prototype. The VHDL was synthesised for the Spartan 6 using the Xilinx ISE design suite. An Apple Macbook Pro was used for running the ISE design suite and synthesis tools as well as the terminal program PuTTY that read the serial output data transmitted by the prototype system.

The prototype implementation was developed for the FPGA board to create a real system operating using the proposed countermeasure. The implementation is similar to a normal embedded system, it contains a processor, a ROM that stores the application and IO devices. The IO devices used by the prototype are two LEDs and the serial output port of the evaluation board. The serial port functionality is used by the prototype implementation to output to a terminal program running on a laptop computer.

The processor included in the prototype system is the PicoBlaze^{TM} open soft-core processor developed by Xilinx and made freely available online [123]. The PicoBlaze^{TM} is attached to one of the block RAMs in the FPGA that is used as a ROM to store the program executed by the prototype. Example VHDL code for using the UART serial interface with a PicoBlaze^{TM} processor is provided with the processor and was used to connect the serial output to the system. A diagram of the prototype system is included in Figure 3.3, the Unmasking Unit is the element developed to implement the proposed scheme.

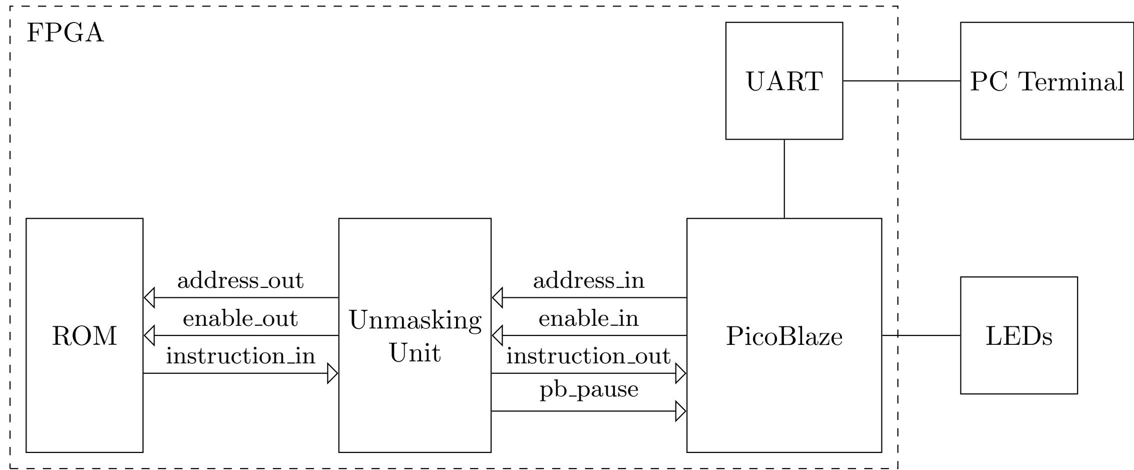


Figure 3.3: Diagram of implemented system.

3.5.2 Demonstration Application

An example program was installed onto the prototype system to use the hardware devices attached to the PicoBlaze™ system (Appendix A). . This enabled the prototype to give an obvious indicator of it was executing correctly. The example program was written in PicoBlaze™ Assembly Code and comprises two different stages of execution. Firstly, the application runs through a set of instructions using the UART serial output to transmit "Hello World!" to the laptop terminal application. This involved loading the characters to be outputted into a register and outputting the value in the register to the UART data transfer component. Serial output from the prototype requires 2608 cycles to transmit one byte to the laptop, because of this it contains a buffer to store the data to be transmitted. However, the UART components buffer for data to be outputted over the serial line can store only a limited amount of data. Therefore, before sending data to be transmitted, the application checks the UART buffer to ensure it is not full before adding the character to be output. This check is carried out by calling a simple procedure that reads from the UART transmit component and waits until the full flag is '0' (indicating that the buffer is not full). Once the UART buffer is not full the procedure finishes by executing a `RETURN` instruction.

The second half of the application ran on the prototype system increments a series of counters to turn two LEDs on the evaluation board on and off. This part of the application comprises an infinite loop that repeatedly turns the two LEDs on and off. To ensure the LED flashing can be observed with the human eye a significant delay is required between each changing of the status of the LEDs. Three counters are used to create three loops

with one loop executed inside another loop inside the third loop. Values of the two most significant bits of the third, outermost counter are assigned to the LEDs resulting in them following a off-off, off-on, on-off, on-on sequence.

3.5.3 Unmasking Unit

The scheme, as proposed in Section 3.4, was implemented using an Unmasking Unit positioned between the PicoBlaze^{TM} and the ROM components. For the prototype system the contents of the ROM were changed outside of execution and the masked values were loaded as the FPGA was programmed. The Unmasking Unit was developed to relay address, enable and instruction values between the PicoBlaze^{TM} and the ROM during execution and also to forward extra values in the case of the execution of `JUMP` or `CALL` instructions. The Unmasking Unit also calculates the mask ($F(C_x)$) values and on receipt of the masked $I'x$ instructions from the ROM it performs the required XOR operation before passing the resultant unmasked I_x instructions. The last function of the Unmasking Unit is to pause execution of the PicoBlaze^{TM} processor when required (this is described and explained in more detail later in this section). Due to the Unmasking Unit being entirely responsible for latching/fetching the masked instructions I'_x, the PicoBlaze^{TM} and FPGA RAM modules did not require any modification before being used in the prototype system (other than the data being stored in the RAM being changed). A diagram showing the inclusion of the Unmasking Unit is included in Figure 3.3.

The scheme implemented by the Unmasking Unit behaved as described in Section 3.4. A function of the contents of the previous memory address is used as a mask value for a memory location. The function used in the prototype is a simple Linear Feedback Shift Register (LFSR). The LFSR calculates the mask value $F(C_x)$ by setting the LFSR state to C_x and then generating three bits for the LFSR according to the taps used. The state of the LFSR after generating the three bits is used as the value $F(C_x)$ which is XOR'd with the masked instruction, I'_x, to reveal the instruction to be executed, I_x.

The Unmasking Unit developed comprises three main processes and two much simpler pieces of logic controlling the `enable_out` and `address_out` ports. During execution of an application on the prototype system there are three situations the Unmasking Unit may encounter. Firstly is the case of normal instruction execution: an instruction I_x is currently

being executed and the next instruction to be executed is the following instruction I_{x+1}. In this case the Unmasking Unit will need to have stored the value I'_x such that I_{x+1} can be unmasked using $F(I'_x)$.

The second possible situation for the Unmasking Unit is that instruction I_x is a `JUMP` or `CALL` instruction. PicoBlaze™ assembly code includes conditional and unconditional branch instructions, however both treated in the same way by the Unmasking Unit. In this situation the next instruction is either I_{x+1} (if the branch is not executed) or it is a different instruction I_y. To prepare for the possible branch instruction two features of the PicoBlaze™ are exploited by the Unmasking Unit. The PicoBlaze™ requires two cycles to execute an instruction, however the RAM can be accessed in only one cycle. To save power, the PicoBlaze™ only enables the RAM when it is in the second cycle of executing an operation, this is also when it sends the address of the next instruction to the RAM. Therefore there is a "spare" cycle available for the Unmasking Unit to use to load extra data it may require from memory. In the example suggested previously, this extra value would be I'_{y-1} and this would be read from memory in the "spare", first, cycle of the execution of I_x. To know the correct address for I'_{y-1} another feature of the PicoBlaze™ is used. Most `JUMP` and `CALL` instructions include the address to be jumped to in the opcode; `0x36031` is the code for `JUMP NZ 031`, if the `ZERO` flag is not set then jump to address `031`. The address that may be branched to is therefore known to the Unmasking Unit allowing it to read the extra data it may need for unmasking the next instruction before the address of the next instruction required is known.

In the case of a `JUMP` or `CALL` instruction the enable and address values from the PicoBlaze™ are not forwarded to the RAM. During the first half of the execution of an instruction the enable signal is 0, however the Unmasking Unit must transmit a 1 for the extra memory access. The address value from the PicoBlaze™ will still be the address of the current instruction, however the Unmasking Unit will transmit an alternative value for the extra memory access.

A combinational process is used to handle changes on the `instruction_in` port of the Unmasking Unit. This process uses the calculated mask value to unmask the instruction read from the ROM. A sequential process, triggered by the system clock, latches the `instruction_in` values and calculates the mask values used. The instruction latch process uses the enable signal to determine if the first or second half of the instruction is being

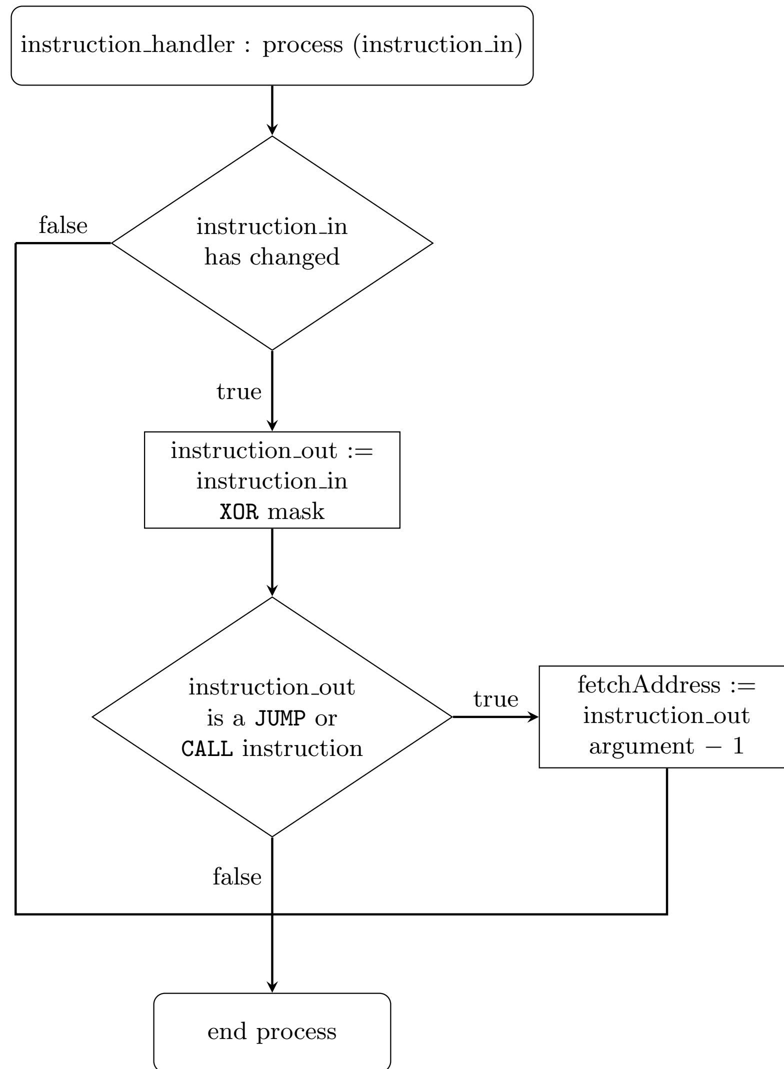


Figure 3.4: A flowchart of the instruction_in handling of the Unmasking Unit.

executed, this is used when deciding if the value read from memory will create the next mask value or not. Diagrams of the operation of the combination `instruction_in` handler process and the clocked instruction latch process are found in Figures 3.4 and 3.5.

The third scenario the Unmasking Unit may encounter is when I_x is a `RETURN` type instruction or either a `JUMP@` or `CALL@` instruction. In these situations the address of the next instruction I_y is not known until halfway through the execution of I_x when the PicoBlaze™ requests the next instruction. Before I_y is executed, the Unmasking Unit will need to calculate the appropriate mask value, $F(I'_{y-1})$, which will require a cycle before I'_y can be read and unmasked. Therefore, the execution of the PicoBlaze™ must be paused for one cycle to allow the Unmasking Unit to read the necessary data from memory and calculate the mask value required before loading, unmasking and outputting the next instruction. Therefore, the Unmasking Unit includes an extra process responsible

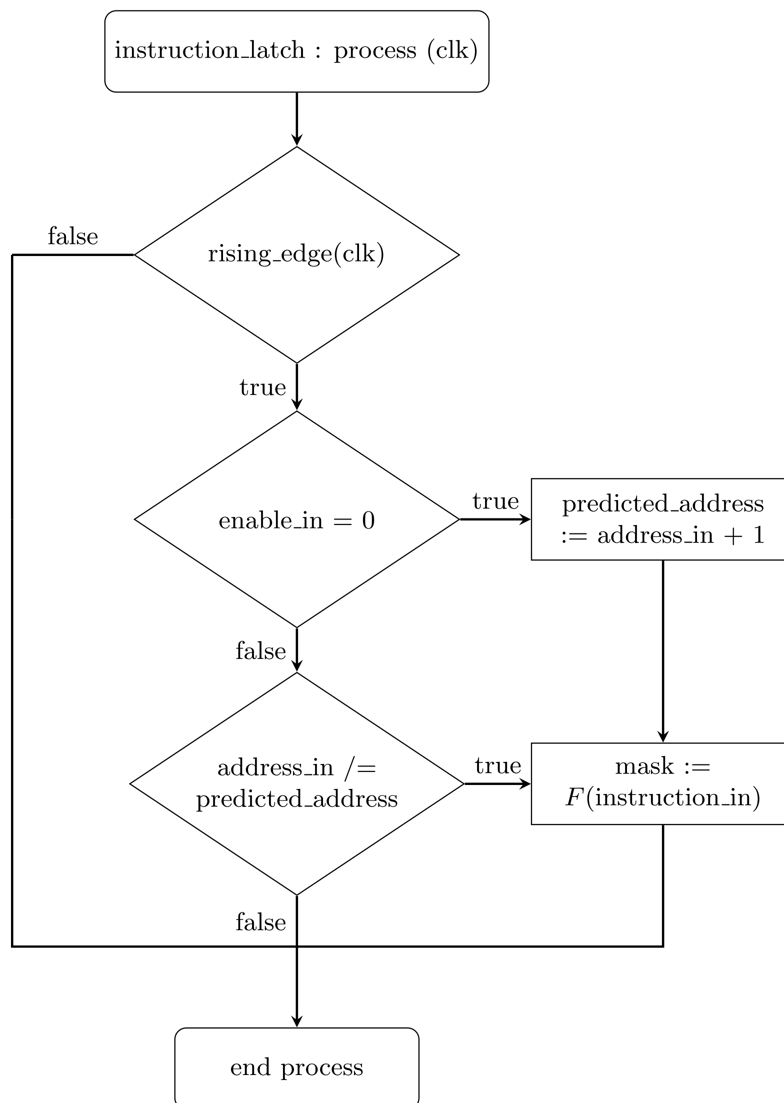


Figure 3.5: A flowchart of the calculation of the instruction mask values.

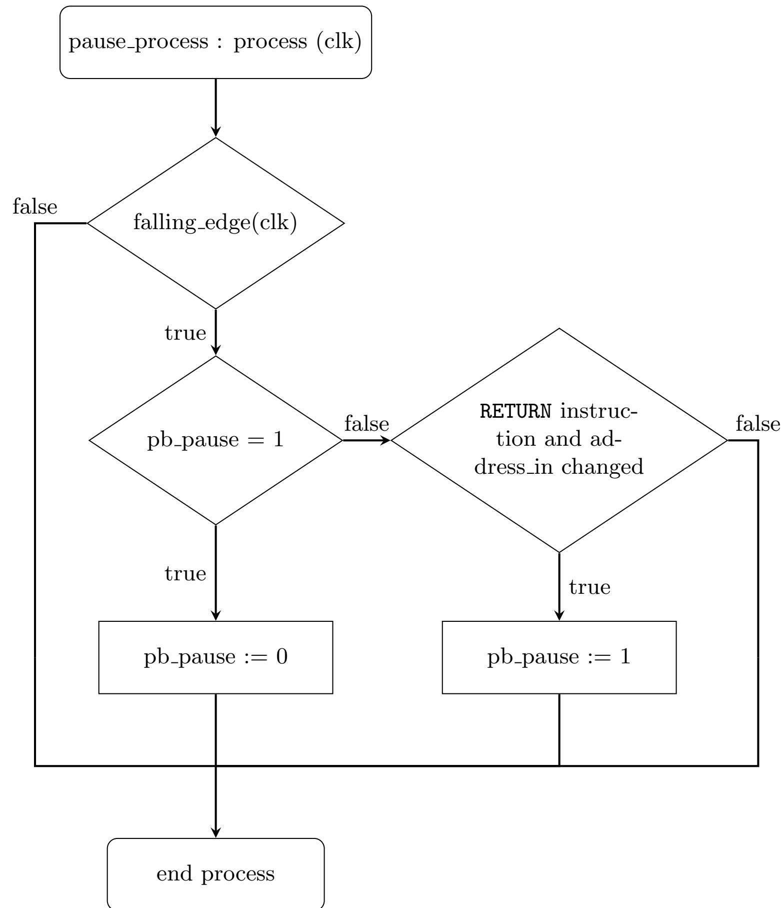


Figure 3.6: A flowchart of the operation of the PicoBlaze™ pause process.

for detecting `RETURN` instructions and pausing the processor when required. A diagram of the operation of the pause process is included in Figure 3.6.

3.5.4 Performance Analysis

This subsection analyses the performance of the prototype described and examines how other functions may be used in the scheme and the costs resulting from using different functions.

As described, the scheme has a performance impact on only one situation faced by the PicoBlaze™. Out of the 69 different instructions supported by the PicoBlaze™ processor only 8 of the instructions require any delay at all before they can be executed. Furthermore, as it is only `RETURN` and the `JUMP@/CALL@` instructions that are slowed down by the scheme the impact on execution performance is likely to be minimal. The affected instructions are likely to be a small subset of the instructions which comprise a program. In a program there

cannot be more `RETURN` instructions executed than `CALL` instructions. It is also reasonable to assume arithmetic, logic, input/output and loading/storing instructions make up a larger amount of any applications executed than the `RETURN` instructions. Considering the prototype system, the example program is comprised of 65 instructions, including only one `RETURN` instruction (although there are 14 `CALL` instructions). The procedure that is called is made up of 5 instructions including the `CALL` and `RETURN`. The impact of the scheme was to increase the execution time from 10 cycles to 11, an increase of 10%. However, the procedure that is called is part of the first half of the application that, including the block called, is made up of 52 instructions. Allowing for the procedure being called multiple times, the first half of the application requires 104 instructions to be executed, completing in 208 cycles. The first part of the application would execute in 222 cycles with the proposed scheme, therefore the actual performance penalty was only to increase execution time by 6.7%.

The prototype has been implemented using an LFSR as the function, F. However, the proposed scheme was designed to be able to use different functions for F, several candidates were described in Section 3.4.3. As described, the performance impact of using the scheme in the prototype was small. This was mainly due to exploiting features of the PicoBlaze^{TM} to prevent processor execution being delayed. If the function required two cycles then the performance impact of the scheme would be greater as any `JUMP` instructions would also cause delays. Currently the scheme has 2 cycles for computation available in the first situation described in Section 3.5.3, 1 in the second and none in the case of `RETURN` instructions. Therefore if the function required 2 cycles to compute the mask value then there would still be no added delay if the next instruction is the next in memory. The delay from the PicoBlaze^{TM} executing a `JUMP` would be 1 cycle, and a `RETURN` would require the processor to be paused for 2 cycles.

3.5.5 Security Analysis

The prototype was developed to demonstrate the viability of the proposed scheme, however, it does not completely meet the security requirements listed in Section 3.2.5.

Firstly, the prototype binds the software to the hardware by using an LFSR for the function F. This use of an LFSR meets neither the first or second requirements described in

Section 3.2.5. The first requirement demands the personalisation be secure, however the attacker is able to emulate the operation of the LFSR and calculate the mask values used because the prototype does not use any information the attacker would not know.

Requirement 2 requires the software personalisation to use of a hardware fingerprinting technique. This has not yet been implemented on the prototype and is a topic for future work. Crefsubsec:binding-choice-of-f stated a PUF would be required to provide hardware-specific information for implementations of F. Therefore, the prototype implementation is limited by this being absent. However, Section 3.4.3 also stated how PUF output could be used to provide key material for block ciphers or other cryptographic algorithms. This key generation would only be performed at start up; afterwards the key would be used to unmask application instructions. Therefore, in most cases, the performance and viability of the scheme is limited by the cryptographic operation/s included in F and not by the PUF used. If a PUF were only included to provide key material, the prototype still gives accurate estimates of the performance and efficiency of the proposed scheme. In the case where a PUF is chosen as the entire masking function, F, once again the prototype allows the performance of the developed system to be estimated by comparing the cost of the error correction/fuzzy extraction against the constraints described in Section 3.5.4.

The third requirement was the solution must protect all of the device data. This has been met because the only data stored on the prototype is the application data that has been bound to the implementation. The current version of the Unmasking Unit does not have the ability to mask/unmask data which is stored/loaded in the device memory. Therefore, if any data were saved to or read from the memory it would currently be unprotected. However, the prototype has demonstrated that the proposed technique may be used for protecting instructions stored in memory and so could be applied to storing data in memory. The current Unmasking Unit is able to access areas of memory without disrupting execution of the process (as is used for unmasking instructions after a jump). It may be possible that the extra loading from memory can be completed without significantly affecting the execution of the PicoBlaze$^{\text{TM}}$. However, this is a problem to be addressed in future work.

3.6 Conclusion

This chapter has presented a novel technique binding applications to the platforms they are installed onto. The proposed technique can be used to prevent unauthorised modification of device firmware by ensuring applications cannot be copied from legitimate platforms and then executed on other devices. Hardware intrinsic properties of the device can be used in the binding to personalise the software to particular hardware instances. A proof-of-concept implementation of the scheme has been presented including the proposed scheme and demonstrating it can be deployed with only a small performance overhead.

The solution proposed securely binds software to a hardware device and protects the data stored on the platform wherever it is stored. The binding is secure against even powerful attackers capable of reading any storage outside of the CPU.

Further work on the scheme would be to consider how firmware updating could be performed if this scheme were to be used. Provisioning a different version of the firmware for every single legitimate device might not be achievable and so a specific update protocol may be needed.

Chapter 4

Installing and Updating Software with Hardware-Software Binding

Contents

4.1 Introduction **70**

4.2 Problem Description **72**

4.3 Models for Software Provisioning **75**

4.4 Related Work **80**

4.5 Proposed Solution **84**

4.6 Analysis **87**

4.7 Conclusion **92**

Chapter 3 considered how to create a secure bond between hardware and software. This chapter takes that idea and examines the consequences of deploying such a scheme. Specifically, how to provision software and updates when a hardware-software binding scheme is being used. To study this problem, a smart city case study is used to elicit the provisioning models that apply and the requirements that result from them. Three protocols for distributing software are presented that meet the requirements stated. A formal analysis using Tamarin Prover is described proving the security of the proposed protocols. Finally, an implementation has been developed using a laptop and Raspberry Pi 3 to demonstrate the proposed protocols in action and their performance.

4.1 Introduction

As discussed in Chapter 3, binding hardware and software allows the creation of a dependency between hardware and software in a product. However, it makes provisioning software to remote devices much more challenging because, by definition, it limits creating software to execute on a device. Therefore we need to work out how to provision software when hardware-software binding is being used.

To explore provisioning software with hardware-software binding, this chapter attempts to address the problem of securely provisioning software to devices in a “Smart City”. “Smart Cities” are a concept of interest to many groups including local and national governments, political unions and academics [34, 36, 51]. Smart cities are an approach to urban development that includes smart concepts and technologies to provide more effective services to the people who live or work in the city.

The UK department for Business Innovation and Skills (BIS) lists the following services that they wish to enhance in smart cities: Intelligent transport systems, Assisted or independent living, Water Management, Smart grids or energy networks and Waste Management [34]. These areas are all part of critical infrastructure so their availability and security are vital. Frequently, government systems such as healthcare, energy grids, water grids and other critical infrastructure systems transmit sensitive information that must be kept confidential. Transport management systems must also be secured because problems can cause significant delays or even physical harm to people. For example, interfering with traffic lights could easily cause an accident. With many risks in the smart city setting it is highly important that the networked devices be secured. In these settings it is also critical for devices to behave as expected. Protecting devices from software tampering ensures they behave as designed. However, smart cities comprise many types of devices in a wide area that require software and software updates to be provisioned in the field instead of relying on secure manufacturing to prevent software compromise. A further complication is that the hardware may not be managed by the manufacturer of it but by a third party contracted by the organisation that owns/manages them.

The smart cities model can be abstracted into a theoretical model that also applies to other scenarios. A smart city can be considered as a wide network of computing devices that perform tasks and regularly communicate with a central server/s. In many large

networks it may be impractical to rely on physical access to devices to change or update software. However, it is often important to ensure software can only be loaded by authorised parties. Examples of such networks include: industrial control systems, sensor networks and vehicular networks.

4.1.1 Contributions

The main contributions of this chapter are:

1. Modelling the security of provisioning software to large numbers of remote hardware devices (Section 4.2).
2. Proposing a set of requirements for securing software provisioning (Section 4.2.3).
3. Discussing the different potential models for managing the security of software provisioning (Section 4.3).
4. Proposing a protocol to solve the problem of provisioning software with hardware-software binding (Section 4.5).
5. Providing a security verification of the proposed protocol performed using Tamarin Prover (Section 4.6).

4.1.2 Structure

This chapter is structured as follows. Section 4.2 contains an overview of the problem considered in this paper. Section 4.3 explores the possible models that could be used for software provisioning and draws the security requirements for a solution to the problem considered. Section 4.4 describes the related work in this area. Section 4.5 contains the solution proposed for securely provisioning software and updates to a network of devices. Section 4.6 analyses the solution proposed against the requirements set out in Section 4.3. Section 4.7 concludes the paper and describes the future work required in this area.

4.2 Problem Description

This work considers how to securely provision software to devices in a wide, smart city network. This work will consider a simplified problem including just one entity attempting to install software onto devices in a network belonging to a particular owner. This section explores the motivations of the entities involved and describes the attacker who seeks to undermine the security of the system. Finally, this section lists and describes the requirements for solving the problem posed.

4.2.1 Entities and Motivations

Provisioning software to devices includes four individuals: the Device (D), the Operator (O), the Manufacturer (M) and City Hall (CH). D wants access to an application provided by O. O is responsible for operating the Devices and is interested in loading their application onto them. However, O wants to control the spread of their application. M is a manufacturer of Devices who wants to create Ds able to securely receive and store software. CH is the organisation that owns the Devices and has contracted their operation to O.

The Device (D) is a generic IoT device that runs applications. D may be a set of traffic lights or a water valve in a treatment system that is loaded with an application from the operator/manager in order to let traffic/water flow. Alternatively, D could provide information back to a central computer system that is used to monitor flow through the network of pipes, wires or roads. D may be deployed in a public area that is considered to be an insecure environment where attackers have physical access to it.

The Operator (O) is a business or government agency contracted with operating or managing some or all of the infrastructure in the smart city such as the road or water network. To operate D, O loads it with software to dictate its behaviour and communications. Producing the software has required a large investment that O wishes to protect. Furthermore, as there is a risk to citizen safety if D is interfered with, protecting device software from tampering benefits O by preventing loss of revenue and any reputation damage caused by incidents.

The Manufacturer (M), produces hardware devices that are installed in smart cities. Different companies may need to manage the devices over their lifetimes, therefore they need to be able to be personalised by different Operators. M wishes to ensure their devices provide protection to the software installed and to provide guarantees as to the security of the software provisioning. If malicious parties replace the device software with their own this may violate citizen privacy or lead to accidents, injuries or lives lost. It is advantageous for M to produce Ds that guarantee the security of installed software because it will make them more desirable to city planners.

Finally, City Hall (CH) is the government agency or department that holds overall responsibility for the smart city. This party is the owner of the Devices and is contracting O with running some city infrastructure. In practice, CH may be an IT department, a centralised organisation or an office of the regional or local government. However, for the context of this work the device owner is called City Hall.

4.2.2 Attacker Model

This chapter considers an Attacker (*Att*), attempting to interfere with the Devices, with two main goals. The first goal of *Att* is to steal the Software (SW) installed on the Device, this may be later used to produce duplicate Smart City devices without authorisation. Secondly, the Attacker wishes to force smart city Devices to execute their own (potentially malicious) software instead of that installed by O.

As stated previously, Ds in the network may be in public places that *Att* has physical access to. This physical access allows *Att* to read the contents of Ds long term storage and RAM. However, *Att* is not able to read from any secure hardware present on D such as a secure element, TPM or HSM.

Communication between CH, D and M or O uses either wired or wireless connections. *Att* has the ability to eavesdrop on communications between these parties and send messages to any of these parties. Wireless communications can be easily overheard and created by an attacker so it is reasonable for *Att* to send and overhear messages to D. Furthermore, in the wired setting it is also reasonable for *Att* to hear and send messages as they have physical access to the device and so could tap any communication cabling.

4.2.3 Protocol Requirements

Based on the model described in Section 4.2.1 we propose the following requirements for a secure software provisioning scheme.

R1) **Entity Authentication:** All parties who receive or send applications must be authenticated to prevent application leakage and installing applications from unknown developers.

R2) **Replay Resistance:** Preventing message replays is required to prevent application leakage via overheard authentication messages.

R3) **Perfect Forward Secrecy:** Leakage of a session key must not allow attackers to access, deduce or compromise any sessions keys used in future protocol runs.

R4) **Transferred Software Integrity:** Guarantees will be provided that ensure the correct transfer of the application.

R5) **Transferred Software Confidentiality:** The application must only be transferred encrypted to prevent leakage.

R6) **Secure Software Binding:** Software must be securely bound to the device it is installed on to ensure it cannot be transferred to other devices.

R7) **Compulsory Software Personalisation:** The Device must only execute software provisioned be the authorised O.

R8) **Secure Masking Agreement:** Both the Device and Operator will contribute to securely establishing keys for application masking.

R9) **Secure Key Establishment:** Generation and use of session keys will ensure message and application confidentiality.

R10) **Single Operator:** Devices must only accept software from the, at most one, Operator currently authorised to manage the Device.

R11) **Operator Handover:** It must be possible for the Operator of the Device to change.

Requirements R1, R2, R4 and R5 are concerned with ensuring software confidentiality and that D will only accept legitimate software. Requirement R1 ensures O only transfers software to legitimate devices and D only accepts software from the authorised O. Similarly,

Requirement R2 ensures applications cannot be leaked to *Att* using captured messages between D and O. Requirement R5 prevents unauthorised installation of the software onto other Ds by ensuring it is protected when transferred to D. Finally, Requirement R4 allows the Device to check the software has not been tampered with in transit.

Requirements R2, R3 and R9 ensure the security of the software provisioning protocol. As stated in Section 4.2.2, *Att* is able to eavesdrop on communication between D and O. Requirement R2 ensures old versions of software cannot be reinstalled onto D and previous authentications cannot be reused by *Att* by replaying old messages. As a critical infrastructure component, D has a long lifetime including software updates, Perfect Forward Secrecy of software provisioning is required to ensure old keys do not leak latest software versions (Requirement R3). Requirement R9 ensures attackers are unable to predict the keys used for securing protocol messages and transferred software.

Requirements R6 to R8 protect the software stored on D. Securely binding the software to D (Requirement R6) ensures it cannot be transplanted from D onto different hardware. Similarly, Requirement R7 ensures that each device will require software to be specially loaded onto it, preventing unauthorised parties from installing software onto Devices. Requirement R8 requires the secret key protecting the software on D be agreed upon by D and the software provider.

D will have only one manager at a time, therefore only one entity should be able to provide software to D at any one time (Requirement R10). However, during the lifetime of D, the Operator may change so it must be possible for control of D to change from one O to another (Requirement R11). This creates an atomicity constraint as it is necessary for there to be no crossover period in which a D will accept software from both the outgoing and incoming O.

4.3 Models for Software Provisioning

Two models are proposed for provisioning software bound to Devices: *Device*-centric application provisioning and *Authority*-centric application provisioning. The difference between the two models is which party is responsible for generating the masking-keys and securely transferring the masked software to D.

Section 4.2.1 describes the entities in the scenario this paper considers. However, from this point we merge the Manufacturer and City Hall into one entity: the Authority (A). The Authority is equivalent to M and CH because A will have the same, or equivalent, motivation and responsibilities as the entities it replaces. For example, M is interested in producing Devices that can be securely provided with software; CH also wishes for D to securely receive software from O. To allow D and O to communicate, they may be provided with certificates or pre-installed keys by M or CH. Whether M or CH is responsible for key/certificate provisioning may be determined by practicality/efficiency reasons or by regulations out of the scope of this work. Abstracting these roles into A will allow for any possible division of labour between M and CH to be allowed in the model proposed.

Most requirements listed in Section 4.2.3 are unaffected by the models described in this section. For example, Requirements R1 to R3, R8 and R9 are all concerned with how the protocols will operate and the security guarantees which will require approximately similar solutions in all provisioning models. Requirements R6 and R7 are entirely unaffected by provisioning models as these will be addressed on the device level. However, Requirements R4, R5, R10 and R11 are affected by the provisioning model chosen and the considerations needed will be discussed in the appropriate sections.

4.3.1 Device-centric Software Provisioning

This model divides almost all of the effort of provisioning software to Devices between D and O. In this Device-centric model, A facilitates software provisioning by providing certificates to the parties but does not take an active part in the protocol. It is assumed that both O and D can perform public-key cryptography to verify their identities. This model assumes that O and D trust A; O and D do not have an *a priori* trust relationship they are willing to communicate due to mutual trust in A allowing them to establish a trust relationship for communication and application installation. As a result of attempting to share information between parties using a trusted entity (the Authority) Device-centric provisioning resembles the common "Key Distribution Centre" models found in cryptographic literature on key provisioning [37]. This setting is modelled in Figure 4.1; prior trust is shown with a dashed line and software transfer with a solid arrow.

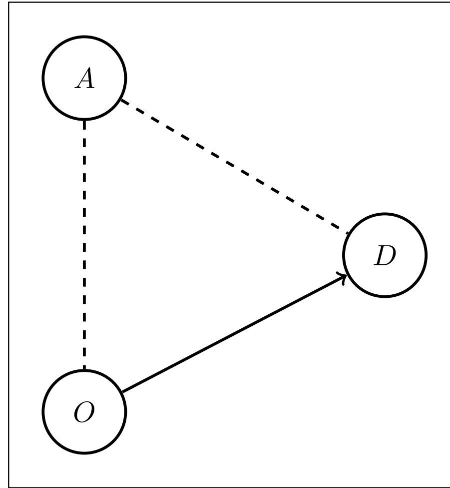


Figure 4.1: Device-centric application provisioning only requires connection between O and D, however this requires prior trust between D and A and between A and O.

In the Device-centric model, O will need a software provisioning certificate from A. This will contain at least a public key, information identifying the Devices O has access to and an expiry time value signed by A. The expiry time value may be a start or finish time for the certificate depending on whether an O is always authorised for fixed period of time or for varying time periods. However, correctly managing the issuing of these provisioning certificates is vital for Requirements R10 and R11. D is only informed of O's validity by the presented provisioning certificate so A correctly issuing these certificates is of vital importance.

For O to load SW onto D it will begin the protocol by contacting D. At the start of the protocol, O and D will perform a fresh mutual authentication. D then checks the certificate provided by O. This check will ensure that the certificate is signed by the Authority responsible for the device as well as checking it has not expired. It is assumed Devices have an accurate measure of date and time for checking certificate expiry. These checks will prove to D that O is permitted to provision D with software. Next, O and D establish a session key for the protocol run to ensure message confidentiality. Once a session key has been established, a masking key is established. O then masks SW and transfers it to D.

After receiving the masked software from O, D is solely responsible for verifying the integrity of the received software and meeting Requirement R4. However, as only O and D participate in the protocol as provisioning time, the proposal will successfully meet Requirement R5 based on the assumption of establishing a secure channel.

4.3.2 Authority-centric Software Provisioning

The alternative to the Device-centric model in Section 4.3.1 is the Authority-centric software provisioning model. In this model, O is prevented from directly communicating with D and instead transfers the application to D via A. However, it allows Requirements R10 and R11 to be addressed more easily as A participates in provisioning directly and can enforce these requirements.

Authority-centric provisioning requires the same number of *a priori* trust relationships as Device-centric provisioning. Both D and O are required to trust A - this trust is used as foundation for Authority-centric provisioning; A is a man-in-the-middle in all communications between O and D. Therefore, this model requires O to have more trust in A than in the Device-centric model as all data between O and D may be observed by A. However, this model addresses the possibility that O is trusted to provide software to D but is not permitted to directly modify its software. This may be necessitated by regulation preventing City Hall from granting complete control to O or simply to allow easier auditing of Devices. The Authority-centric approach comprises two main use-cases; the “Application-Relay” and “Application-Broadcast” models. As with Device-centric provisioning, Authority-centric provisioning parallels parallels a model used in key management literature - the “Key Translation Centre” approach in which a trusted third party acts as an intermediary and root of trust between two communicating entities [37].

The first model is the “Application-Relay” model shown in Figure 4.2. This model is similar to the Device-centric model however it involves O transferring the SW to D via A instead of directly. In this use case, one D is provided with a software update or installation from O. Like in the previous model, O will establish communication with A, the parties will authenticate, establish a session key and the application will be transferred. Next, A and D will communicate, first authenticating themselves then establishing keys for communication and masking. Finally, A will mask the application before transferring it to D.

The second model is the “Application-Broadcast” model depicted in Figure 4.3. This use case is an extension on the “Application-Relay” model and considers a set of identical Devices, $D_{1...n}$, managed by O. When O wishes to install or update software on the Devices then they establish communication with A and transfer the software to them as in the

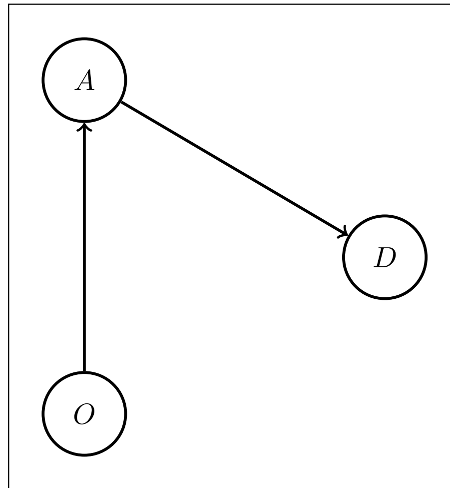


Figure 4.2: The Application-Relay model allows an Authority to monitor application provisioning from O to D.

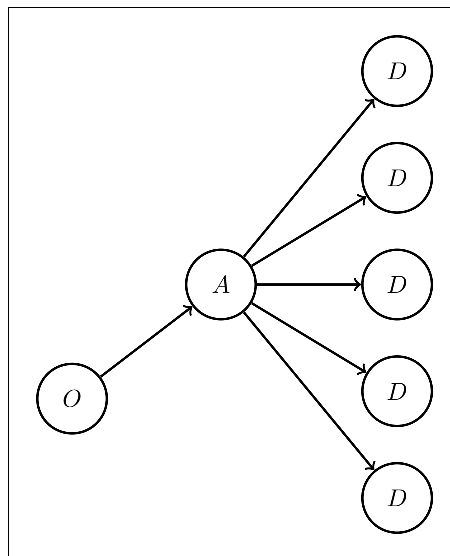


Figure 4.3: Application-Broadcast model considers the situation in which O is loading a large number of Devices with a single application.

previous model. However, once A has received the software, it establishes communication with all of the Devices and transfers it to them. In practice, this setting could apply when a smart city contains a number of the same model of traffic lights or a different smart city device. O may need to update the software on the Devices to update the communications protocol to a different cipher or to expand the data being sent from D to O. In the A-centric model, software transfers must be performed via A, however if many devices require the same software update it would be more efficient to allow A to receive the software once and then update the devices. Once A has received the application they establish sessions and masking keys with each of $D_{1 \ldots n}$, mask the application for each Device and transfer each version to each D in $D_{1 \ldots n}$.

An advantage of the manufacturer-centric model is it is compatible with less powerful devices. Low-end devices have less computational power, sources of randomness and sometimes battery life than a server ran by A that is better equipped to generate strong masking keys.

One argument against the Authority-centric models proposed is they both require O to transfer the unmasked application to A. This requires the transferred software to be carefully protected to ensure Requirement R5, and relies on O trusting A to securely transfer the software to D. Furthermore, it requires A to be trusted to ensure software integrity and Requirement R4 when software is issued to D. The latter can check the software provided, but may have fewer guarantees about the software sent from O to A. Initially, it may seem as if this requires an unreasonable amount of trust in the Authority. However, as A is responsible for producing the keys used by D and owns the devices so also has physical access to them it is always in a position to extract the application. If A would always be able to access the unmasked application via D then it requires no more trust to be placed in A to transfer the application to D via them than to use Authority-centric software provisioning at all.

4.4 Related Work

This section describes existing work related to the problem described, starting with securely binding hardware and software (Section 4.4.1). Later, this section describes two significant industry approaches in software provisioning: MULTOS (Section 4.4.2) and GlobalPlatform (Section 4.4.3).

4.4.1 Binding Hardware and Software

Binding hardware and software applies to various problems including: Digital Rights Management, preventing product counterfeiting and preventing unauthorised firmware modification [67]. Hardware/Software bindings can either be created bi-directionally or uni-directionally. Krasinski and Rosner and Atallah et al. consider binding hardware and software uni-directionally as ensuring a piece of software requires a particular hardware device to execute [67, 10]. Alternatively, is the bi-directional approach taken in Chapter 3

that describes binding hardware and software as creating a bond to ensure that both require the other for execution.

Due to the different bonds formed, the binding schemes described by Krasinski and Rosner, Atallah et al., and this work approach the problem in different ways. All three schemes use a device specific item to bind software and hardware, however the use and source of the items differ. Krasinski and Rosner use a device specific method for generating a key that is checked whenever the copy protection software they are binding executes, however they are not clear how the key is used [67]. Atallah et al. use a PUF (see Section 2.3) to provide response data to bind an instantiation of RSA to a particular hardware device that ensures it is being executed on the intended hardware [10]. Both of the uni-directional schemes are limited to checking the software has not been transplanted to a different device, however the scheme proposed in Chapter 3 also ensures the hardware instance only executes software bound to it. The scheme in Chapter 3 binds the hardware and software at an instruction level ensuring neither the hardware or application can be changed.

Due to Requirement R6, a solution is required to prevent software being transferred from one D to another by Att. Furthermore, Requirement R7 states that D must only execute software that has been provisioned to it by the authorised O. Therefore, the binding scheme that provides the level of security required is that described in Chapter 3. However, that scheme only offers secure binding between hardware and software and has no mechanism for deploying the scheme in the real world. Section 3.6 lists provisioning updates as future work and assume that installation takes place in a secure environment. Therefore, it requires extension before it can be applied to the problem described in Section 4.2.

4.4.2 MULTOS

MULTOS is an operating system for multi application devices developed and standardised by the MULTOS Consortium. Historically, it has been used in smart cards for payment, ID, passports and transport ticketing [83]. MULTOS does not provide protection for installed applications, however the specification requires that development of MULTOS devices be security tested to at least EAL4+ Common Criteria or C.A.S.T. standard [85].

The security of MULTOS is primarily constructed using public key cryptography to pro-

vide secure communication channels between Devices, Issuers and Application Providers. A mechanism for securely provisioning software and firmware updates to devices is included in the MULTOS specification. To install or update an application on a MULTOS device an application provider must work with the device Issuer to produce an Application Load Unit (ALU) and an Application Load Certificate (ALC). The ALU is a container object comprising the application, any application personalisation data needed and a hash of the application signed by the application provider. ALUs can be either public or confidential depending on the security required, confidential ALUs are protected using the public key installed during the enablement of the MULTOS device [84]. The ALC is an authorisation certificate generated by the device issuer containing the public key of the Application Provider and the application header containing the application ID, hash and storage requirements. Before an application can be installed on a device, the installation must be authorised with an ALC.

MULTOS software provisioning considers a similar problem to that described in this chapter, however the solution does not address several points of the problem we are interested in solving. Firstly, MULTOS does not provide any solution to the problem of how to protect applications installed on a MULTOS device. Instead, the MULTOS specification requires that installed software is protected on the device but does not define how it is to be protected. Therefore, it does not meet with Requirements R6 and R7.

Secondly, it does not fit with either of the two models proposed in Section 4.3 as it considers a model that is a combination of the two proposed in this work. MULTOS software provisioning is most similar to the Authority-centric model, however the issuer is not aware of the application being installed on the device. In producing the ALC, the issuer approves installation of an application matching the provided application header, but the actual application is unknown. Mass updating of all managed Devices (Figure 4.3) is not possible in MULTOS; each device must be contacted by the application provider separately as no broadcast model is supported.

4.4.3 Global Platform

GlobalPlatform is a smart card specification concerned with allowing smart cards to run applications from multiple providers in a secure manner. The standard has a very wide

scope and covers system, card and security architectures, life cycle models, secure communication and card and application management. Both the secure provisioning of applications and how to protect them in storage are included in the standard so GlobalPlatform [42] may solve some of the problems considered in this chapter.

In GlobalPlatform, applications are protected using the GlobalPlatform Runtime Environment. The Runtime Environment provides an API for applications, secure storage and secure execution space for applications that separates each applications code and data from other applications. Finally, the Runtime Environment provides communication between the card and off-card entities [86].

GlobalPlatform provides a strong mechanism for secure application installation, management and control via Security Domains. Security Domains are on-card representatives of the Card Issuer and the Application Providers and provide secure services such as key storage, encryption, decryption and signing/verifying digital signatures. Each GlobalPlatform device contains at least one Security Domain: the Issuer Security Domain. The Issuer Security Domain can be responsible for all the security critical tasks listed above or it can allow Application Providers their own on-device space by authorising Application Provider Security Domains to perform the listed tasks [42].

Security Domains allow application installation by verifying the Data Authentication Patterns (DAP) that are provided in the Load File Data Blocks used to transfer applications to GlobalPlatform devices. If a Security Domain has the relevant permissions it is able to accept applications that are accompanied by a DAP using a shared symmetric or asymmetric key from the Application Provider. Security Domains can be locked to prevent their use and their permissions can be limited to ensure the Issuer retains overall control of the device [42].

With the correct configuration a GlobalPlatform based solution could meet with almost all of the requirements described in Section 4.2.3 and fit with the models in Section 4.3. The only requirements that native GlobalPlatform does not meet are Requirements R6 and R7. Although as GlobalPlatform is hardware agnostic a hardware using the system described in Chapter 3 could allow the solution to meet all requirements. The hybrid provisioning required with hardware/software binding could be implemented using a custom protocol to include masking key establishment as GlobalPlatform permits custom protocols.

However, while GlobalPlatform could be combined with a hardware/software binding scheme to address the problem considered in this work, any solution based on GlobalPlatform would almost certainly be not optimal. GlobalPlatform is a large, complicated specification designed to apply to many different scenarios and using it in solving the problem described would introduce unnecessary complication to the system. A bespoke solution would offer more simplicity and efficiency than using a large standard such as GlobalPlatform for a small amount of the functionality offered.

4.5 Proposed Solution

As described in Section 4.4, no solution exists to solving the problem of provisioning software to devices with hardware/software binding. This section describes a new solution based on the scheme proposed in Chapter 3.

4.5.1 Overview of Solution

The scheme proposed in Chapter 3 is the only existing scheme providing hardware/software binding compliant with Requirements R6 and R7. Therefore, it will be used to protect software installed on the Devices.

As stated in Section 4.4.3, a new protocol is required to provision software to the Devices in the network. A solution using a bespoke protocol will be more efficient than one based on a niche configuration of an existing standard and will also be easier to analyse. However, the different models of software provisioning described in Section 4.3 will require separate protocols due to the significantly differing roles of A and O. The new protocols for the different software provisioning models are described in Section 4.5.2 and Section 4.5.3.

The notation used to denote the proposed protocols is listed in Table 4.1.

Table 4.1: Protocol Notation

n_x	A nonce with ID x.
m_x	The payload component of a message with ID x.
g^x	A Diffie-Hellman key share.
$\mathsf{Enc}_k(m)$	The encryption of the message m using the key k.
O-cert	The O-Cert of the Device Operator.
sk_X	The signing key of X.
$\mathsf{Sig}_k(m_x)$	The hash-based signature of message with ID x produced using the key k.
$\mathsf{H}(m)$	The hash digest of a message m.
$M_k(App)$	The software App masked using a key k.
$\mathsf{MAC}_k(m_x)$	A Message Authentication Code calculated for a message with ID x using a key k.

4.5.2 Device-centric Software Provisioning

This section describes the protocol for provisioning software in the Device-centric model presented in Section 4.3.1. The developed protocol is presented in Table 4.2.

The D-centric protocol is a three message protocol that establishes freshness and authentication by O and D signing freshly generated nonces with the other message data. These signatures are written as $\mathsf{Sig}_{\mathsf{sk}_O}(m_x)$ with x equal to 1, 2 or 3 where the signature is calculated over the first, second or third message. This allows the protocol to meet Requirements R1 and R2.

Keys for the session and masking are established using Diffie-Hellman key agreement. These are carried out by the exchanging of g^a, g^b, g^c and g^d; allowing the mutual agreement upon the session and masking keys: g^{ab} and g^{cd}. This generation and use of session and masking keys for each protocol run ensures the scheme satisfies Requirements R3, R5, R8 and R9.

Requirements R10 and R11 dictate that only one, authorised O may install software on D at a time and that it must be possible for one O to cease provisioning SW and another to start. In the Device-centric protocol this is achieved using the O-cert. The O-cert is a certificate that is provided to O by A and includes pk_O signed by A's public key as well as a certificate expiry time and information describing the devices that O is permitted to manage. Certificate expiry times and only granting one valid O-cert at a time will ensure the protocol meets Requirements R10 and R11. Alternatively, the O-cert could contain a start time and always have a fixed lifetime, however specific expiry times allow certificate

Table 4.2: Device-centric Software Provisioning

Device-centric Protocol		
1.	$O \rightarrow D$	$O \| D \| n_1 \| g^a \| O\text{-cert} \| \mathsf{Sig}_{\mathsf{sk}_O}(m_1)$
2.	$D \rightarrow O$	$D \| O \| n_1 \| n_2 \| g^b \| \mathsf{Enc}_{g^{ab}}(g^c) \| \text{cert}_D \| \mathsf{Sig}_{\mathsf{sk}_D}(m_2)$
3.	$O \rightarrow D$	$O \| D \| n_2 \| \mathsf{Enc}_{g^{ab}}(M_{g^{cd}}(SW) \| \mathsf{H}(SW) \| g^d) \| \mathsf{Sig}_{\mathsf{sk}_O}(m_3)$

expiry to more easily align with trial periods/Operator contracts. The certificate of D in the Device-centric protocol is a normal certificate containing the public key of D signed by A.

Finally, the inclusion of the application signature ensures that Requirement R4 is met by the protocol.

4.5.3 Authority-centric Software Provisioning

This section will describe the protocols for provisioning software according to the Authority-centric software provisioning model in Section 4.3.2. The main difference between the Authority-centric and Device-centric protocols is that the former requires a two stage process as the application must be transferred to D via A instead of directly from O to D. Therefore, the Authority-centric approach is longer and less efficient than the Device-centric protocol proposed in Section 4.5.2. The Authority-centric protocols proposed (Table 4.3) are a two stage protocol consisting of transferring the application from O to A (messages 1-3) and then from A to D (messages 4-6). This allows for Application Relay as depicted in Figure 4.2 and Application-Broadcast as depicted in Figure 4.3. The Application-Broadcast would require running the latter half of the protocol (messages 4-6) multiple times, once for each device.

The second half of the protocol in the A-centric model can be approached in two ways depending on the relationship between A and D: with public key cryptography to build trust or using pre-existing trust. In D-centric software provisioning O makes contact with D to install software. It is assumed that both parties are communicating based on their mutual trust in the Authority (A) rather than an existing trust relationship. By contrast in the A-centric model it is plausible that the existing trust relationship between D and A may include having a pre-shared symmetric key. In such a case D may not be required to do as many public key cryptographic operations which is advantageous in resource-

Table 4.3: Authority-centric Software Provisioning (with Public-Key Cryptography)

Authority-centric Provisioning		
1.	$O \to A$	$O \| A \| n_1 \| g^a \| O\text{-cert} \| \mathsf{Sig}_{\mathsf{sk}O}(m_1)$
2.	$A \to O$	$A \| O \| n_1 \| n_2 \| g^b \| \text{cert}_A \| \mathsf{Sig}_{\mathsf{sk}_A}(m_2)$
3.	$O \to A$	$O \| A \| n_2 \| \mathsf{Enc}_{g^{ab}}(SW \| \mathsf{H}(SW)) \| \mathsf{Sig}_{\mathsf{sk}_O}(m_3)$
4.	$A \to D$	$A \| D \| n_3 \| g^c \| \mathsf{Sig}_{\mathsf{sk}_A}(m_4)$
5.	$D \to A$	$D \| A \| n_3 \| n_4 \| g^d \| \mathsf{Enc}_{g^{cd}}(g^e) \| \text{cert}_D \| \mathsf{Sig}_{\mathsf{sk}_D}(m_5)$
6.	$A \to D$	$A \| D \| n_4 \| \mathsf{Enc}_{g^{cd}}(M_{g^{ef}}(SW) \| \mathsf{H}(SW) \| g^f) \| \mathsf{Sig}_{\mathsf{sk}_A}(m_6)$

Table 4.4: Authority-centric Software Provisioning (with Pre-Shared Keys)

Authority-centric Provisioning with Pre-Shared Keys		
1.	$O \to A$	$O \| A \| n_1 \| g^a \| O\text{-cert} \| \mathsf{Sig}_{\mathsf{sk}O}(m_1)$
2.	$A \to O$	$A \| O \| n_1 \| n_2 \| g^b \| \text{cert}_A \| \mathsf{Sig}_{\mathsf{sk}_A}(m_2)$
3.	$O \to A$	$O \| A \| n_2 \| \mathsf{Enc}_{g^{ab}}(SW \| \mathsf{H}(SW)) \| \mathsf{Sig}_{\mathsf{sk}_O}(m_3)$
4.	$A \to D$	$A \| D \| n_3 \| g^c \| \mathsf{MAC}_{PSK}(m_4)$
5.	$D \to A$	$D \| A \| n_3 \| n_4 \| g^d \| \mathsf{Enc}_{g^{cd}}(g^e) \| \mathsf{MAC}_{PSK}(m_5)$
6.	$A \to D$	$A \| D \| n_4 \| \mathsf{Enc}_{g^{cd}}(M_{g^{ef}}(SW) \| \mathsf{H}(SW) \| g^f) \| \mathsf{MAC}_{PSK}(m_6)$

constrained settings. Therefore a second A-centric protocol is proposed here that uses a pre-shared key to replace some signatures with MACs (Table 4.4).

Requirements R1 to R3, R5, R8 and R9 are all achieved in the Authority-centric protocols in the same manner as described in Section 4.5.2 for the Device-centric protocol. However, Requirements R10 and R11 do not require a solution in the Authority-centric model as A is able to trivially ensure them by only accepting software from the currently authorised O and no others. Finally, Requirement R4 is met using digital signatures ($\mathsf{Sig}_{\mathsf{sk}_O}(m_x)$) in the Public-Key scheme and by using MACs ($\mathsf{MAC}_{PSK}(m_5)$) in the Pre-Shared Key protocol.

4.6 Analysis

Further to the analyses of the proposed protocols in Sections 4.5.2 and 4.5.3 a formal, mechanical analysis of the protocols has been carried out using Tamarin Prover [91]. This section describes the protocol models, the analysis carried out and the security results proven.

4.6.1 Description of Protocol Models

The protocol analysis was carried out using three models: one for each protocol proposed. The models represent the protocols as state machines that are progressed through as messages are sent and received by the Device, Authority and Operator.

A digram of the state machine for analysing the D-centric protocol is included in Figure 4.4. This model contains two types of transitions between the rules representing the states of the protocol: messages and entity-states. Both the correct message and entity-state are required before a rule can be executed. The messages are the outputs of the $D1$, $O1$ and $D2$ rules that are sent between the Operator and Device and that are also sent to the attacker.

The entity-states are facts that store the information that a particular entity in the protocol run is aware of and needs to retain. These states are created as private facts the attacker is not shown and are used to retain information such as Diffie-Hellman exponents. This state machine representation of the protocol allows a legitimate execution path to be represented with the rules evaluated in the order: $O1$, $D1$, $O2$, $D2$. It also limits attacker behaviour to only realistic attacks such as trying to forge a session with a legitimate A for part or all of a protocol run. The attacker can also try to break the scheme security by eavesdropping a legitimate protocol run. This is reasonable as it ensures the attacker cannot start the protocol by forging $m2$; a legitimate Operator would notice that $m1$ had not been sent.

Similar state models for the Authority-centric protocols were developed to produce Tamarin models for the proposed protocols, but are omitted for brevity.

For the requirements listed in Section 4.2.3, those verified by mechanical analysis are tested by validating lemmas against the protocols represented by rules. The full sets of rules and lemmas are included in Appendices B to D. One omission from included Tamarin code is the lemmas used to prove reachability of all protocol states. This choice was made for reasons of brevity and clarity.

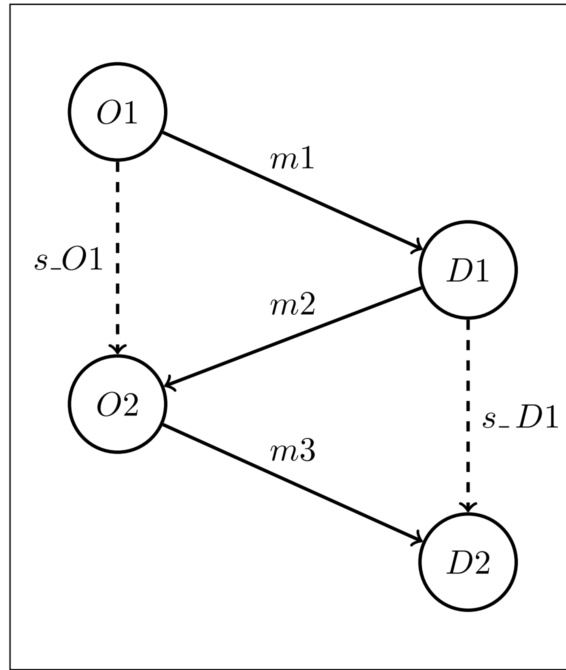


Figure 4.4: Modelling Device-centric application provisioning requires a state machine of four states with five connections.

4.6.2 Scope of Mechanical Analysis

The Tamarin models prove the communication requirements of the protocol. Rules and lemmas were only included to test Requirements R1 to R5 and R8 to R11. Requirements R6 and R7 were not included in the model and it was assumed that installed software is securely stored.

The method for the analysis was to first prove the protocols satisfy Requirements R1 and R2. Intuitively, some requirements (such as Requirements R8 and R9) rely on fresh authentication. Therefore, the Tamarin model first proves Requirements R1 and R2 and then uses those to test if other requirements have been met.

To limit the complexity of the model, it is assumed an entity in the model is either a Authority, Device or an Operator and they do not switch roles. However, a result of this is different messages in the protocol will only be created by certain entities, for example, message 5 is only ever output by a Device.

4.6.3 Modelling Assumptions Required

To model the proposed protocols several assumptions were made. Some were required to allow Tamarin Prover to reason about the protocols. However, most assumptions were chosen to limit model complexity without compromising its value. An example is preventing entities from switching roles. This is not unreasonable because a traffic light in a city is not going to change into a different device or become an Authority or Operator during its lifetime. This assumption did not weaken the security proven by the model but did simplify the protocol verification.

Other assumptions were made to prevent Diffie-Hellman key-exchange exponent edge cases that could theoretically lead to attacks. An example of this is assuming all parties will only generate safe Diffie-Hellman shares. This excludes all shares or share combinations equal to the generator as these would allow for a trivial break by the attacker. The chance of these scenarios is negligible so preventing them by assumption is reasonable and allows the model to prove the security properties without weakening the result.

Another example of a simplifying assumption is to assume all generated nonces are unique. In practice, an implementation would check that received nonces are fresh so this assumption is reasonable as it removes the need to check nonce freshness in the model.

4.6.4 Results of Analysis

When verifying the protocols met Requirement R10, the model showed that implementations of this protocol will need to avoid problems with O-Certs expiring between messages. Sending the O-Cert in the initial message makes sense in terms of practicality: D or A need not continue to communicate with O if they do not have a valid O-Cert. However, the Tamarin model highlighted the possibility of a time-of-check, time-of-attack vulnerability. Preventing this vulnerability requires implementations to check the O-Cert is still valid at install time and has not expired since receipt.

After including an extra O-Cert validity check, the protocol models met all security requirements. Therefore, when implemented correctly, the three protocols described in Section 4.5 will meet the requirements described in Section 4.2.3.

One limitation of the analysis carried out is it does not consider the consequences of compromised devices. The current model assumes the parties are honest but communicating over a channel controlled by the attacker. This allows the model to cover any combination of honest cases but it does not consider if individual devices are compromised. The security of the key establishment has been proven in the context of the attacker being able to break some session keys. However, a malicious device can always leak the current key or even the application being transferred so some requirements will not be able to be met by the current scheme.

A further limitation to the analysis is that it is only interested in full breaks or application leaks. If one version of a piece of software is leaked to the attacker, this might be of use to an attacker in guessing parts of the next version of the code. However, the existing model has only considered proving the security of the protocol against total software leakage and does not model partial information leakage.

4.6.5 Further Observations

After modelling and verifying the proposed protocols, one emergent property is a very strong resemblance to protocols proposed in ISO 9798 [55]. This standard is concerned with entity authentication; a property required of our proposals and listed in Section 4.2.3. Requirements R1 to R3 and R9 were highly significant in designing the proposals of this chapter and this combination of authentication, replay resistance, forward secrecy and secure key establishment are a feature of many standardised protocols such as those in published by ISO. These common features are the most significant cause of the observed resemblance.

However, a further factor which is likely to have contributed to the similarity between this chapter and ISO 9798 is the 2002 work of Basin et al. [13]. In their paper, the authors used Tamarin Prover to assess the security of the protocols standardised in ISO 9798. The results of their study were that they found several issues. The authors proposed fixes for the protocol which were accepted by ISO and included into the standard [13]. However, it also means that not only have both our protocols and those of ISO 9798 been designed with similar requirements but they have also been verified by the same tool which has likely also contributed to the similarity in the results.

4.7 Conclusion

This chapter has presented a new problem setting for provisioning software to devices in smart cities. The new scenario applies to the real world and includes the important issue of protecting the software installed on the devices.

Two models for distributing software to devices in a smart city are described and protocols solving the problems have been suggested. The protocols have been proven to be secure using a formal analysis tool that considers large numbers of possible combinations of messages when looking for flaws.

Future work in this area would be to expand the analysis to check the security of the scheme when one or more devices are compromised. The attacker described in Section 4.2.2 has physical access to the Devices so could potentially mount an attack to gain control over some Devices. Another area for future work would be examining the performance cost of the proposed protocols by implementing them using various microcontrollers. This would provide cost measurements in terms of time and energy required to use the proposed scheme that would allow for easier comparison with other protocols.

Chapter 5

Securing Application Execution and Binding Hardware and Software

Contents

5.1 Introduction 94

5.2 Problem Description 95

5.3 Related Work 100

5.4 Proposed Solution 105

5.5 Implementation 110

5.6 Analysis 119

5.7 Conclusion 121

Chapter 3 proposed a scheme by which hardware and software can be bound together to create a secure dependency between them. This bond prevents software from being moved between devices. However, platform piracy and software modification are just some attacks against IoT devices. Exploiting existing software to force an unauthorised execution path is another type of attack. This chapter considers the problem of ensuring Platform Specific Execution while also Securing Application Execution to guarantee correct, unmodified execution of legal, application execution paths and no others. To solve this problem, an expanded hardware-software binding scheme is proposed. A prototype of the proposed scheme, implemented as a soft-core processor, is presented to demonstrate the new scheme.

5.1 Introduction

Chapters 3 and 4 considered how to prevent counterfeiting and device tampering attacks against embedded systems. A binding scheme was proposed in Chapter 3 and Chapter 4 extended that work by considering use and deployment. This chapter follows Chapter 3 differently by expanding the problem considered and proposing an extended scheme. As stated in Chapters 3 and 4, device counterfeiting and firmware modification are serious threats to IoT devices. However, they are a subset of attacks on IoT devices with others attempting to gain control of a device by exploiting legitimate applications to run arbitrary program code or leak sensitive information. One protection against these types of attacks is including measures to ensure Secure Application Execution (SAE), this prevents arbitrary code execution and program exploitation by securing application control flow [62].

Related to SAE is the problem of Platform Specific Execution (PSE) or securely binding hardware and software together. This problem seeks to ensure only applications from authorised parties can execute. However, the two problems differ in two main ways. Firstly, SAE ensures the current application is executing correctly, but not the legitimacy of the author of the code [33]. Conversely, PSE, as defined in Chapter 3, focusses on ensuring only applications personalised to a device by an authorised party can execute but does not consider legitimate applications failing to operate correctly. Therefore, securing IoT devices requires a combination of both problems: ensuring devices correctly execute legitimate applications and nothing else.

5.1.1 Contributions

The main contributions of this chapter are:

1. Modelling a new problem combining hardware/software binding with Secure Application Execution (Section 5.2).
2. Providing a transition based perspective of Secure Application Execution (Section 5.2.2).
3. Proposing a set of requirements for securing application control flow and execution location (Section 5.2.4).

4. Proposing a scheme to solve the problem considered which combines a prior scheme, SOFIA [32], to our previous work, Chapter 3, to ensure SAE and PSE (Section 5.4).

5. Implementing the proposed scheme and describing the components of the developed system (Section 5.5).

6. Analysing the scheme and implementation (Section 5.6).

5.1.2 Chapter Structure

This chapter is structured as follows: Section 5.2 contains an overview of the problem considered in this chapter. Section 5.3 describes the related work in hardware/software binding and Secure Application Execution. Section 5.4 contains the solution proposed for securing application confidentiality and execution. Section 5.5 describes the implementation of the proposed scheme and includes a breakdown of the main engineering decisions and constraints. Section 5.6 analyses the proposed scheme and implementation. Section 5.7 concludes the chapter and describes potential future work in this area.

5.2 Problem Description

This paper considers ensuring secure, unmodified and platform specific execution of applications. Informally, if an application, *App*, has finished executing, we guarantee it has executed correctly and on the specific hardware device it is intended for. In this work, correct execution is defined as following a legitimate control flow with no operations modified or omitted from the program as it was installed. The following subsections define Platform Specific Execution (Section 5.2.1), Secure Application Execution (Section 5.2.2), the attacker model considered (Section 5.2.3) and the requirements for a proposed solution (Section 5.2.4).

5.2.1 Platform Specific Execution

Platform Specific Execution (PSE) is the problem of ensuring a Software instance SW installed on a Device (D) is able to execute on D and on no other device, D'. Usually, an

SW is deployed to multiple devices or systems. Therefore to achieve PSE, each SW must be made unique for the D it is executed by.

PSE can be guaranteed by using hardware-software binding to securely personalise a software instance, SW, to many devices. With this technique an installed software, SW_D is made reliant on some element or information held or known only by D, the result of this being that SW loaded onto a different device, D', will not execute correctly. Therefore a hardware device is restricted to executing only the SW bound to it and no different SW'. This can prevent code injection attacks. Chapters 3 and 4 used Hardware-Software two-way bonds to prevent counterfeiting, firmware modification and for securing software in smart cities.

5.2.2 Secure Application Execution

As well as PSE, we are concerned with Secure Application Execution (SAE), which is defined as ensuring applications only follow correct control flows as defined by the developer [1]. A correct control flow has been followed if every instruction has been executed without any being skipped or modified. While instructions not executed due to correctly executed conditional `JUMP` instructions have been legitimately skipped, instructions not executed due to modification or deletion are illegitimately skipped and compromise SAE. Therefore, achieving SAE comprises two main requirements: ensuring instruction integrity and ensuring CFI. PSE considers instruction integrity, therefore this subsection will focus on CFI.

Examples of legitimate and illegitimate execution of an application are found in Figure 5.1. The figure shows a code snippet with instructions marked with ticks (✓) and crosses ($\times$). A legitimate execution through the snippet would be to execute every instruction marked with a tick, when instruction 3 was executed, the zero flag was not set and control flow jumps to instruction 6. If only the instructions marked with crosses were executed, this would be an illegitimate execution because the conditional `JUMP` of instruction 3 is entirely avoided.

In this work, the legitimacy of logical tests based on external data is not included in the scope of SAE. For example, attacks tampering with sensors/sensor data to ensure one

```
1  Op ✓ ×
2  Op ✓ ×
3  Jmp NZ, 6 ✓
4  Op  ×
5  Jmp 8  ×
6  Op ✓
7  Op ✓
8  Op ✓ ×
9  Op ✓ ×
```

Figure 5.1: Legitimate executions follow each instruction in a control path such as the ticked instructions, illegitimate executions skip instructions such as the path avoiding the `JUMP NZ, 6` that is marked by crosses.

side of an `if` instruction is always chosen is out of scope. However, skipping a conditional `JUMP` operation to force execution of one path of an `if` statement, as shown in Figure 5.1, is within the scope of this work. We also include attacks attempting to force control flow to alternative portions of the application (perhaps ahead of schedule or to code not in the current flow).

The example of illegitimate execution shown in Figure 5.1 shows one type of instruction transition being attacked. To ensure SAE, we argue there are three types of instruction transitions to protect. These are: sequential transitions, instruction-dependent transitions and instruction-independent transitions. Each of these are described below with examples of attacks.

Sequential transitions are the simplest and most common instruction transition. They occur when an instruction at location x is executed and the following instruction to be executed is at location $x + 1$. The program counter increases by 1 and no jump occurs. These are found after arithmetic, logical, I/O or load/store instructions. However, they also occur after conditional `JUMP` instructions whenever the logic test fails and the jump is not executed. The example of illegitimate execution in Figure 5.1 shows the sequential transition between instructions 2 and 3 changed to prevent instruction 3 executing.

Instruction-dependent transitions happen after unconditional `JUMP` or `CALL` instructions, if the destination of the `JUMP` is determined by the operand of the instruction. For example, `JUMP x` or `CALL x` causes an instruction-dependent transition by changing the value of the program counter (the location of the next instruction) to `x`. The transition from instruction 5 to instruction 8 in Figure 5.1 is an example of an instruction-dependent transition. Relative `JUMP` instructions are also considered as causing instruction-dependent transitions, as the distance jumped and instruction location can be used to replace the

instruction with an equivalent, absolute `JUMP` instruction.

Finally, instruction-independent transitions occur when the next instruction is not identifiable from the instruction only. The most common causes of instruction-independent transitions are `RETURN` and conditional `JUMP` instructions. A `RETURN` can transition to different locations based on where the function was called from, the location of the next instruction is loaded from the stack and not from the current instruction. This allows functions to be called from different locations and execution flow to return to where the function was called from after execution. However, it also introduces a weakness exploited by attacks such as Return Oriented Programming (ROP) [19]. Conditional `JUMP` instructions lead to the immediately proceeding instruction or a remote instruction depending on the processor state at execution time. Therefore, the instruction alone is insufficient to identify the location of the next instruction to execute as processor flags, such as the Zero or Carry flags, are needed. In the legitimate execution in Figure 5.1, the transition from instruction 3 to instruction 6 is an instruction-independent transition. The `JUMP` was chosen because the Zero flag was not set.

If all three types of instruction transitions are secured, it follows that SAE has been achieved as it is not possible for program flow to deviate from the designed path.

5.2.3 Attacker Model

The attacker considered in this work is a Dolev-Yao attacker [35] able to read and manipulate data but unable to break cryptographic primitives. They can read and write to any memory location exterior to the CPU, such as RAM or long-term storage, but they are unable to modify CPU cache memory or registers directly. The attacker has access to personalised applications extracted from legitimate devices but does not have access to the original application binary. This simulates the attacker buying the device protected with the SAE-PSE scheme and extracting the memory contents.

We assume the device is assumed to have protection mechanisms preventing side-channel leakage and so the attacker is not able to use side-channel vulnerabilities such as power analysis [65], acoustic side-channels [41] or cache timing side-channels [64, 74] to attack the proposed scheme.

The attackers goal is to achieve any of the following:

1. Skip the execution of an instruction without detection (execution must continue as if no instruction has been skipped).
2. Modify an instruction on the device that successfully executes.
3. Redirect software control flow to an alternative portion of code from memory. The code must execute earlier than designed or else appear in the control flow when it would have not appeared at all.
4. Execute software taken from one device, D, on a device, D' where $D \neq D'$.
5. Execute unauthorised software written by the attacker on a device D.
6. Unmask software taken from a device D, in order to discover the original, unmasked, source binary.

5.2.4 Requirements

The requirements for a solution to the problem considered are:

R1) **Software Confidentiality:** the software must not be revealed to attackers. When stored in memory, the software must be encrypted.

R2) **Authorised Execution:** only applications bound to a device can be executed by the device.

R3) **Immutable Applications:** securely personalised software must not be meaningfully modifiable by unauthorised parties. Attackers are unable to make targeted changes to the software.

R4) **SAE - Secure Sequential Transitions:** sequential transitions where no `JUMP`, `CALL` or `RETURN` occurs must be secured in the proposed scheme.

R5) **SAE - Secure Instruction-dependent Transitions:** instruction-dependent transitions must be protected to ensure that after the transition, the program counter value can only be the value determined by the instruction.

R6) SAE - Secure Instruction-independent Transitions: instruction-independent transitions must result in the program counter being changed to the appropriate value after a conditional `JUMP` or the address following the last executed `CALL` if the transition is caused by a `RETURN`.

5.3 Related Work

This section considers previous work in Platform Specific Execution and Secure Application Execution and analyses it against the requirements listed in Section 5.2.4.

5.3.1 Platform Specific Execution

Platform Specific Execution is also known as binding hardware and software together and has been applied to several problems including: counterfeiting prevention, securing device firmware, protection of smart cities and in Digital Rights Management (DRM) - see [67] and Chapters 3 and 4. In 2003, Krasinski and Rosner patented a method for binding a piece of software to a specific hardware instance. Their method created a unidirectional bond to attach a piece of DRM software to a device to protect digital audio or visual content. A key is derived from unique or distinctive hardware, software or firmware identifiers to ensure only the specific hardware instance may access the protected media [67].

Later, in 2008, Atallah et al. published a mechanism for binding software to a specific hardware instance in a Virtual Machine environment. Like Krasinski and Rosner, Atallah et al. created a unidirectional bond to attach software to hardware. To prevent distinctive hardware, software or firmware features being virtualised, Atallah et al. used a Physically Uncloneable Function (PUF) to confirm the presence of legitimate hardware to the executing software [10].

Chapter 3 is the first work to propose the need for a bi-directional bond between hardware and software. It is important to connect both to ensure software executes only on legitimate hardware and also that hardware only executes legitimate software, which would ensure the integrity of deployed products as neither the software nor the hardware could be inauthentic. A PUF may be used to provide a device-specific secret for use in creating

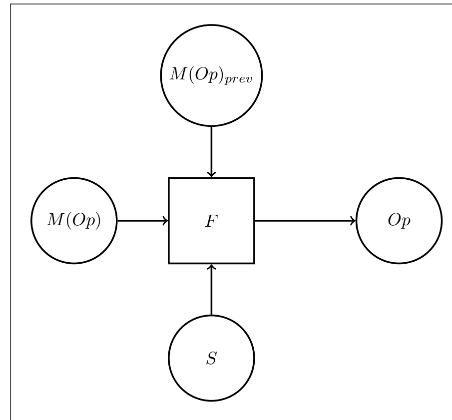


Figure 5.2: The binding scheme from Chapter 3 uses a secret, S, and the previous, masked instruction, $M(Op)_{prev}$, to unmask $M(Op)$ and reveal Op for execution.

the bond (Section 3.4.3). The proposed scheme produces the bond between hardware and software by masking each instruction using the previous stored instruction as well as a device-specific secret which ensures the application will execute correctly on no other device and that applications bound to the device are able to execute correctly. Using the previous instruction in memory prevents attackers creating alternative, executable applications by moving instructions. It also prevents the masking from behaving as an electronic code book, a block cipher mode with known flaws. A diagram of the scheme from Chapter 3 is included in Figure 5.2.

Requirements R2 and R3 make it necessary for the PSE scheme used in the proposed solution to be bidirectional. The Krasinski-Rosner and Atallah et al. schemes are therefore not suitable for inclusion. Our previous scheme provides a two way bond sufficient for the two requirements, however like the Krasinski-Rosner and Atallah et al. schemes, it offers no guarantees about application execution. These would need to be included by extension or modification of the original proposal.

5.3.2 Secure Application Execution

A significant work in SAE is the 2005 paper by Abadi et al. introducing CFI enforcement as a means of preventing various attacks. Prior to their work, most schemes were developed to protect against individual attacks, however Abadi et al. proposed a method for protecting existing code without hardware changes. Their technique uses a Control-Flow Graph (CFG) drawn from an application binary using static analysis. Security of execution is

ensured by only allowing control flow transfers included in the CFG [1].

In 2017, Clercq and Verbauwhede surveyed CFI literature and considered the problem using CFGs and what they labelled as backward and forward edges [33]. This is similar to our identification of instruction transitions, however Clercq and Verbauwhede focus on instructions causing control flow transfers and not on sequential transitions. Repeated sequential transitions are merged into CFG nodes and it is assumed the single entry/exit nature of basic blocks holds. The authors describe *forward edges* as control flow transfers caused by jumps and calls and *backward edges* as those caused by returns [33]. One disadvantage of considering all jumps and calls equally is it loses the distinction between jumps to register-determined and instruction-determined locations. This is significant as, when implementing a scheme, instruction-dependent transitions are simpler to process than instruction-independent transitions, as described in Chapter 3. Therefore, considering transitions is important when also considering hardware-software binding and creating schemes for implementation.

The rest of this section describes some proposed methods relevant to this work for ensuring CFI.

5.3.2.1 Jump Labelling

Jump labelling is a CFI protection technique proposed by Abadi et al. in 2005 [1]. They suggested it as a defence, implementable in software, for protecting against various attacks. The mechanism works by labelling the destinations of control flow instructions with IDs checked as part of the jump so that if the label at the destination fails to match that at the jump then program execution is aborted [1].

It was originally proposed for execution in software only [1], however it can be implemented in hardware and was by Christoulakis et al. and Sullivan et al. in later works [22, 113]. However, the labelling approach has several issues preventing it from meeting Requirements R4 to R6. Firstly, it does not protect instructions executed after the jump instructions, it only secures the jump instructions. Secondly, jump labelling relies on the attacker being unable to interfere with the program in memory, while our attacker can write to memory (Section 5.2.3). Finally, it is vulnerable to attacks changing a return

address from the original address to an alternative address. These attacks are described in more detail in Section 5.4.2.

HAFIX extends on the labelling approach by maintaining a table of labels and marking them active when a call is made from a function. On the return instruction, the table is consulted to ensure the label is active. An error occurs whenever an execution attempts to return to an inactive function. This approach lends itself to hardware implementation and could be included in the processor pipeline. Overall, the cost of HAFIX is small as it merely requires a one bit flag for every call instruction in the program [30]. However, HAFIX does not protect all transitions and does not protect the instruction jumped to, so it does not meet Requirements R5 and R6.

5.3.2.2 Shadow Call Stacks

A Shadow Control Stack (SCS) is a mechanism used to protect return operations from being misused by attackers. It is effective against ROP or return-into-libc attacks and is regularly included with jump/call protections in CFI schemes. Essentially, an SCS is a second stack storing the return addresses added to the "normal" stack. The SCS is checked when a return instruction executes to ensure the top of both the SCS and the "normal" stack are equal. SCS is simple to implement as it only requires extending the logic of `RETURN` operations and does not require performing program analysis [33].

As identified by Clercq et al. in 2017, the location of the SCS is critical to the security of the scheme. A stack in main memory would allow a large call depth but would be vulnerable to attackers controlling memory. Different approaches suggest using dedicated hardware buffers or memory only accessible by `CALL` and `RETURN` instructions or using memory divided on a kernel level to store the SCS [33]. Therefore, SCS meets Requirement R6 but not Requirements R4 and R5.

5.3.2.3 SOFIA

A significant, recent work in the area of SAE is SOFIA, a software and control flow integrity architecture proposed by Clercq et al. in 2017 [32]. They propose a method for

securing CFI and describe the performance of an implementation developed. This work sets five goals for protecting application execution: software integrity, control flow integrity, tampered code protection, code confidentiality and reverse engineering protection. These are chosen to ensure applications correctly execute and to minimise attackers ability to examine the code to look for flaws/vulnerabilities [32].

In essence, SOFIA protects application execution by ensuring attackers are unable to change any instructions. Furthermore, the architecture creates location-specific protections for each of the instructions to ensure they cannot be moved to different locations in memory and be executed in a different order. Integrity checks are included to detect modified instructions before execution [32].

The instruction protection mechanism in SOFIA is partly in the form of a masking scheme such as that proposed in Chapter 3. However, instead of chaining instructions, Clercq et al. have used a nonce and the preceding and current PC values as the information used by the function to protect instructions [32]. As a result, the instruction masking calculation is as follows (adapted from [32]):

$$Op_x = Op'_x \oplus E_{k_1}(\omega || x_{\text{prev}} || x)$$

Where x is the PC value of the current instruction, x_{prev} is the PC value of the previously executed instruction, ω is a nonce and Op_x and Op'_x are the instruction and masked instruction in location x respectively. Unlike in the scheme proposed in Chapter 3, in the SOFIA masking function PC values are used based on the order of instruction execution rather than the order of instruction storage. Note, that ω must be unique for each version of the each secured application; Clercq et al. do not require it to be stored in a secure location. Also calculated with the decrypted instructions are blockwise CBC-MAC values used as an integrity check using a second key k_2. The MAC of each block is encrypted and then stored in the first location of each block and is loaded and compared with the calculated block MAC as the instructions are decrypted. The combination of these checks provides an integrity check ensuring attacks on the stored instructions are detected and control flow specific instruction masking prevents instructions being moved in the application [32].

One strength of SOFIA is it considers a powerful attacker with full control of external storage, I/O and buses [32]. However, one disadvantage of SOFIA is how it handles reused code accessed from multiple locations. Due to the instruction encryption, each instruction can only be accessed from one location. To allow code to be accessed from multiple locations, Clercq et al. mark some blocks as accessible from multiple locations. These special blocks start with a series of entry points, one for each `JUMP` or `CALL` leading to the block. Each of these entry points are modified instructions allowing the calculated MAC and instruction decryption values to each be computed from a designated "first" instruction in the block instead of the actual first instruction in the block [32]. This is rather inflexible and requires code analysis to be performed to identify all the locations each function is called from before appropriate entry points can be appended. Therefore, while SOFIA meets Requirements R1 to R6 it is not a perfect solution as handling reused code requires extra program analysis and blocks of code adding to each function based on the number of locations calling the function.

5.4 Proposed Solution

This section describes the proposed solution to the problem posed in Section 5.2. The solution proposed is based on the hardware-software binding scheme proposed in Chapter 3 and summarised in Section 5.3.1. However, that scheme, hereafter referred to as simple hardware-software binding or the simple scheme, meets only some of the requirements from Section 5.2.4. Chapter 3 listed similar requirements to Requirements R1 to R3 when the scheme was proposed. However, extensions to provide SAE and meet Requirements R4 to R6 are needed. The scheme proposed in this section extends the simple scheme using ideas from the proposals listed in Section 5.3.2.

5.4.1 Protecting Sequential Transitions

The first type of instruction transition defined in Section 5.2.2 is sequential transitions. Transitions of this type are incidentally and implicitly protected by the simple scheme due to the cascading nature of the hardware-software bond. An instruction Op stored in location x is masked using the data stored in location $x-1$ (the previous, masked instruc-

tion) to produce Op'. For efficiency, the authors reduce the memory operations needed for unmasking by latching data read from memory after the instruction is unmasked. This allows, in the case of sequential transitions, masked data to be used in unmasking the next instruction without being loaded again. This latching allows simple hardware-software binding to provide implicit sequential transition security.

Extra memory accesses are only required in the simple binding scheme in the case of control flow instructions. Therefore, the only helper data available in unmasking an instruction after a sequential transition is the previously executed instruction. This provides implicit sequential transition security and consequently, instructions in series execute correctly and can only be modified by someone who knows the masking key. Therefore, the simple scheme protects instructions executed in series by guaranteeing sequential transitions.

5.4.2 Protecting Instruction-Dependent Transitions

While the simple hardware-software binding scheme protects sequential transitions between instructions, it does not secure instruction-dependent transitions. These can be manipulated by an attacker to execute arbitrary program code and violate Requirement R3, an example is included in Figure 5.3. Consider the application on the left side of Figure 5.3, execution starts from the top and executes two arbitrary instructions followed by a `JUMP`. The `JUMP` leads to a short block of code, x, that performs a security check before outputting sensitive data. The simple scheme only secures the `JUMP` instruction that sets the PC value to x, but not the *contents* of x. Therefore, it is possible for the code to be shifted in memory to skip the security check before outputting the sensitive data. The attacker could change the code into that on the right of Figure 5.3 to output the sensitive data without executing the security check. Eventually, the attack may be discovered when execution reached $x-1$ and the instruction unmasked incorrectly, however this might happen long after the leak, or never.

We therefore propose modifying the simple scheme to prevent instructions from being moved from the locations they were originally installed in. The proposed modification is to combine simple hardware-software binding with SOFIA from Clercq et al. In SOFIA (described in Section 5.3.2.3), so that an instruction is masked using secret information, the PC value of the instruction and the PC value of the previous instruction. This produces a

Figure 5.3: When protected by only simple hardware-software binding, the code snippet on the left can be modified to leak sensitive data by moving code into a location reached by a `JUMP` operation.

device *and* memory location specific masking preventing instructions from being moved. If an instruction at location y were moved to location x it would no longer unmask correctly because the unmasking would use x and $x-1$ instead of y and $y-1$. Using the location in calculating the instruction masking value prevents instructions from being moved. SOFIA introduces the need for a tree-like structure at function entry adding a new instruction for each call to the function. However, as the jump source is protected by the application binding (Section 5.4.1), preventing instruction movement is the only additional property needed. The new masking equation will be as follows:

$$Op'_x = F(Op_x, Op'_{x-1}, x)$$

Where x is a memory location or PC value, Op_x is the instruction in location x in the application, Op'_x is the masked version of Op_x and F is the function used to unmask Op'_x. This extension binds each instruction to the specific device and also the specific memory location it is to be executed from. This ensures attacks such as demonstrated in Figure 5.3 are no longer possible.

5.4.3 Securing Instruction-Independent Transitions

After combining elements from simple hardware-software binding and the Clercq et al. scheme, the result is a masking scheme ensuring code immutability and that control flow instructions lead to the *instructions* intended. However, there is still a weakness associated with function code accessed by `CALL` and `RETURN`-type operations. The weakness is that

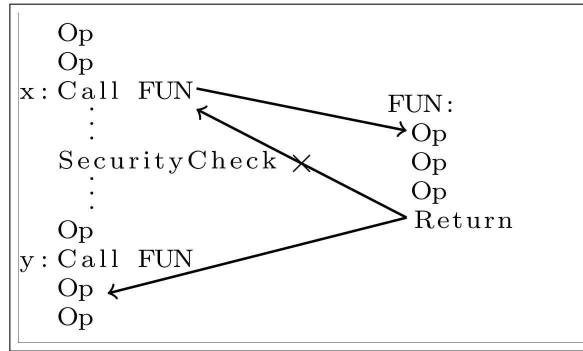


Figure 5.4: Changing return addresses can be used to skip security critical instructions.

while the scheme ensures that `CALL` instructions lead to the correct code it does not ensure `CALL` and `RETURN` instructions are executed in correct pairs.

Consider the code example in Figure 5.4. A function is called from two locations, x and y, and a security check is computed between the calls. As stated in Section 5.2.3, the attacker can interfere with any memory location including the stack. Therefore, the check can be avoided by waiting until `FUNC` is called and then changing the return address on the stack from $x+1$ to $y+1$. This avoids executing the security check and allows unauthorised access to later parts of the application. This attack is possible in the combined simple-binding-SOFIA scheme as no instructions are modified.

Therefore, a mechanism is required to ensure the destination of `RETURN` instructions matches (+1) the location of the `CALL` transferring control flow to the function. We propose an instruction set extension to transform `CALL` and `RETURN` into two part instructions. This change adds `CALL-IN` and `RETURN-OUT` to the instruction set. These instructions must be executed before `CALL` and after `RETURN` instructions respectively to prevent attacks as shown in Figure 5.4. The check will be to store a value provided with the `CALL-IN` when it executes and compare it with the value embedded in the `RETURN-OUT`. Check value mismatches will cause an error and stop execution. This ensures `RETURN` instructions transfer control flow to after the correct `CALL` instruction and not to after a different `CALL` to the same function. The extra instructions ensure secure execution of re-used functions with only two extra instructions per function call. The use of these instructions is demonstrated in Figure 5.5.

However, some attackers may attempt to avoid the `RETURN-OUT` check by returning to $x + 2$ or $y + 2$. Therefore, the scheme must protect against attackers skipping `CALL-IN`

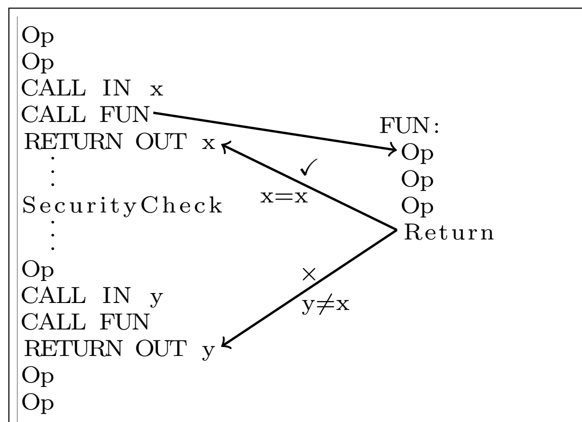


Figure 5.5: The extra operations added to the instruction set secure the `RETURN` after execution of re-used function code.

or `RETURN-OUT` instructions. This necessitates an extra check to ensure the instruction executed after a `CALL-IN` is a `CALL` and after a `RETURN` is a `RETURN-OUT`. This check must also ensure `CALL` is only executed after `CALL-IN`, and that `RETURN-OUT` may only follow a `RETURN`. Furthermore, `CALL-IN` not followed by `CALL` or `RETURN` not followed by `RETURN-OUT` also indicates an error. These checks are made using two one-bit processor flags. These flags are `CALL-CHECK` and `RETURN-CHECK` and are set by the executing of `CALL-IN` and `RETURN` and are reset by `CALL` and `RETURN-OUT`. These instructions must always be executed in series, so the check will ensure this by throwing an error if any instruction is executed while a flag is set other than the instruction to reset the flag. Similarly, an error must occur when an instruction to reset a flag is called when the flag is not set.

For example, a `CALL-IN` instruction executes setting the `CALL-CHECK` flag and storing the Sec Value. If the next instruction is an `ADD` instruction or any other instruction other than a `CALL`, an error will occur. However, if the next instruction is a `CALL`, the `CALL-CHECK` flag will be reset and the processor will execute the function. These additional registers protect the `CALL-IN` to `CALL` and `RETURN` to `RETURN-OUT` transitions and with the value checks ensure the security of instruction-independent transitions.

However, it is common for functions to call functions. In these cases, a second stack similar to the SCS described in Section 5.3.2.2 will be used. The check value is added to a stack with the execution of the `CALL-IN` instruction. The check is made when the `RETURN-OUT` instruction is executed by comparing the value in the instruction with the top of the Sec Stack. To aid performance, the check value could be stored in a processor register and

compared with the instruction check value with no memory access. After the check, the stack is popped to the register at any time before the next `RETURN-OUT` instruction. This may not help when `RETURN` and `RETURN-OUT` instructions are executed in a series of pairs, but may otherwise improve performance.

5.5 Implementation

This section describes the Secure-Execution Processor (SEP), the proof-of-concept implementation of the scheme proposed in Section 5.4. The details of the prototype are described in three subsections. Section 5.5.1 discusses design decisions made when developing the prototype and the hardware used. Section 5.5.2 describes the assembler developed to compile programs for execution on the SEP. Finally, Section 5.5.3 covers the hardware of the prototype and how it realises the scheme proposed.

5.5.1 Design Decisions

The implementation was developed on a Xilinx Spartan-6 FPGA SP601 evaluation board [124]. This board contains a Spartan-6 as well as block RAMs, LEDs, push-buttons, switches, an ethernet port, general-purpose I/O headers and a serial interface. All can be attached to designs loaded into the FPGA [124]. The prototype used block RAM to store software and the serial port and LEDs to demonstrate operation. A high-level system diagram of the prototype is shown in Figure 5.6.

As well as the hardware developed, a basic application was written to execute on the SEP. This is a simple program written in assembly language to output data to the connected PC via a UART interface[1]. Two tasks were carried out by the application: printing “Hello World!” to the terminal via the UART port and then flashing LEDs based on counter values. In total, the application (Appendix E) comprised 68 instructions, including multiple `CALL`/`RETURN` pairs and loops using conditional `JUMP` instructions.

[1] VHDL code from the PicoBlaze [123] soft-core processor was used to control the UART interface.

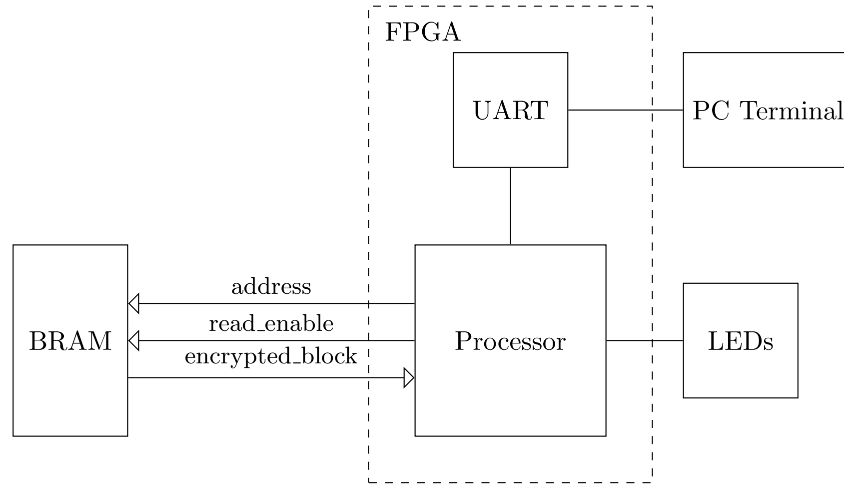


Figure 5.6: The prototype included serial and LED outputs to demonstrate correct execution.

5.5.1.1 Choice of F

Section 5.4.2 stated the masking function combine the operation, previous masked instruction and location of the operation. However, this idea requires more precise definition before implementation. As suggested in Section 3.4.3, the prototype uses a basic block structure for the application. In the basic block model, an application is split into blocks containing a fixed number of instructions where each block is executed in entirety or not at all. Execution of a basic block always starts with the first instruction and always finishes with the last instruction. `JUMP`s to or from the middle of a block are forbidden. This increases the overall size of the application as `NOP` (no-operation) instructions have to be added to comply with the model. Obviously, extending program length increases execution time. However, it allows the unmasking process more time to complete as once a block is ready there is a period of time based on the block length until the next block is needed. As a result, the prototype did not need to pause execution at any time to perform extra load or unmask instructions, unlike the simple hardware-software binding scheme from Chapter 3.

Basic blocks allow more efficient use of the block cipher by matching basic block size to the block size of the block cipher used. However, the function F needs to be explicit for implementation. The function used in the simple binding scheme is not suitable for two reasons. It is reproduced below with a basic block model translation:

$$Op'_x = Op_x \oplus F(Op'_{x-1})$$
$$B'_x = B_x \oplus B(B'_{x-1}),$$

where B_x is the basic block at location x and B'_x is the masked block. The first reason this is not suitable is the value x is absent from the computation, this allows attacks on the scheme as stated in Section 5.4.2. The second reason is that any bits flipped in B'_x will be flipped in the unmasked block, allowing an attacker to change instructions executed, violating Requirement R3. Therefore a new masking function is needed to protect instructions from modification. The properties needed of the masking function are: inclusion of the previous block, inclusion of the block address and prevention of attackers from making targeted changes to any bits in the unmasked instructions. To achieve these goals, the following functions for masking and unmasking will be used:

$$B'_x = B'_{x-1} \oplus Enc_{k \oplus x}(B_x)$$
$$B_x = Dec_{k \oplus x}(B'_x \oplus B'_{x-1}).$$

The exclusive-OR before decryption ensures attackers are unable to target a bit or bits to change in the resulting unmasked instructions. As each masking is now dependent on the location of the block, this is due to mixing the address into the encryption key, this does not decrease scheme security as long as the cipher used is secure against related key attacks. Since the masking key is now dependent on the location of the instruction, any rearrangement of basic blocks ensures they are not correctly unmasked. With a strong block cipher, decryption can be considered a pseudorandom function. Therefore an attacker modifying B_x or B'_{x-1} cannot make predictable output changes.

This masking function takes influence from SOFIA by Clercq et al. [32]. However, the difference between schemes is that this, updated, scheme replaces using the PC value of the previously *executed* instruction with that of the instruction *stored* in the memory location before the current instruction. This will simplify the process of producing bound executables for devices by minimising how sensitive the assembler must be to the context of each instruction in the application control-flow.

Cipher	Block Size	Key Size
AES	128	128
KATAN	32	80
	64	80
PRESENT	64	80
	64	128
PRINCE	64	128
RECTANGLE	64	80
SIMON	32	64
	64	128
SPECK	32	64
	64	128

Table 5.1: Lightweight block ciphers implemented by Maene and Verbauwhede in [78].

5.5.1.2 Cipher Used

As discussed previously, the function used to protect instructions needs to produce pseudorandom outputs unpredictable by an attacker. Block ciphers provide the guarantees required and so several were considered for use in securing the application. The list of ciphers considered is included in Table 5.1, all are considered lightweight apart from AES which is included for reference. However, each cipher has different block sizes, key lengths and complexities. The ciphers in Table 5.1 were considered because they have single-cycle implementations developed by Maene and Verbauwhede in 2015 [78], that are publicly available via Github [77]. The execution of a basic block may only take a small number of cycles, therefore a single-cycle implementation allows the scheme to be used with less time overheads. This also avoids complicated use of pauses and unpauses such as those encountered in Section 3.5.

For the prototype, the PRINCE cipher was used. PRINCE is a lightweight block cipher developed by Borghoff et al. in 2012 [16]. It features a 64 bit block size, 128 bit key size and has been the target of multiple competitions with prizes awarded based on attack strengths [16, 100, 101]. It was designed for efficiency in hardware and was the smallest implementation in the 64/128 class and fastest overall cipher in Maene and Verbauwhede's study [78]. Therefore, due to the high-security, good block size and presence of a published, good, single-cycle implementation, PRINCE was the best choice for use in F.

5.5.2 Assembler

To assist in preparing applications for the SEP by converting instructions into machine code, performing necessary encryptions and enforcing the program structure required, an

assembler was developed. The assembler takes as input a program written in SEP-assembly and outputs the memory contents to be loaded and executed by the SEP. This section describes the assembly code, instructions implemented and considerations for ensuring a legal program is output from the assembler.

For the SEP, a new assembly language dialect was written. The language developed is based on Picoblaze assembly [123] with most instructions removed - instructions not used by the example application were not implemented leaving a minimal instruction set. This instruction set allowed the opcode of an instruction to require only four bits, granting the SEP a 16-bit instruction set rather than the 18-bit size of PicoBlaze. However, the main benefit of shrinking the instruction size is instructions fit into block sizes of 2^n bits[2]. This enables basic blocks to easily match common block cipher block sizes of 64 or 128 bits, such as that of PRINCE [78]. The complete set of SEP opcodes is listed in Table 5.2.

As stated, SEP instructions are 16 bits wide. This is split between four bit opcodes and 12-bits for the operand/s. There are five types of operand in SEP instructions: register IDs, data values, IO addresses, memory addresses and security values. Register IDs are 4-bit numbers that select one of the 16 registers available to the SEP Arithmetic and Logic Unit (ALU). Data values are 8-bit numbers used by the ALU, these can be saved, AND-ed, added or subtracted into registers. IO addresses are 8-bit ID numbers of Input or Output devices attached to the SEP, IO addresses select devices to send data to or receive data from one of the ALU registers. Memory addresses are 12-bit addresses used with `CALL` or `JUMP` instructions to determine the new value for the PC (subject to a flag check for conditional `JUMP`s). Finally, security values are 12-bit values used by `CALL-IN` and `RETURN-OUT` instructions.

As well as translating human-readable instructions into SEP machine code, the assembler also enforces the basic block structure required. The assembler also adds the required `CALL-IN` and `RETURN-OUT` instructions and generated the security values. Therefore, translation and insertion of instructions are the main tasks of the assembler.

To enforce the basic block model there are two main criteria the assembler has to manage. As stated, a basic block always starts with the first instruction and finishes with the final instruction. Therefore any control-flow instruction must meet two criteria: always be

[2]Where $n \geq 4$

Opcode	Instruction	Operands
0	LOAD sX, sY	2
1	LOAD sX, kk	2
2	AND sX, kk	2
3	INPUT sX, pp	2
4	ADD sX, kk	2
5	SUB sX, kk	2
6	CALL aaa	1
7	CALL-IN ccc	1
8	JUMP aaa	1
9	JUMP NZ, aaa	1
A	JUMP C, aaa	1
B	RETURN	0
C	RETURN-OUT ccc	1
D	OUTPUT sX, pp	2

Table 5.2: Only the instructions needed for the test application were implemented and thus some common instructions are missing.

at the end of a basic block and always point towards the start of a basic block. These can intersect if one control-flow instruction is the destination of a second control flow instruction. In this case, the first control-flow instruction must be placed at the end of a basic block by inserting `NOP` instructions until this achieved. Then the second control-flow instruction must be modified to point toward the start of the block containing the first instruction. In most cases however, both criteria do not intersect and in the example program most instructions moved needed to be sent to the start of a basic block or the end.

Assembler program modification can thus be summarised in three points: ensuring `JUMP` instructions are only at the end of basic blocks, ensuring `JUMP` destinations are only at the start of blocks and adding `CALL-IN` and `RETURN-OUT` instructions where needed.

5.5.3 Hardware

This section describes the hardware element of the prototype. The hardware developed is a softcore processor, specified in VHDL, for synthesis to a Xilinx Spartan-6 FPGA. Use of an FPGA allows for more rapid prototyping and allows the design to be modified and shared more easily.

The SEP is comprised of six main components: the unmask unit, the data reader, the PC update, the PC/Sec stack, the IO handler and the ALU. Each component in the system completes different tasks and is responsible for updating different signals in the processor. A simplified model of the processor is shown in Figure 5.7. For clarity, some

of the helper signals have not been shown such as internal flags identifying the type of the last instruction in the current block, the zero and carry flags and access to the top of the stacks given to data reader and PC update. We will now describe the components in Figure 5.7.

The unmask unit receives data from memory and saves it as either an instruction block or helper data block. Use of a received block is calculated based on the current PC value and the state of the execution of the current instruction. The unmask unit also latches helper data used in sequential transitions. The PRINCE cipher component implemented by Maene and Verbauwhede [78] is included in the unmask unit. The cipher decrypts the next block depending on the PC value set during execution of the fourth instruction in the current block. Unmask unit tasks are time critical and follow the schedule in Figure 5.8.

The data reader sends addresses and enable signals to the memory to request data blocks. BRAM devices on the SP601 require the address and enable signals to be held for one cycle before a memory block is returned. The data reader uses an internal instruction state with the current PC value to time when to set and reset address and read enable signals. Like the unmask unit, data reader tasks are time critical and must follow the execution schedule shown in Figure 5.8.

The PC update maintains the current Program Counter value and updates it as each instruction is executed. The PC is updated based on the internal clock of the processor and also the current operation executing. Each instruction requires two cycles for execution, the PC is updated after the first cycle to indicate the location of the next instruction. PC update is critical to scheme security as it handles processing instruction transitions.

When a `CALL` instruction is executed, the PC/Sec stack stores the return address and the last `CALL-IN` security value. The PC/Sec stack, upon `CALL` instruction execution, stores the return address and the last `CALL-IN` security value. Security values from `CALL-IN` instructions are latched by the PC/Sec stack before they are pushed by the `CALL` instruction. The PC/Sec Stack displays the top of the stack to the data reader to aid execution of `RETURN` instructions. Access to the return address allows `RETURN` instructions to be processed similarly to unconditional `JUMP` instructions as the next address is known to the data reader.

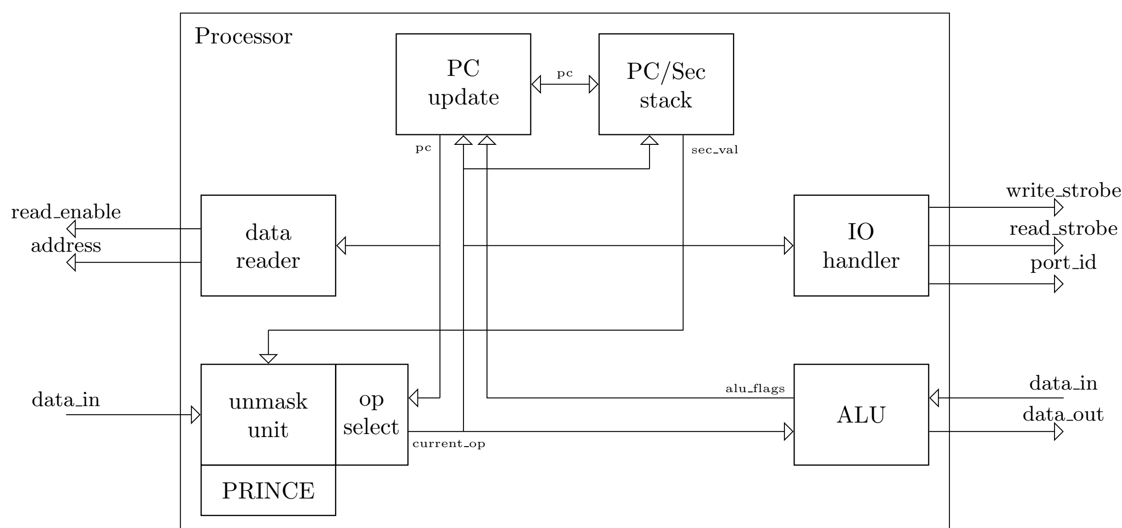


Figure 5.7: The Secure-Execution Processor consists of six main elements.

The IO handler sets the read/write strobe signals and the IO port ID values based on the current operation being executed. When an input or output signal is written or read, the write/read strobe signal is set high for one cycle. This synchronises SEP IO reads/writes with the attached input/output devices providing/receiving data.

Finally, the ALU executes arithmetic or logical instructions and maintains the processor registers and zero/carry flags. The ALU also connects registers with IO devices when `INPUT` or `OUTPUT` instructions are executed. As the instruction set (Table 5.2) does not include a `NOP` instruction, loading a register with itself is used instead as this makes no change to data stored. Specifically, `LOAD s0, s0` is added by the assembler to enforce the basic block model. However, this requires extra ALU logic to maintain program functionality. Normally, when a `LOAD` is executed, the carry flag is reset and the zero flag is only set if the register changed becomes equal to 0. However, this affects `JUMP NZ` instructions if a `LOAD s0, s0` is inserted before the instruction as flags will set/reset according to the value of s0, not last instruction before the `LOAD s0, s0`. Therefore, the ALU was modified to preserve the zero and carry flags if the two registers provided with a `LOAD` instruction are equal. This prevents `NOP/LOAD s0, s0` instructions from effecting program logic.

When executing the application, different transitions between basic blocks have different requirements for the data needed. For example, if the transition from the current block to the next is sequential, the only memory load required is loading the next block in memory. In the sequential case, the helper data for the next block is the current block

Op	Cycle	PC	Type of JUMP at end of block		
			Sequential	Unconditional	Conditional
00	0	00			
	1	01			
01	2	01			
	3	10			load B'_x
10	4	10			load B'_{j-1}
	5	11	load B'_x	load B'_{j-1}	load B'_j
11	6	11	load B'_{x+1}	load B'_j	load B'_{x+1}
	7	00	Unmask next block		
$[x+1, j] \| 00$	8	00	Execute next block		

Table 5.3: Sequential transitions require less memory accesses than unconditional JUMPs which require less than conditional JUMPs.

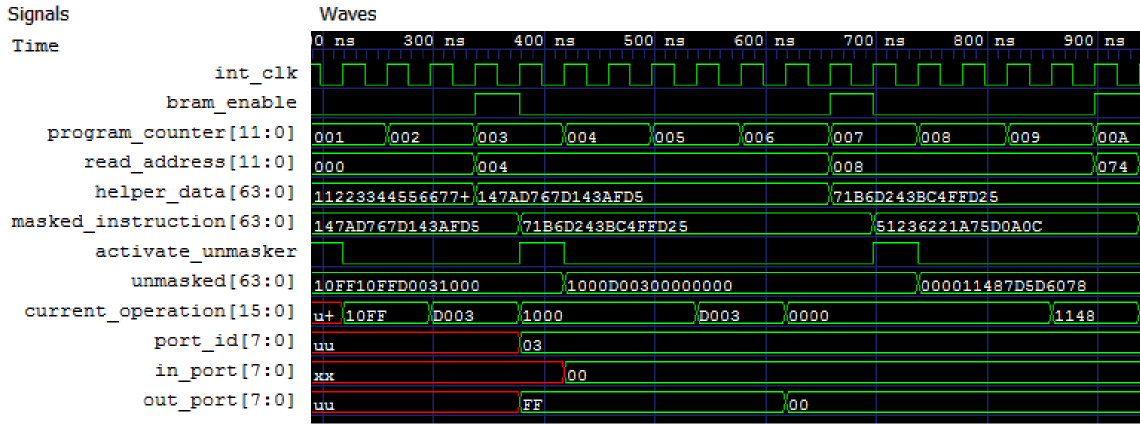


Figure 5.8: The Secure-Execution Processor updates each signal according to a strict schedule.

which is latched to protect sequential transitions. The most demanding case is when the final transition is conditional as this requires the next block, the JUMP block and the JUMP helper data to all be read by the data reader. As only one memory read can be performed per cycle, the timing of the memory reads for each transition must be met precisely. A table of the data read actions for each type of transition is found in Table 5.3. Note: each instruction requires two cycles for execution. Changes to the program signals in a simulation of the prototype are shown in Figure 5.8.

To enforce the security requirements of the proposed scheme, a "kill switch" was added to the design of the processor. Each component shown in Figure 5.7 is driven by an internal clock, which is defined simply by:

$$int_clk = clk \& kill_switch,$$

where *kill switch* is initialised as 1 and set to 0 whenever a security problem occurs.

We define this security problems as: `CALL-IN/RETURN-OUT` security value mismatch, `CALL` without `CALL-IN` (and vice versa), `RETURN` without `RETURN-OUT` (and vice versa).

5.6 Analysis

This section analyses the scheme proposed and implemented in this work. First, the proposal in Section 5.4 is checked against the requirements derived in Section 5.2.4. The second half of this section analyses the implementation described in Section 5.5.

5.6.1 Requirements Analysis

Recall that Section 5.2.4 listed six requirements for the proposed scheme. Requirement R1 is to provide software confidentiality to ensure attackers cannot reproduce the software and violate the software binding. A function F is used to conceal the instructions making up the application executed by the SEP. The F implemented includes PRINCE for encrypting/decrypting blocks of application instructions and ensuring software confidentiality.

Requirement R2 needs each application to be bound to just one device. Each device is initialised with a different encryption key, so each masking is unique to a single device. No other device other than the intended device will be able to unmask the blocks and execute the program.

Requirement R3 requires the scheme to prevent attackers making meaningful changes to the application. Section 5.5.1.1 describes the F used in the implementation and how the use of PRINCE prevents attackers from predicting the consequences of any changes made.

Requirement R4 demands that sequential transitions between instructions be secured. This is achieved by the block chaining in the unmaking functions. The previous executed block is used as helper data without being loaded from memory, this ensures the security of sequential transitions..

Requirement R5 requires that instruction-dependent transitions, such as those following

`JUMP` and `CALL` instructions are secured. This is achieved by including the location of the block into the masking function, ensuring the block jumped to cannot execute correctly if it has been modified as each masking is address dependent.

Finally, Requirement R6 states instruction-independent transitions (such as after `RETURN` instructions) must be secured. In the proposed scheme this is met by using `CALL-IN` and `RETURN-OUT` instructions with the PC/Sec Stack to store and check the security values. These instructions must be executed before and after `CALL` and `RETURN` instructions, and the security values must match for execution to continue otherwise the kill switch activates. Instruction-independent transitions are secured by the proposed scheme.

5.6.2 Implementation Analysis

The prototype implemented the scheme proposed that meets the requirements listed in Section 5.2.4. This subsection describes the strengths and limitations of the prototype.

The main design decision required when implementing the proposed scheme was to use the basic block model and PRINCE for encrypting the basic blocks. PRINCE was chosen due to the low latency and small size of the single-cycle implementation of Maene and Verbauwhede [78]. The single cycle implementation aided the SEP to meet deadlines for instruction readiness. However, as shown in Table 5.3, for each possible transition there were free cycles available while the block was executing. Therefore, the single cycle implementation is faster than required and a slower implementation could be used if the timing of conditional `JUMP` instructions could be met. Multiple unmask operations while executing the block might be required in this case to avoid the need for pauses. A current advantage of the implementation over other schemes is the avoidance of delay in the execution of instructions.

Cipher choice determined the size of basic blocks to 64 bits/four instructions. This limits the number of `NOP` instructions needed to be inserted to three before a `JUMP`. However, in the case of a `CALL` instruction finishing a block, four additional instructions are needed, as we will need a `CALL-IN` as well as three `NOP`s. Furthermore, the need for control-flow instructions to point towards the start of basic blocks requires even more `NOP` instructions to be added. Due to the extra instructions added, the test application was expanded

from containing 68 operations to containing 160 operations after the basic block model conversion and CALL-IN/RETURN-OUT instruction addition. However, enforcing the basic block model only extended the application to 108 instructions, the proposed scheme added 52 instructions more than a basic block-only scheme. The example program contained 14 CALL instructions, meaning a minimum of 28 instructions needed to be added, and the remaining 24 additions were required to fit the basic block model. Therefore, despite the implementation requiring no pauses in execution, it does still reduce application performance. To achieve the extra security requirements, the length of the program (and thus the number of cycles required to execute) increased substantially.

5.7 Conclusion

In this paper we presented a novel work combining two previously studied problems, Secure Application Execution and Platform Specific Execution. The proposed scheme ensures applications executing on a device were put there by the manufacturer and are executing correctly from start to finish. The scheme presented is a framework that could be implemented in various ways, one instantiation was implemented to demonstrate the scheme in practice. A description of the prototype was provided to make clear the steps followed and the constraints present.

The proposed scheme combines SOFIA, by Clercq et al., and our scheme from Chapter 3. The extension over Chapter 3 was to also consider secure execution as well as platform-specific execution. Our proposed used elements from SOFIA, the location-specific masking operation, but also simplified the protection of reused code; simplifying the scheme and removing the need to handle certain exceptional cases. This resulted in a scheme reaching wider security goals than previously while retaining the consistent application and execution from our previous scheme.

Further work in this area would look to different methods for using the CALL-IN/RETURN-OUT security values. One potential approach could be to remove the need for a "Sec Stack" and instead use a multiplicative counter if the different Sec Values were prime numbers. Another extension of this work would expand the prototype to a fuller instruction set including storing values in memory. With a full instruction set, an in-depth study of the

performance cost of the scheme could be carried out by compiling real-world applications and libraries for execution on the SEP.

Chapter 6

Conclusion

Contents

6.1 Summary and Conclusions 124

6.2 Future Work 125

This chapter concludes the thesis by summarising the work presented and by considering areas for future work in the area of binding hardware and software in computing devices.

6.1 Summary and Conclusions

This work has considered methods for protecting computer systems from software tampering. Primarily, the use case considered has been embedded computing and IoT devices, however the work would apply equally to computer systems in other domains. The consequences of attacks on computer systems can be severe and so we must continue to develop security solutions. This thesis has presented three distinct studies that develop, or consider the deployment of, security solutions.

Primarily, this work concerns protecting systems by binding hardware and software together. Previous work in this area is limited so initial work explored the problem space by developing requirements for a binding scheme. From the initial set of requirements, a solution was developed to meet them. Many potential instantiations of the proposed binding were suggested, however a simple LFSR-based scheme was implemented to explore the cost of the scheme, the sources of any delays in execution and the overall efficacy of the scheme.

As few hardware-software binding schemes have been proposed, there has been very little study into deploying them. Therefore, in the setting of a smart city, provisioning software bound to devices was explored and examined. Three models were developed to describe modes of software distribution. From the three models, requirements and protocols were developed for provisioning software and generating binding keys in a smart city. The protocols were analysed using Tamarin Prover and their security proven. Each protocol was implemented using a laptop and Raspberry Pi 3 to test their performance.

Finally, the initial problem and solution were expanded to also consider a similar, but distinct problem - securing application execution. A new binding scheme was proposed to ensure deployed software executes correctly and only the device it is intended for. The Secure-Execution Processor was developed to demonstrate the new scheme and it uses it to secure all installed applications. Using high speed cryptographic hardware, the software executed with no penalty to instruction throughput although enforcing program compliance with the scheme added significant overheads.

6.2 Future Work

Binding hardware and software has large scope for future research. The work presented defined unidirectional and bidirectional schemes but did not exhaustively list possible schemes. Many more may be developed to bind software and hardware.

This work only considered creating application-level bonds. However, one extension would be to expand on the work of Atallah et al. (Section 2.4.2) and consider methods of creating algorithm-level bonds. In their work, Atallah et al. considered the RSA algorithm and bound it to the hardware by including device specific information into the computation. Algorithm-binding has more potential to be discovered mixed into schemes like those presented in this document or by considering distinct approaches.

Future work may also include studying the cost of masking data stored in and loaded from main memory. The schemes implemented in Chapters 3 and 5 ran applications only requiring processor registers and with no external memory used. Extending the prototypes to secure the mutable data stored in main memory would increase understanding of the costs of using hardware-software binding to protect real-world applications.

Other future work could also study the impact of hardware-software binding by testing with established benchmark code. This would require expanding the prototype implementations to secure data in main memory and then expanding the instruction set of the SEP before installing or cross-compiling several known libraries. With these extensions, the performance cost of each scheme could be assessed and compared other, similar security schemes. Expanded implementations could also include different masking functions to test the cost of different approaches and cryptographic function choices.

Finally, a larger area to be researched is considering how binding hardware and software may apply in alternative domains to embedded systems, such as distributed computing or in securing cloud applications. A wide study of potential use cases would focus future work to domains where ensuring platform specific execution can make the greatest benefit to application security.

Appendix A

Simple Binding Scheme Test Application Code

```
CONSTANT LED_PORT, 02
CONSTANT UART_TX_STATUS, 00
CONSTANT UART_RX_READ, 01
CONSTANT UART_TX_DATA_IN, 01

start:
  OUTPUTK FF, 1
  LOAD s0, s0
  LOAD s0, s0
  OUTPUTK 00, 1
  LOAD s0, s0
  LOAD s0, s0
  LOAD s1, "H"
  CALL waitForUARTFree
  OUTPUT s1, UART_TX_DATA_IN
  LOAD s1, "e"
  CALL waitForUARTFree
  OUTPUT s1, UART_TX_DATA_IN
  LOAD s1, "l"
  CALL waitForUARTFree
  OUTPUT s1, UART_TX_DATA_IN
  LOAD s1, "l"
  CALL waitForUARTFree
  OUTPUT s1, UART_TX_DATA_IN
  LOAD s1, "o"
  CALL waitForUARTFree
  OUTPUT s1, UART_TX_DATA_IN
  LOAD s1, " "
  CALL waitForUARTFree
  OUTPUT s1, UART_TX_DATA_IN
  LOAD s1, "W"
  CALL waitForUARTFree
  OUTPUT s1, UART_TX_DATA_IN
  LOAD s1, "o"
  CALL waitForUARTFree
  OUTPUT s1, UART_TX_DATA_IN
  LOAD s1, "r"
  CALL waitForUARTFree
  OUTPUT s1, UART_TX_DATA_IN
  LOAD s1, "l"
  CALL waitForUARTFree
  OUTPUT s1, UART_TX_DATA_IN
  LOAD s1, "d"
  CALL waitForUARTFree
  OUTPUT s1, UART_TX_DATA_IN
  LOAD s1, "!"
  CALL waitForUARTFree
  OUTPUT s1, UART_TX_DATA_IN
```

```
  LOAD s1, " "
  CALL waitForUARTFree
  OUTPUT s1, UART_TX_DATA_IN
  LOAD s1, 0D
  CALL waitForUARTFree
  OUTPUT s1, UART_TX_DATA_IN
  JUMP ledStart

waitForUARTFree:
  INPUT s0, UART_TX_STATUS
  AND s0, 04
  JUMP NZ, waitForUARTFree
  RETURN
```

```
ledStart: LOAD s0, 00
loop:   OUTPUT s0, LED_PORT
        LOAD s2, FF
x:LOAD s1, FF
y:SUB s1, 01
        JUMP NZ, y
        SUB s2, 01
        JUMP NZ, x
        ADD s0, 01
        JUMP C, reset
back:   JUMP loop
reset: LOAD s0, 00
        JUMP back
```

Appendix B

Tamarin Model for Device-Centric Provisioning

```
theory DeviceCentric
begin

builtins: diffie-hellman, symmetric-encryption, signing, hashing

rule Register_pk:
  [ Fr(~ltkA) ]
  -->
  [ !Ltk($A, ~ltkA), !Pk($A, pk(~ltkA)), Out(pk(~ltkA)) ]

rule create_O_cert:
  let
    ocertnew = <$O, pkO, ~cid>
  in
  [ Fr(~cid), !Pk($O, pkO) ]
--[ OnlyOnce(), CreateOCert(ocertnew), FirstOCert(ocertnew) ]->
  [ !OCert(ocertnew), Out(ocertnew) ]

rule update_O_cert:
  let
    ocertold = <$O1, pkO1, ~cid1>
    ocertnew = <$O, pkO, ~cid>
  in
  [ Fr(~cid), !Pk($O, pkO), !OCert(ocertold) ]
```

```
--[ ValidCert(ocertold), ReCert($O, $O1, ~cid, ~cid1)
  , OCertExpired(ocertold), OCertByUpdate(ocertnew)
  , CreateOCert(ocertnew) ]->
  [ !OCert(ocertnew), Out(ocertnew) ]

rule reveal_session_key:
  [ !SessionKey(seshk, nc1, nc2) ]
--[ RevealSessionKey(seshk) ]->
  [ Out(seshk) ]

//First Message (O->D)
rule O1:
  let
    a = ~a
    nc1 = ~nc1
    pid = ~pid
    D = $D
    O = $O
    theOCert = <$O, pk(ltkO), cid>
    ga = 'g' ^ a
    m1 = <'message1', $O, $D, nc1, ga, theOCert>
    m1Sig = sign(h(m1), ltkO)
  in
  [ !Ltk($O, ltkO), !OCert(theOCert), Fr(nc1), Fr(a), Fr(pid) ]
--[ O1(pid), OUT_O1(pid, m1), Start(pid, $O, 'operator')
  , Neq(ga, 'g'), ValidCert(theOCert), SendFreshNonce(O, D, nc1)
  , Role(O, 'operator', pid) ]->
  [ Out(<m1, m1Sig>), St_O1($O, $D, nc1, a, pid, theOCert) ]

//Second Message (D->O)
rule D1:
  let
    b = ~b
    nc2 = ~nc2
```

```
    c = ~c
    D = $D
    O = $O
    ga = 'g' ^ new_a
    gb = 'g' ^ b
    seshK = ga ^ b
    recOCert = <$O, pk(ltkO), cid>
    m1 = <'message1', $O, $D, nc1, ga, recOCert>
    m1Sig = sign(h(m1), ltkO)
    m2 = <'message2', $D, $O, nc1, nc2, gb, senc('g' ^ c, seshK)>
    m2Sig = sign(h(m2), ltkD)
  in
  [ !Ltk($D, ltkD), !Ltk($O, ltkO), !OCert(recOCert), Fr(nc2)
  , Fr(b), Fr(c), Fr(pid), In(<m1, m1Sig>) ]
--[ D1(pid), IN_D1(pid, m1), Start(pid, $D, 'device')
  , SendAuth($D, $O, nc1), SendFreshNonce(D, O, nc2)
  , ValidCert(recOCert), In_d1_nc1_ga(nc1, ga, m1), Neq(ga, 'g')
  , Neq(seshK, 'g'), OUT_D1(pid, m2), Role(D, 'device', pid)
  , EstablishSessionKey(O, D, nc1, nc2, seshK) ]->
  [ Out(<m2, m2Sig>), !SessionKey(seshK, nc1, nc2)
  , St_D1($D, $O, nc1, nc2, ga ^ b, c, recOCert, pid) ]

//Third Message (O->D)
rule O2:
  let
    a = ~a
    DHd = ~DHd
    nc1 = ~nc1
    pid = ~pid
    SWid = ~SWid
    O = $O
    D = $D
    theOCert = <$O, pk(ltkO), cid>
    App = <'sw', SWid>
```

```
    gb = 'g' ^ b
    seshk = gb ^ ~a
    gc = 'g' ^ new_c
    gd = 'g' ^ DHd
    gcenc = senc(gc, seshk)
    maskingKey = gc ^ DHd
    MaskedApp = senc(App, maskingKey)
    AppHash = h(<App, 'SWHash'>)
    m2 = <'message2', $D, $O, nc1, nc2, gb, gcenc>
    m2Sig = sign(h(m2), ltkD)
    m3 = <'message3', O, D, nc2, senc(<MaskedApp, AppHash, gd>, seshk)>
    m3Sig = sign(h(m3), ltkO)
  in
  [ !Ltk($O, ltkO), !Ltk($D, ltkD)
  , St_O1($O, $D, nc1, a, pid, theOCert), In(<m2, m2Sig>)
  , Fr(DHd), Fr(SWid), !OCert(theOCert) ]
--[ O2(pid), IN_O2(pid, m2), Provisioned($D, $O, App, pid)
  , SendAuth($O, $D, nc2), Authentic($O, $D, nc1)
  , AppInstallSend($O, $D, App, nc1, nc2, theOCert)
  , Neq(gb, 'g'), Neq(gc, 'g'), Neq(gd, 'g'), OUT_O2(pid, m3)
  , Role(O, 'operator', pid), ValidCert(<$O, pk(ltkO), cid>) ]->
  [ Out(<m3, m3Sig>) ]


//SW Installation
rule D2:
  let
    c = ~c
    pid = ~pid
    nc2 = ~nc2
    iid = ~iid
    D = $D
    O = $O
    gd = 'g' ^ DHd
```

```
    MaskSW = senc(SW, gd ^ c)
    AppHash = h(<SW, 'SWHash'>)
    stOCert = <$O, pk(ltkO), cid>
    m3 = <'message3', O, D, nc2, senc(<MaskSW, AppHash, gd>, sessionkey)>
    m3Sig = sign(h(m3), ltkO)
  in
  [ !Ltk($O, ltkO)
  , St_D1($D, $O, nc1, nc2, seshk, c, stOCert, pid)
  , In(<m3, m3Sig>), Fr(iid) ]
--[ D2(pid), IN_D2(pid, m3), Authentic($D, $O, nc2)
  , ValidCert(stOCert), Role(D, 'device', pid)
  , Installed($D, $O, SW, iid, nc1, nc2, stOCert) ]->
  [    ]

//Restrictions
restriction OnlyOnce:
  "All #i #j. OnlyOnce()@#i & OnlyOnce()@#j ==> #i = #j"

restriction Equality:
  "All x y #i.
    Eq(x, y) @i
    ==> x = y"

restriction Inequality:
  "All x y #i.
    Neq(x, y) @i
    ==> not (x = y)"

restriction CertValidity:
  "All cert #i.
    ValidCert(cert) @i
    ==> (Ex #j.
            CreateOCert(cert)@#j
            & j < i)
```

```
        & not (Ex #k.
            OCertExpired(cert)@#k
            & k < i)"

restriction FreshAuth:
  "All A B n #i #j.
    SendAuth(A, B, n) @i
    & SendAuth(A, B, n) @j
    ==> #i = #j"

restriction FreshNonce:
  "All A B n #i #j.
    SendFreshNonce(A, B, n) @i
    & SendFreshNonce(A, B, n) @j
    ==> #i = #j"

//Source Lemmas
lemma types [sources]:
  " All tid m1 #i.
      IN_D1(tid, m1) @i
      ==> (Ex #j. KU(m1) @ #j & j < i)
        | (Ex tidO #j. OUT_O1(tidO, m1) @j & j < i)
   & All tid m2 #i.
      IN_O2(tid, m2) @i
      ==> (Ex #j. KU(m2) @j & j < i)
        | (Ex tidD #j. OUT_D1(tidD, m2) @j & j < i)
   & All tid m3 #i.
      IN_D2(tid, m3) @i
      ==> (Ex #j. KU(m3) @j & j < i)
        | (Ex tidO #j. OUT_O2(tidO, m3) @j & j < i)"

//Cert Lemmas
lemma CertChain [reuse, use_induction]:
  "All cert #i #j.
```

```
    CreateOCert(cert) @i
    & OCertByUpdate(cert) @j
    ==> (Ex first #k.
            FirstOCert(first) @k
            & k < i
            & k < j)"

lemma OCertChain2 [use_induction]:
  "All cert #i.
    CreateOCert(cert) @i
    ==> (Ex #j. OCertByUpdate(cert) @j)
        | (Ex #j. FirstOCert(cert) @j)"

//OCert Ordering
lemma CreateThenExpire [reuse, use_induction]:
  "All cert #i.
    OCertExpired(cert)@i
    ==> (Ex #j.
          CreateOCert(cert) @#j
          & j < i)"

//Entity authentication
lemma easy_auth [use_induction, reuse]:
  "All A B nc #i.
    Authentic(A, B, nc) @i
    ==> (Ex #j. SendAuth(B, A, nc)@j
          & j < i)"

lemma fresh_entity_authentication [use_induction, reuse]:
  "All A B nc #i.
    Authentic(A, B, nc)@i
    ==> (Ex #j. SendAuth(B, A, nc)@j
        & j < i
        & not (Ex #i2. SendAuth(B, A, nc)@i2
```

```
                    & (#i2 = #i)))"

//Post-compromise secrecy and secure key generation
lemma pcs_key:
  "All A B n1 n2 key #i #j.
    K(key)@i
    & EstablishSessionKey(A, B, n1, n2, key)@j
    ==> (Ex #k.
          RevealSessionKey(key)@#k)"

//Installation Replay Resistance
lemma replay_resist_install:
  "All D O SW iid nc1 nc2 theOCert #i.
    Installed(D, O, SW, iid, nc1, nc2, theOCert)@i
    ==> Ex #j.
          AppInstallSend(O, D, SW, nc1, nc2, theOCert)@#j
          & #j < #i
          & Authentic(D, O, nc2)@i"

//Single Operator
lemma single_operator:
  "All D O SW iid nc1 nc2 theOCert #i.
    Installed(D, O, SW, iid, nc1, nc2, theOCert)@i
    ==> not (Ex #j.
              OCertExpired(theOCert) @j
              & j < i)"

end
```

Appendix C

Tamarin Model for Authority-Centric Provisioning

```
theory OperatorCentricPKI
begin

builtins: diffie-hellman, symmetric-encryption, signing, hashing
functions: mask/2

rule Register_pk:
  [ Fr(~ltkA) ]
--[ PublicKeyRegistered($A, pk(~ltkA)) ]->
  [ !Ltk($A, ~ltkA), !Pk($A, pk(~ltkA)), Out(pk(~ltkA)) ]

rule create_O_cert:
let
  ocertnew = <$O, pkO, ~cid>
in
  [ Fr(~cid), !Pk($O, pkO) ]
--[ OnlyOnce(), CreateOCert(ocertnew), FirstOCert(ocertnew) ]->
  [ !OCert(ocertnew), Out(ocertnew) ]

rule update_O_cert:
let
  ocertold = <$O1, pkO1, ~cid1>
  ocertnew = <$O, pkO, ~cid>
in
```

```
  [ Fr(~cid), !Pk($O, pkO), !OCert(ocertold) ]
--[ ValidCert(ocertold), ReCert($O, $O1, ~cid, ~cid1)
  , OCertExpired(ocertold), OCertByUpdate(ocertnew)
  , CreateOCert(ocertnew) ]->
  [ !OCert(ocertnew), Out(ocertnew) ]

rule reveal_session_key:
  [ !SessionKey(seshk, nc1, nc2) ]
--[ RevealSessionKey(seshk) ]->
  [ Out(seshk) ]

//First Message (O->A)
rule O1:
let
  DHa = ~DHa
  nc1 = ~nc1
  pid = ~pid
  A = $A
  O = $O
  theOCert = <$O, pk(ltkO), cid>
  ga = 'g' ^ DHa
  m1 = <'message1', O, A, nc1, ga, theOCert>
  m1Sig = sign(h(m1), ltkO)
in
  [ !Ltk($O, ltkO), !OCert(theOCert), Fr(nc1), Fr(DHa), Fr(pid) ]
--[ O1(pid), OUT_O1(pid, nc1, ga, m1, m1Sig)
  , Role(O, 'operator', pid), Neq(ga, 'g'), ValidCert(theOCert)
  , SendFreshNonce(O, A, nc1), HonestDHExp(DHa), HonestDH(ga, DHa)
  ]->
  [ Out(<m1, m1Sig>, St_O1(O, A, nc1, DHa, pid, theOCert) ]

//Second Message (A->O)
rule A1:
let
```

```
  b = ~b
  nc2 = ~nc2
  A = $A
  O = $O
  ga = 'g' ^ new_a
  gb = 'g' ^ b
  seshK = ga ^ b
  recOCert = <O, pk(ltkO), cid>
  m1 = <'message1', O, A, nc1, ga, recOCert>
  m1Sig = sign(h(m1), ltkO)
  m2 = <'message2', A, O, nc1, nc2, gb>
  m2Sig = sign(h(m2), ltkA)
in
  [ !Ltk(A, ltkA), !Ltk(O, ltkO), !OCert(recOCert), Fr(nc2), Fr(b)
  , Fr(pid), In(<m1, m1Sig>) ]
--[ A1(pid), IN_A1(pid, nc1, ga, m1, m1Sig)
  , Role(A, 'authority', pid), SendAuth(A, O, nc1)
  , SendFreshNonce(A, O, nc2), ValidCert(recOCert), HonestDHExp(b)
  , HonestDH(gb, b), Neq(ga, 'g'), Neq(gb, 'g'), Neq(seshK, 'g')
  , OUT_A1(pid, m2), EstablishKey(seshK)
  , EstablishSessionKey(O, A, nc1, nc2, seshK) ]->
  [ Out(<m2, m2Sig>)
  , St_A1(A, O, nc1, nc2, ga, b, ga ^ b, recOCert, pid)
  , !SessionKey(seshK, nc1, nc2) ]

//Third Message (O->A)
rule O2:
let
  DHa = ~DHa
  nc1 = ~nc1
  pid = ~pid
  SWid = ~SWid
  O = $O
  A = $A
```

```
  D = $D
  theOCert = <O, pk(ltkO), cid>
  App = <'sw', SWid>
  AppHash = h(<App, 'swhash'>)
  gb = 'g' ^ b
  seshK = gb ^ ~DHa
  m2 = <'message2', A, O, nc1, nc2, gb>
  m2Sig = sign(h(m2), ltkA)
  m3 = <'message3', O, A, nc2, senc(<D, App, AppHash>, seshK)>
  m3Sig = sign(h(m3), ltkO)
in
  [ !Ltk($O, ltkO), !Ltk($A, ltkA)
  , St_O1(O, A, nc1, DHa, pid, theOCert), In(<m2, m2Sig>, Fr(SWid)
  , !OCert(theOCert) ]
--[ O2(pid), IN_O2(pid, nc2, m2), Role(O, 'operator', pid)
  , SendAuth(O, A, nc2), Authentic(O, A, nc1)
  , AppTransfer(O, A, D, App, nc1, nc2, theOCert)
  , Neq(gb, 'g' ^ DHa), Neq(gb, 'g'), Neq(seshK, 'g')
  , ValidCert(<$O, pk(ltkO), cid>, OUT_O2(pid, m3) ]->
  [ Out(<m3, m3Sig>) ]

rule A2:
let
  c = ~c
  pid = ~pid
  nc2 = ~nc2
  nc3 = ~nc3
  tid = ~tid
  A = $A
  D = $D
  O = $O
  SWHash = h(<SW, 'swhash'>)
  stOCert = <$O, pk(ltkO), cid>
  gc = 'g' ^ c
```

```
  m3 = <'message3', O, A, nc2, senc(<D, SW, SWHash>, seshk)>
  m3Sig = sign(h(m3), ltkO)
  m4 = <'message4', A, D, nc3, gc>
  m4Sig = sign(h(m4), ltkA)
in
  [ !Ltk(O, ltkO), !Ltk(A, ltkA), In(<m3, m3Sig>), Fr(c), Fr(tid)
  , Fr(nc3), St_A1(A, O, nc1, nc2, ga, b, seshk, stOCert, pid) ]
--[ A2(pid), IN_A2(pid, m3), Role(A, 'authority', pid)
  , Authentic($A, $O, nc2), ValidCert(stOCert)
  , SWReceived($A, $O, $D, SW, tid, nc1, nc2, stOCert)
  , SendFreshNonce(A, D, nc3), HonestDHExp(c), HonestDH(gc, c)
  , Neq(gc, 'g'), Neq(gc, ga), Neq(gc, 'g'^b), OUT_A2(pid, m4) ]->
  [ St_A2(A, O, D, nc1, nc2, nc3, seshk, ga, b, c, stOCert, pid, SW)
  , Out(<m4, m4Sig>) ]

rule D1:
let
  DHd = ~DHd
  e = ~e
  nc4 = ~nc4
  pid = ~pid
  A = $A
  D = $D
  O = $O
  gc = 'g'^c
  gd = 'g'^DHd
  ge = 'g'^e
  seshK = gc^DHd
  m4 = <'message4', A, D, nc3, gc>
  m4Sig = sign(h(m4), ltkA)
  m5 = <'message5', D, A, nc3, nc4, gd,
                                senc(<'dhexponents', ge>, seshK)>
  m5Sig = sign(h(m5), ltkD)
in
```

```
  [ !Ltk(D, ltkD), !Ltk(A, ltkA), In(<m4, m4Sig>, Fr(nc4), Fr(DHd)
  , Fr(e), Fr(pid) ]
--[ D1(pid), IN_D1(pid, nc3, m4), Role(D, 'device', pid)
  , SendAuth(D, A, nc3), SendFreshNonce(D, A, nc4)
  , EstablishSessionKey(A, D, nc3, nc4, seshK)
  , EstablishKey(seshK), HonestDHExp(DHd), HonestDH(gd, DHd)
  , HonestDHExp(e), HonestDH(ge, e), Neq(DHd, e), Neq(gc, 'g')
  , Neq(gd, 'g'), Neq(ge, 'g'), OUT_D1(pid, m5) ]->
  [ St_D1(D, A, nc3, nc4, DHd, e, seshK, pid)
  , !SessionKey(nc3, nc4, seshK), Out(<m5, m5Sig>) ]

rule A3:
let
  nc2 = ~nc2
  nc3 = ~nc3
  b = ~b
  c = ~c
  f = ~f
  pid = ~pid
  A = $A
  D = $D
  O = $O
  gd = 'g' ^ new_d
  ge = 'g' ^ new_e
  gf = 'g' ^ f
  seshKAD = gd ^ c
  maskingKey = ge ^ f
  MaskedSW = mask(SW, maskingKey)
  m5 = <'message5', D, A, nc3, nc4, gd,
                            senc(<'dhexponents', ge>, seshKAD)>
  m5Sig = sign(h(m5), ltkD)
  m6 = <'message6', A, D, nc3, nc4,
                            senc(<MaskedSW, h(SW), gf>, seshKAD)>
  m6Sig = sign(h(m6), ltkA)
```

```
in
  [ !Ltk(A, ltkA), !Ltk(D, ltkD), In(<m5, m5Sig>, Fr(f)
  , St_A2(A, O, D, nc1, nc2, nc3, seshKAO, ga, b, c,
                                            stOCert, pid, SW) ]
--[ A3(pid), IN_A3(pid, nc4, m5), Role(A, 'authority', pid)
  , SendAuth(A, D, nc4), Authentic(A, D, nc3), ValidCert(stOCert)
  , SendAppInstall(O, A, D, nc1, nc2, nc3, nc4, SW, stOCert)
  , HonestDHExp(f), HonestDH(gf, f), Neq(gd, 'g', Neq(ge, 'g')
  , Neq(gf, 'g'), Neq(seshKAD, 'g'), Neq(maskingKey, 'g')
  , EstablishMaskingKey(A, D, nc3, nc4, maskingKey)
  , EstablishKey(maskingKey), OUT_A3(pid, m6) ]->
  [ Out(<m6, m6Sig>), !MaskingKey(maskingKey, nc3, nc4) ]

rule D2:
let
  DHd = ~DHd
  e = ~e
  nc4 = ~nc4
  iid = ~iid
  A = $A
  D = $D
  O = $O
  gf = 'g' ^ new_f
  maskingKey = gf ^ e
  MaskedSW = mask(SW, maskingKey)
  m6 = <'message6', A, D, nc3, nc4,
                              senc(<MaskedSW, h(SW), gf>, seshK)>
  m6Sig = sign(h(m6), ltkA)
in
  [ !Ltk(A, ltkA), St_D1(D, A, nc3, nc4, DHd, e, seshK, pid)
  , Fr(iid), In(<m6, m6Sig>) ]
--[ D2(pid), IN_D2(pid, m6), Role(D, 'device', pid)
  , Authentic(D, A, nc4), Installed(D, O, A, SW, nc3, nc4, iid)
  , Neq(gf, 'g'), Neq(gf, 'g'^DHd), Neq(gf, 'g'^e)
```

```
  , Neq(seshK, 'g') ]->
  [    ]

//Restrictions
restriction OnlyOnce:
  "All #i #j. OnlyOnce()@#i & OnlyOnce()@#j ==> #i = #j"

restriction Equality:
  "All x y #i.
     Eq(x, y) @i
     ==> x = y"

restriction Inequality:
  "All x y #i.
     Neq(x, y) @i
     ==> not (x = y)"

restriction CertValidity:
  "All cert #i.
     ValidCert(cert) @i
     ==> (Ex #j.
             CreateOCert(cert)@#j
             & j < i)
         & not (Ex #k.
             OCertExpired(cert)@#k
             & k < i)"

restriction FreshAuth:
  "All A B n #i #j.
    SendAuth(A, B, n) @i
    & SendAuth(A, B, n) @j
    ==> #i = #j"

restriction FreshNonce:
```

```
  "All A B n #i #j.
    SendFreshNonce(A, B, n) @i
    & SendFreshNonce(A, B, n) @j
    ==> #i = #j"

restriction FreshKeys:
  "All key #i #j.
    EstablishKey(key) @i
    & EstablishKey(key) @j
    ==> #i = #j"

restriction one_ltk:
  "All A x y #i #j.
    PublicKeyRegistered(A, x) @i
    & PublicKeyRegistered(A, y) @j
    ==> #i = #j"

restriction one_role:
  "All A tid tid2 role role2 #i #j.
    Role(A, role, tid) @i
    & Role(A, role2, tid2) @j
    ==> role = role2"

restriction FreshDHShares:
  "All x #i #j.
  HonestDHExp(x) @i
  & HonestDHExp(x) @j
  ==> #i = #j"

restriction dh_shares_arent_nonces:
  "All ga a #i.
    HonestDH(ga, a) @i
    ==> not ((Ex A B #j. SendFreshNonce(A, B, ga) @j )
        | (Ex A B #j. SendAuth(A, B, ga) @j)
```

```
        | (Ex A B #j. Authentic(A, B, ga) @j))"

lemma DHsec [use_induction, reuse]:
  "All ga a #i #j.
    HonestDH(ga, a) @i
    & KU(ga) @j
    ==> not (Ex #k. KU(a)@k)"

//Cert Lemmas
lemma CertChain [reuse, use_induction]:
  "All cert #i #j.
    CreateOCert(cert) @i
    & OCertByUpdate(cert) @j
    ==> (Ex first #k.
           FirstOCert(first) @k
           & k < i
           & k < j)"

lemma OCertChain2 [use_induction]:
  "All cert #i.
    CreateOCert(cert) @i
    ==> (Ex #j. OCertByUpdate(cert) @j)
        | (Ex #j. FirstOCert(cert) @j)

//OCert Ordering
lemma CreateThenExpire [reuse, use_induction]:
  "All cert #i.
    OCertExpired(cert)@i
    ==> (Ex #j.
          CreateOCert(cert) @#j
          & j < i)"

//Entity authentication
lemma easy_auth [use_induction, reuse]:
```

```
  "All A B nc #i.
     Authentic(A, B, nc) @i
     ==> (Ex #j. SendAuth(B, A, nc)@j
            & j < i)"

lemma fresh_entity_authentication [use_induction, reuse]:
  "All A B nc #i.
     Authentic(A, B, nc)@i
     ==> (Ex #j. SendAuth(B, A, nc)@j
          & j < i
          & not (Ex #i2. SendAuth(B, A, nc)@i2
                               & (#i2 = #i)))"

//Post-compromise secrecy and secure key generation
lemma sessionKeySec [use_induction, reuse]:
  "All A B n1 n2 key #i #j.
     K(key)@i
     & EstablishSessionKey(A, B, n1, n2, key)@j
     ==> (Ex #k.
           RevealSessionKey(key)@#k)"

lemma maskingKeySec [use_induction, reuse]:
  "All A D n1 n2 key #i.
     EstablishMaskingKey(A, D, n1, n2, key) @i
     ==> not (Ex #j. KU(key) @j)"

//Install Replay Resistance
lemma secure_install [use_induction]:
  "All D O A SW iid nc3 nc4 #i.
     Installed(D, O, A, SW, nc3, nc4, iid) @ #i
     ==> Authentic(D, A, nc4) @i"

lemma secure_app_transfer [use_induction]:
  "All A D O SW nc1 nc2 theOCert #i.
```

```
    AppTransfer(O, A, D, SW, nc1, nc2, theOCert) @i
    ==> Authentic(O, A, nc1) @i"

lemma software_transfer_confidentiality [use_induction]:
  "All A D O SW nc1 nc2 theOCert #i #j.
    AppTransfer(O, A, D, SW, nc1, nc2, theOCert) @i
    & KU(SW) @j
    ==> Ex key #k. RevealSessionKey(key) @k & #k < #j"

lemma software_install_confidentiality [use_induction]:
  "All A D O SW nc1 nc2 iid #i #j.
    Installed(D, O, A, SW, nc1, nc2, iid) @ #i
    & KU(SW) @j
    ==> Ex key #k. RevealSessionKey(key) @k & #k < #j"

end
```

Appendix D

Tamarin Model for Authority-Centric Provisioning with Pre-Shared Keys

---

```
theory OperatorCentricPSK
begin

builtins: diffie-hellman, symmetric-encryption, signing, hashing
functions: mask/2, mac/2

//PKI
rule Register_pk:
  [ Fr(~ltkA) ]
--[ PublicKeyRegistered($A, pk(~ltkA)) ]->
  [ !Ltk($A, ~ltkA), !Pk($A, pk(~ltkA)), Out(pk(~ltkA)) ]

rule Register_pre_shared_key:
  [ Fr(~pskAB) ]
--[ PreSharedKeyEstablished($A, $B, ~pskAB) ]->
  [ !Psk($A, $B, ~pskAB) ]

rule create_O_cert:
let
  ocertnew = <$O, pkO, ~cid>
in
  [ Fr(~cid), !Pk($O, pkO) ]
--[ OnlyOnce(), CreateOCert(ocertnew), FirstOCert(ocertnew) ]->
  [ !OCert(ocertnew), Out(ocertnew) ]
```

```
rule update_O_cert:
let
  ocertold = <$O1, pkO1, ~cid1>
  ocertnew = <$O, pkO, ~cid>
in
  [ Fr(~cid), !Pk($O, pkO), !OCert(ocertold) ]
--[ ValidCert(ocertold), ReCert($O, $O1, ~cid, ~cid1)
  , OCertExpired(ocertold), OCertByUpdate(ocertnew)
  , CreateOCert(ocertnew) ]->
  [ !OCert(ocertnew), Out(ocertnew) ]

rule reveal_session_key:
  [ !SessionKey(seshk, nc1, nc2) ]
--[ RevealSessionKey(seshk) ]->
  [ Out(seshk) ]

//First Message (O->A)
rule O1:
let
  DHa = ~DHa
  nc1 = ~nc1
  pid = ~pid
  A = $A
  O = $O
  theOCert = <$O, pk(ltkO), cid>
  ga = 'g' ^ DHa
  m1 = <'message1', O, A, nc1, ga, theOCert>
  m1Sig = sign(h(m1), ltkO)
in
  [ !Ltk($O, ltkO), !OCert(theOCert), Fr(nc1), Fr(DHa), Fr(pid) ]
--[ O1(pid) , OUT_O1(pid, nc1, ga, m1, m1Sig)
  , Role(O, 'operator', pid), Neq(ga, 'g'), ValidCert(theOCert)
```

```
  , SendFreshNonce(O, A, nc1), HonestDHExp(DHa), HonestDH(ga, DHa)
  ]->
  [ Out(<m1, m1Sig>), St_O1(O, A, nc1, DHa, pid, theOCert) ]

//Second Message (A->O)
rule A1:
let
  b = ~b
  nc2 = ~nc2
  A = $A
  O = $O
  ga = 'g' ^ new_a
  gb = 'g' ^ b
  seshK = ga ^ b
  recOCert = <O, pk(ltkO), cid>
  m1 = <'message1', O, A, nc1, ga, recOCert>
  m1Sig = sign(h(m1), ltkO)
  m2 = <'message2', A, O, nc1, nc2, gb>
  m2Sig = sign(h(m2), ltkA)
in
  [ !Ltk(A, ltkA), !Ltk(O, ltkO), !OCert(recOCert), Fr(nc2)
  , Fr(b), Fr(pid), In(<m1, m1Sig>) ]
--[ A1(pid), IN_A1(pid, nc1, ga, m1, m1Sig)
  , Role(A, 'authority', pid), SendAuth(A, O, nc1)
  , SendFreshNonce(A, O, nc2), ValidCert(recOCert), HonestDHExp(b)
  , HonestDH(gb, b), Neq(ga, 'g'), Neq(gb, 'g'), Neq(seshK, 'g')
  , OUT_A1(pid, m2), EstablishSessionKey(O, A, nc1, nc2, seshK)
  , EstablishKey(seshK) ]->
  [ Out(<m2, m2Sig>), !SessionKey(seshK, nc1, nc2)
  , St_A1(A, O, nc1, nc2, ga, b, ga^b, recOCert, pid) ]

//Third Message (O->A)
rule O2:
let
```

```
  DHa = ~DHa
  nc1 = ~nc1
  pid = ~pid
  SWid = ~SWid
  O = $O
  A = $A
  D = $D
  theOCert = <O, pk(ltkO), cid>
  App = <'sw', SWid>
  AppHash = h(<App, 'swhash'>)
  gb = 'g' ^ b
  seshK = gb ^ ~DHa
  m2 = <'message2', A, O, nc1, nc2, gb>
  m2Sig = sign(h(m2), ltkA)
  m3 = <'message3', O, A, nc2, senc(<D, App, AppHash>, seshK)>
  m3Sig = sign(h(m3), ltkO)
in
  [ !Ltk($O, ltkO), !Ltk($A, ltkA), Fr(SWid)
  , St_O1(O, A, nc1, DHa, pid, theOCert)
  , In(<m2, m2Sig>), !OCert(theOCert) ]
--[ O2(pid), IN_O2(pid, nc2, m2), Role(O, 'operator', pid)
  , SendAuth(O, A, nc2), Authentic(O, A, nc1)
  , AppTransfer(O, A, D, App, nc1, nc2, theOCert)
  , Neq(gb, 'g' ^ DHa), Neq(gb, 'g'), Neq(seshK, 'g')
  , ValidCert(<$O, pk(ltkO), cid>), OUT_O2(pid, m3) ]->
  [ Out(<m3, m3Sig>) ]

rule A2:
let
  c = ~c
  pid = ~pid
  nc2 = ~nc2
  nc3 = ~nc3
  tid = ~tid
```

```
  A = $A
  D = $D
  O = $O
  SWHash = h(<SW, 'swhash'>)
  stOCert = <$O, pk(ltkO), cid>
  gc = 'g' ^ c
  m3 = <'message3', O, A, nc2, senc(<D, SW, SWHash>, seshk)>
  m3Sig = sign(h(m3), ltkO)
  m4 = <'message4', A, D, nc3, gc>
  m4MAC = mac(m4, pskAD)
in
  [ !Ltk(O, ltkO), St_A1(A, O, nc1, nc2, ga, b, seshk, stOCert, pid)
  , In(<m3, m3Sig>), Fr(c), Fr(tid), Fr(nc3), !Psk(A, D, pskAD) ]
--[ A2(pid), IN_A2(pid, m3), Role(A, 'authority', pid)
  , Authentic($A, $O, nc2), ValidCert(stOCert)
  , SWReceived($A, $O, $D, SW, tid, nc1, nc2, stOCert)
  , SendFreshNonce(A, D, nc3), HonestDHExp(c), HonestDH(gc, c)
  , Neq(gc, 'g'), Neq(gc, ga), Neq(gc, 'g' ^ b), OUT_A2(pid, m4) ]->
  [ St_A2(A, O, D, nc1, nc2, nc3, seshk, ga, b, c, stOCert, pid, SW)
  , Out(<m4, m4MAC>) ]

rule D1:
let
  DHd = ~DHd
  e = ~e
  nc4 = ~nc4
  pid = ~pid
  A = $A
  D = $D
  O = $O
  gc = 'g' ^ c
  gd = 'g' ^ DHd
  ge = 'g' ^ e
  seshK = gc ^ DHd
```

```
  m4 = <'message4', A, D, nc3, gc>
  m4MAC = mac(m4, pskAD)
  m5 = <'message5', D, A, nc3, nc4, gd, senc(<'dhexponents', ge>, seshK)>
  m5MAC = mac(m5, pskAD)
in
  [ In(<m4, m4MAC>), Fr(nc4), Fr(DHd), Fr(e), Fr(pid)
  , !Psk(A, D, pskAD) ]
--[ D1(pid), IN_D1(pid, nc3, m4), Role(D, 'device', pid)
  , SendAuth(D, A, nc3), SendFreshNonce(D, A, nc4)
  , EstablishSessionKey(A, D, nc3, nc4, seshK)
  , EstablishKey(seshK), HonestDHExp(DHd), HonestDH(gd, DHd)
  , HonestDHExp(e), HonestDH(ge, e), Neq(DHd, e), Neq(gc, 'g')
  , Neq(gd, 'g'), Neq(ge, 'g'), OUT_D1(pid, m5) ]->
  [ St_D1(D, A, nc3, nc4, DHd, e, seshK, pid)
  , !SessionKey(nc3, nc4, seshK), Out(<m5, m5MAC>) ]

rule A3:
let
  nc2 = ~nc2
  nc3 = ~nc3
  b = ~b
  c = ~c
  f = ~f
  pid = ~pid
  A = $A
  D = $D
  O = $O
  gd = 'g' ^ new_d
  ge = 'g' ^ new_e
  gf = 'g' ^ f
  seshKAD = gd ^ c
  maskingKey = ge ^ f
  MaskedSW = mask(SW, maskingKey)
  m5 = <'message5', D, A, nc3, nc4, gd,
```

```
                              senc(<'dhexponents', ge>, seshKAD)>
  m5MAC = mac(m5, pskAD)
  m6 = <'message6', A, D, nc3, nc4,
                              senc(<MaskedSW, h(SW), gf>, seshKAD)>
  m6MAC = mac(m6, pskAD)
in
  [ In(<m5, m5MAC>), Fr(f), !Psk(A, D, pskAD)
  , St_A2(A, O, D, nc1, nc2, nc3, seshKAO, ga, b, c,
                                            stOCert, pid, SW) ]
--[ A3(pid), IN_A3(pid, nc4, m5), Role(A, 'authority', pid)
  , SendAuth(A, D, nc4), Authentic(A, D, nc3)
  , SendAppInstall(O, A, D, nc1, nc2, nc3, nc4, SW, stOCert)
  , ValidCert(stOCert), HonestDHExp(f), HonestDH(gf, f)
  , Neq(gd, 'g'), Neq(ge, 'g'), Neq(gf, 'g'), Neq(seshKAD, 'g')
  , Neq(maskingKey, 'g')
  , EstablishMaskingKey(A, D, nc3, nc4, maskingKey)
  , EstablishKey(maskingKey), OUT_A3(pid, m6) ]->
  [ Out(<m6, m6MAC>), !MaskingKey(maskingKey, nc3, nc4) ]

rule D2:
let
  DHd = ~DHd
  e = ~e
  nc4 = ~nc4
  iid = ~iid
  A = $A
  D = $D
  O = $O
  gf = 'g' ^ new_f
  maskingKey = gf ^ e
  MaskedSW = mask(SW, maskingKey)
  m6 = <'message6', A, D, nc3, nc4,
                              senc(<MaskedSW, h(SW), gf>, seshK)>
  m6MAC = mac(m6, pskAD)
```

```
in
  [ St_D1(D, A, nc3, nc4, DHd, e, seshK, pid), Fr(iid)
  , In(<m6, m6MAC>), !Psk(A, D, pskAD) ]
--[ D2(pid), IN_D2(pid, m6), Role(D, 'device', pid)
  , Authentic(D, A, nc4), Installed(D, O, A, SW, nc3, nc4, iid)
  , Neq(gf, 'g'), Neq(gf, 'g'^DHd), Neq(gf, 'g'^e)
  , Neq(seshK, 'g')]->
  [   ]

//Restrictions
restriction OnlyOnce:
  "All #i #j. OnlyOnce()@#i & OnlyOnce()@#j ==> #i = #j"

restriction Equality:
  "All x y #i.
     Eq(x, y) @i
     ==> x = y"

restriction Inequality:
  "All x y #i.
     Neq(x, y) @i
     ==> not (x = y)"

restriction CertValidity:
  "All cert #i.
     ValidCert(cert) @i
     ==> (Ex #j.
             CreateOCert(cert)@#j
             & j < i)
         & not (Ex #k.
             OCertExpired(cert)@#k
             & k < i)"

restriction FreshAuth:
```

```
  "All A B n #i #j.
    SendAuth(A, B, n) @i
    & SendAuth(A, B, n) @j
    ==> #i = #j"

restriction FreshNonce:
  "All A B n #i #j.
    SendFreshNonce(A, B, n) @i
    & SendFreshNonce(A, B, n) @j
    ==> #i = #j"

restriction FreshKeys:
  "All key #i #j.
    EstablishKey(key) @i
    & EstablishKey(key) @j
    ==> #i = #j"

restriction one_ltk:
  "All A x y #i #j.
    PublicKeyRegistered(A, x) @i
    & PublicKeyRegistered(A, y) @j
    ==> #i = #j"

restriction one_psk:
  "All A B k1 k2 #i #j.
    PreSharedKeyEstablished(A, B, k1) @i
    & PreSharedKeyEstablished(A, B, k2) @j
    ==> #i = #j"

restriction one_psk_direction:
  "All A B k #i.
    PreSharedKeyEstablished(A, B, k) @i
    ==> not (Ex k2 #j. PreSharedKeyEstablished(B, A, k2) @j)"
```

```
restriction one_role:
  "All A tid tid2 role role2 #i #j.
    Role(A, role, tid) @i
    & Role(A, role2, tid2) @j
    ==> role = role2"

restriction FreshDHShares:
  "All x #i #j.
  HonestDHExp(x) @i
  & HonestDHExp(x) @j
  ==> #i = #j"

restriction dh_shares_arent_nonces:
  "All ga a #i.
    HonestDH(ga, a) @i
    ==> not ((Ex A B #j. SendFreshNonce(A, B, ga) @j )
          | (Ex A B #j. SendAuth(A, B, ga) @j)
          | (Ex A B #j. Authentic(A, B, ga) @j))"

//Source Lemmas
lemma DHsec [use_induction, reuse]:
  "All ga a #i #j.
    HonestDH(ga, a) @i
    & KU(ga) @j
    ==> not (Ex #k. KU(a)@k)"

//Cert Lemmas
lemma CertChain [reuse, use_induction]:
  "All cert #i #j.
    CreateOCert(cert) @i
    & OCertByUpdate(cert) @j
    ==> (Ex first #k.
            FirstOCert(first) @k
            & k < i
```

```
          & k < j)"

lemma OCertChain2 [use_induction]:
  "All cert #i.
    CreateOCert(cert) @i
    ==> (Ex #j. OCertByUpdate(cert) @j)
        | (Ex #j. FirstOCert(cert) @j)"



//OCert Ordering
lemma CreateThenExpire [reuse, use_induction]:
  "All cert #i.
    OCertExpired(cert)@i
    ==> (Ex #j.
         CreateOCert(cert) @#j
        & j < i)"

//Entity authentication
lemma easy_auth [use_induction, reuse]:
  "All A B nc #i.
    Authentic(A, B, nc) @i
    ==> (Ex #j. SendAuth(B, A, nc)@j
          & j < i)"

lemma fresh_entity_authentication [use_induction, reuse]:
  "All A B nc #i.
    Authentic(A, B, nc)@i
    ==> (Ex #j. SendAuth(B, A, nc)@j
        & j < i
        & not (Ex #i2. SendAuth(B, A, nc)@i2
                      & (#i2 = #i)))"

//Post-compromise secrecy and secure key generation
lemma sessionKeySec [use_induction, reuse]:
```

```
  "All A B n1 n2 key #i #j.
    K(key)@i
    & EstablishSessionKey(A, B, n1, n2, key)@j
    ==> (Ex #k.
          RevealSessionKey(key)@#k)"

lemma maskingKeySec [use_induction, reuse]:
  "All A D n1 n2 key #i.
    EstablishMaskingKey(A, D, n1, n2, key) @i
    ==> not (Ex #j. KU(key) @j)"

//Install Replay Resistance
lemma secure_install [use_induction]:
  "All D O A SW iid nc3 nc4 #i.
    Installed(D, O, A, SW, nc3, nc4, iid) @ #i
    ==> Authentic(D, A, nc4) @i"

lemma secure_app_transfer [use_induction]:
  "All A D O SW nc1 nc2 theOCert #i.
    AppTransfer(O, A, D, SW, nc1, nc2, theOCert) @i
    ==> Authentic(O, A, nc1) @i"

lemma software_transfer_confidentiality [use_induction, reuse]:
  "All A D O SW nc1 nc2 theOCert #i #j.
    AppTransfer(O, A, D, SW, nc1, nc2, theOCert) @i
    & KU(SW) @j
    ==> Ex key #k. RevealSessionKey(key) @k & #k < #j"

lemma software_install_confidentiality [use_induction]:
  "All A D O SW nc1 nc2 iid #i #j.
    Installed(D, O, A, SW, nc1, nc2, iid) @ #i
    & KU(SW) @j
    ==> Ex key #k. RevealSessionKey(key) @k & #k < #j"
```

```
end
```

Appendix E

Secure-Execution Processor Test Application Code

```
CONSTANT LED_PORT, 02
CONSTANT UART_TX_STATUS, 00
CONSTANT UART_TX_RESET, 03
CONSTANT UART_RX_READ, 01
CONSTANT UART_TX_DATA_IN, 01

start:
  LOAD s0, FF
  LOAD s0, FF
  OUTPUT s0, UART_TX_RESET
  LOAD s0, 00
  LOAD s0, 00
  OUTPUT s0, UART_TX_RESET
  LOAD s0, s0
  LOAD s0, s0
  LOAD s1, "H"
  CALL waitForUARTFree
  OUTPUT s1, UART_TX_DATA_IN
  LOAD s1, "e"
  CALL waitForUARTFree
  OUTPUT s1, UART_TX_DATA_IN
  LOAD s1, "l"
  CALL waitForUARTFree
  OUTPUT s1, UART_TX_DATA_IN
  LOAD s1, "l"
  CALL waitForUARTFree
  OUTPUT s1, UART_TX_DATA_IN
  LOAD s1, "o"
  CALL waitForUARTFree
  OUTPUT s1, UART_TX_DATA_IN
  LOAD s1, " "
  CALL waitForUARTFree
  OUTPUT s1, UART_TX_DATA_IN
  LOAD s1, "W"
  CALL waitForUARTFree
  OUTPUT s1, UART_TX_DATA_IN
  LOAD s1, "o"
  CALL waitForUARTFree
  OUTPUT s1, UART_TX_DATA_IN
  LOAD s1, "r"
  CALL waitForUARTFree
  OUTPUT s1, UART_TX_DATA_IN
  LOAD s1, "l"
  CALL waitForUARTFree
  OUTPUT s1, UART_TX_DATA_IN
  LOAD s1, "d"
  CALL waitForUARTFree
  OUTPUT s1, UART_TX_DATA_IN
```

```
  LOAD s1, "!"
  CALL waitForUARTFree
  OUTPUT s1, UART_TX_DATA_IN
  LOAD s1, " "
  CALL waitForUARTFree
  OUTPUT s1, UART_TX_DATA_IN
  LOAD s1, 0D
  CALL waitForUARTFree
  OUTPUT s1, UART_TX_DATA_IN
  JUMP ledStart

waitForUARTFree:
  INPUT s0, UART_TX_STATUS
  AND s0, 04
  JUMP NZ waitForUARTFree
  RETURN
```

```
ledStart:
        LOAD s0, 00
loop:   OUTPUT s0, LED_PORT
        LOAD s2, FF
x:      LOAD s1, FF
y:      SUB s1, 01
        JUMP NZ y
        SUB s2, 01
        JUMP NZ x
        ADD s0, 01
        JUMP C reset
back:   JUMP loop
reset:  LOAD s0, 00
        JUMP back
```

Bibliography

[1] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity. In V. Atluri, C. A. Meadows, and A. Juels, editors, *Proceedings of the 12th ACM Conference on Computer and Communications Security, CCS 2005, Alexandria, VA, USA, November 7-11, 2005*, pages 340–353. ACM, 2005.

[2] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Trans. Inf. Syst. Secur.*, 13(1):4:1–4:40, 2009.

[3] Adafruit Industries. Bus pirate basic probe set, 2019.

[4] A. C. Aldaya, C. P. García, L. M. A. Tapia, and B. B. Brumley. Cache-timing attacks on RSA key generation. *IACR Cryptology ePrint Archive*, 2018:367, 2018.

[5] B. Anckaert, M. Madou, B. D. Sutter, B. D. Bus, K. D. Bosschere, and B. Preneel. Program obfuscation: a quantitative approach. In G. Karjoth and K. Stølen, editors, *Proceedings of the 3th ACM Workshop on Quality of Protection, QoP 2007, Alexandria, VA, USA, October 29, 2007*, pages 15–20. ACM, 2007.

[6] J. H. Anderson. A PUF design for secure FPGA-based embedded systems. In *Proceedings of the 15th Asia South Pacific Design Automation Conference, ASP-DAC 2010, Taipei, Taiwan, January 18-21, 2010*, pages 1–6. IEEE, 2010.

[7] A. Appel. Deobfuscation is in np. *Princeton University, Aug*, 21:2, 2002.

[8] D. Arora, S. Ravi, A. Raghunathan, and N. K. Jha. Secure embedded processing through hardware-assisted run-time monitoring. In *2005 Design, Automation and Test in Europe Conference and Exposition (DATE 2005), 7-11 March 2005, Munich, Germany*, pages 178–183. IEEE Computer Society, 2005.

[9] K. Ashton. That 'Internet of Things' Thing. *RFID journal*, 22(7):97–114, 2009.

[10] M. J. Atallah, E. D. Bryant, J. T. Korb, and J. R. Rice. Binding software to specific native hardware in a VM environment: The PUF challenge and opportunity. In *Proceedings of the 1st ACM Workshop on Virtual Machine Security*, VMSec '08, pages 45–48, New York, NY, USA, 2008. ACM.

[11] L. Atzori, A. Iera, and G. Morabito. The Internet of Things: A Survey. *Computer Networks*, 54(15):2787–2805, 2010.

[12] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. P. Vadhan, and K. Yang. On the (im)possibility of obfuscating programs. In J. Kilian, editor, *Advances in Cryptology - CRYPTO 2001, 21st Annual International Cryptology Conference, Santa Barbara, California, USA, August 19-23, 2001, Proceedings*, volume 2139 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2001.

[13] D. A. Basin, C. J. F. Cremers, and S. Meier. Provably repairing the ISO/IEC 9798 standard for entity authentication. In P. Degano and J. D. Guttman, editors, *Principles of Security and Trust - First International Conference, POST 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012, Proceedings*, volume 7215 of *Lecture Notes in Computer Science*, pages 129–148. Springer, 2012.

[14] D. Bauder. An anti-counterfeiting concept for currency systems. *Sandia National Labs, Albuquerque, NM, Tech. Rep. PTK-11990*, 1983.

[15] G. T. Becker. The gap between promise and reality: On the insecurity of XOR arbiter pufs. In T. Güneysu and H. Handschuh, editors, *Cryptographic Hardware and Embedded Systems - CHES 2015 - 17th International Workshop, Saint-Malo, France, September 13-16, 2015, Proceedings*, volume 9293 of *Lecture Notes in Computer Science*, pages 535–555. Springer, 2015.

[16] J. Borghoff, A. Canteaut, T. Güneysu, E. B. Kavun, M. Knezevic, L. R. Knudsen, G. Leander, V. Nikov, C. Paar, C. Rechberger, P. Rombouts, S. S. Thomsen, and T. Yalçin. PRINCE - A low-latency block cipher for pervasive computing applications - extended abstract. In X. Wang and K. Sako, editors, *Advances in Cryptology - ASIACRYPT 2012 - 18th International Conference on the Theory and Application of Cryptology and Information Security, Beijing, China, December 2-6, 2012.*

Proceedings, volume 7658 of *Lecture Notes in Computer Science*, pages 208–225. Springer, 2012.

[17] L. Bossuet and B. Colombier. Comments on "a PUF-FSM binding scheme for FPGA IP protection and pay-per-device licensing". *IEEE Trans. Information Forensics and Security*, 11(11):2624–2625, 2016.

[18] J. Brodkin. Mobile internet devices will outnumber humans this year, cisco predicts, 2012.

[19] E. Buchanan, R. Roemer, S. Savage, and H. Shacham. Return-oriented programming: Exploitation without code injection. *Black Hat*, 8, 2008.

[20] M. Cheminod, I. C. Bertolotti, L. Durante, R. Sisto, and A. Valenzano. Experimental comparison of automatic tools for the formal analysis of cryptographic protocols. In *2007 International Conference on Dependability of Computer Systems (DepCoS-RELCOMEX 2007), June 14-16, 2007, Szklarska Poreba, Poland*, pages 153–160. IEEE Computer Society, 2007.

[21] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai. SGXPECTRE attacks: Leaking enclave secrets via speculative execution. *CoRR*, abs/1802.09085, 2018.

[22] N. Christoulakis, G. Christou, E. Athanasopoulos, and S. Ioannidis. HCFI: hardware-enforced control-flow integrity. In E. Bertino, R. Sandhu, and A. Pretschner, editors, *Proceedings of the Sixth ACM on Conference on Data and Application Security and Privacy, CODASPY 2016, New Orleans, LA, USA, March 9-11, 2016*, pages 38–49. ACM, 2016.

[23] C. Cifuentes and K. J. Gough. Decompilation of binary programs. *Softw., Pract. Exper.*, 25(7):811–829, 1995.

[24] C. S. Collberg and C. D. Thomborson. Watermarking, tamper-proofing, and obfuscation-tools for software protection. *IEEE Trans. Software Eng.*, 28(8):735–746, 2002.

[25] L. Columbus. 2017 Roundup of internet of things forecasts, 2017.

[26] Corero. Mirai botnet DDoS attack type, 2018.

[27] C. J. F. Cremers. The scyther tool: Verification, falsification, and analysis of security protocols. In A. Gupta and S. Malik, editors, *Computer Aided Verification,*

20th International Conference, CAV 2008, Princeton, NJ, USA, July 7-14, 2008, Proceedings, volume 5123 of *Lecture Notes in Computer Science*, pages 414–418. Springer, 2008.

[28] C. J. F. Cremers, P. Lafourcade, and P. Nadeau. Comparing state spaces in automatic security protocol analysis. In V. Cortier, C. Kirchner, M. Okada, and H. Sakurada, editors, *Formal to Practical Security - Papers Issued from the 2005-2008 French-Japanese Collaboration*, volume 5458 of *Lecture Notes in Computer Science*, pages 70–94. Springer, 2009.

[29] W. Dai, T. P. Parker, H. Jin, and S. Xu. Enhancing data trustworthiness via assured digital signing. *IEEE Trans. Dependable Sec. Comput.*, 9(6):838–851, 2012.

[30] L. Davi, M. Hanreich, D. Paul, A. Sadeghi, P. Koeberl, D. Sullivan, O. Arias, and Y. Jin. HAFIX: hardware-assisted flow integrity extension. In *Proceedings of the 52nd Annual Design Automation Conference, San Francisco, CA, USA, June 7-11, 2015*, pages 74:1–74:6. ACM, 2015.

[31] R. de Clercq, J. Götzfried, D. Übler, P. Maene, and I. Verbauwhede. SOFIA: software and control flow integrity architecture. *Computers & Security*, 68:16–35, 2017.

[32] R. de Clercq, R. D. Keulenaer, B. Coppens, B. Yang, P. Maene, K. D. Bosschere, B. Preneel, B. D. Sutter, and I. Verbauwhede. SOFIA: software and control flow integrity architecture. In L. Fanucci and J. Teich, editors, *2016 Design, Automation & Test in Europe Conference & Exhibition, DATE 2016, Dresden, Germany, March 14-18, 2016*, pages 1172–1177. IEEE, 2016.

[33] R. de Clercq and I. Verbauwhede. A survey of hardware-based control flow integrity (CFI). *CoRR*, abs/1706.07257, 2017.

[34] Department for Business Innovation and Skills. Smart Cities: Background Paper. Technical report, HMG, October 2013.

[35] D. Dolev and A. C. Yao. On the security of public key protocols. *IEEE Trans. Information Theory*, 29(2):198–207, 1983.

[36] European Commission. Smart Cities, 2015.

[37] W. Fumy. Key management techniques. In B. Preneel and V. Rijmen, editors, *State of the Art in Applied Cryptography, Course on Computer Security and Industrial*

Cryptography, Leuven, Belgium, June 3-6, 1997. Revised Lectures, volume 1528 of *Lecture Notes in Computer Science*, pages 142–162. Springer, 1997.

[38] F. Ganji, S. Tajik, F. Fäßler, and J. Seifert. Strong machine learning attack against pufs with no mathematical model. In B. Gierlichs and A. Y. Poschmann, editors, *Cryptographic Hardware and Embedded Systems - CHES 2016 - 18th International Conference, Santa Barbara, CA, USA, August 17-19, 2016, Proceedings*, volume 9813 of *Lecture Notes in Computer Science*, pages 391–411. Springer, 2016.

[39] B. Gassend, D. E. Clarke, M. van Dijk, and S. Devadas. Silicon physical random functions. In V. Atluri, editor, *Proceedings of the 9th ACM Conference on Computer and Communications Security, CCS 2002, Washington, DC, USA, November 18-22, 2002*, pages 148–160. ACM, 2002.

[40] Genesis Toys. My friend cayla, 2014.

[41] D. Genkin, A. Shamir, and E. Tromer. Acoustic cryptanalysis. *J. Cryptology*, 30(2):392–443, 2017.

[42] GlobalPlatform. Global Platform Card Specification. Version 2.3, GlobalPlatform, October 2015.

[43] GlobalPlatform. Global platform overview, 2018.

[44] M. A. Gora, A. Maiti, and P. Schaumont. A flexible design flow for software IP binding in commodity FPGA. In *IEEE Fourth International Symposium on Industrial Embedded Systems, SIES 2009, Ecole Polytechnique Federale de Lausanne, Switzerland, July 8-10, 2009*, pages 211–218. IEEE, 2009.

[45] J. Guajardo, S. S. Kumar, G. J. Schrijen, and P. Tuyls. FPGA intrinsic PUFs and their use for IP protection. In P. Paillier and I. Verbauwhede, editors, *Cryptographic Hardware and Embedded Systems - CHES 2007, 9th International Workshop, Vienna, Austria, September 10-13, 2007, Proceedings*, volume 4727 of *Lecture Notes in Computer Science*, pages 63–80. Springer, 2007.

[46] J. Guajardo, S. S. Kumar, G. J. Schrijen, and P. Tuyls. Brand and IP protection with physical unclonable functions. In *International Symposium on Circuits and Systems (ISCAS 2008), 18-21 May 2008, Sheraton Seattle Hotel, Seattle, Washington, USA*, pages 3186–3189. IEEE, 2008.

[47] S. Haria, M. D. Hill, and M. M. Swift. Devirtualizing memory in heterogeneous systems. In X. Shen, J. Tuck, R. Bianchini, and V. Sarkar, editors, *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2018, Williamsburg, VA, USA, March 24-28, 2018*, pages 637–650. ACM, 2018.

[48] C. Helfmeier, C. Boit, D. Nedospasov, and J. Seifert. Cloning physically unclonable functions. In *2013 IEEE International Symposium on Hardware-Oriented Security and Trust, HOST 2013, Austin, TX, USA, June 2-3, 2013*, pages 1–6. IEEE Computer Society, 2013.

[49] B. Holdsworth and C. Woods. *Digital Logic Design*. Newnes, Linacre House, Oxford, UK, 4 edition, 11 2002.

[50] IBM. Connected cars with IBM Watson IoT, 2018.

[51] IEEE. Readings on Smart Cities, 2017.

[52] INCIBE. Introduction to embedded systems, 2018.

[53] ISO/IEC. ISO/IEC 15693:2006: Identification cards – Contactless integrated circuit cards – Vicinity cards. Standard, International Organisation for Standardisation, Geneva, Switzerland, 2006.

[54] ISO/IEC. ISO/IEC 7501:2008: Identification cards – Machine readable travel documents. Standard, International Organisation for Standardisation, Geneva, Switzerland, 2008.

[55] ISO/IEC. ISO/IEC 9798:2010: Information technology – Entity authentication. Standard, International Organisation for Standardisation, Geneva, Switzerland, 2010.

[56] ISO/IEC. ISO/IEC 14443:2011: Identification cards – Contactless integrated circuit cards – Proximity cards. Standard, International Organisation for Standardisation, Geneva, Switzerland, 2011.

[57] ISO/IEC. ISO/IEC 7816:2013: Identification cards – Integrated circuit cards. Standard, International Organisation for Standardisation, Geneva, Switzerland, 2013.

[58] Joel Clark, MWR Labs. Don't Try This at Home: Decapping ICs with Boiling Acid, 2016.

[59] S. Katzenbeisser, Ü. Koçabas, V. Rozic, A. Sadeghi, I. Verbauwhede, and C. Wachsmann. PUFs: Myth, fact or busted? A security evaluation of physically unclonable functions (PUFs) cast in silicon. In E. Prouff and P. Schaumont, editors, *Cryptographic Hardware and Embedded Systems - CHES 2012 - 14th International Workshop, Leuven, Belgium, September 9-12, 2012. Proceedings*, volume 7428 of *Lecture Notes in Computer Science*, pages 283–301. Springer, 2012.

[60] T. Kean. Cryptographic rights management of FPGA intellectual property cores. In Schlag and Trimberger [106], pages 113–118.

[61] J. Kinder, F. Zuleger, and H. Veith. An abstract interpretation-based framework for control flow reconstruction from binaries. In N. D. Jones and M. Müller-Olm, editors, *Verification, Model Checking, and Abstract Interpretation, 10th International Conference, VMCAI 2009, Savannah, GA, USA, January 18-20, 2009. Proceedings*, volume 5403 of *Lecture Notes in Computer Science*, pages 214–228. Springer, 2009.

[62] V. Kiriansky, D. Bruening, and S. P. Amarasinghe. Secure Execution via Program Shepherding. In D. Boneh, editor, *Proceedings of the 11th USENIX Security Symposium, San Francisco, CA, USA, August 5-9, 2002*, pages 191–206. USENIX, 2002.

[63] S. Kleber, F. Unterstein, M. Matousek, F. Kargl, F. Slomka, and M. Hiller. Secure execution architecture based on PUF-driven instruction level code encryption. *IACR Cryptology ePrint Archive*, 2015:651, 2015.

[64] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre attacks: Exploiting speculative execution. *ArXiv e-prints*, Jan. 2018.

[65] P. C. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In M. J. Wiener, editor, *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer, 1999.

[66] P. Koopman. Embedded system security. *IEEE Computer*, 37(7):95–97, 2004.

[67] R. Krasinski and M. Rosner. Method for binding a software data domain to specific hardware, May 2003.

[68] S. S. Kumar, J. Guajardo, R. Maes, G. J. Schrijen, and P. Tuyls. The butterfly PUF: protecting IP on every FPGA. In M. Tehranipoor and J. Plusquellic, editors,

IEEE International Workshop on Hardware-Oriented Security and Trust, HOST 2008, Anaheim, CA, USA, June 9, 2008. Proceedings, pages 67–70. IEEE Computer Society, 2008.

[69] R. Langner. Stuxnet: Dissecting a cyberwarfare weapon. *IEEE Security & Privacy*, 9(3):49–51, 2011.

[70] J. W. Lee, D. Lim, B. Gassend, G. E. Suh, M. van Dijk, and S. Devadas. A technique to build a secret key in integrated circuits for identification and authentication applications. In *2004 Symposium on VLSI Circuits. Digest of Technical Papers (IEEE Cat. No.04CH37525)*, pages 176–179. IEEE, 2004.

[71] R. P. Lee, K. Markantonakis, and R. N. Akram. Binding hardware and software to prevent firmware modification and device counterfeiting. In J. Zhou and J. López, editors, *Proceedings of the 2nd ACM International Workshop on Cyber-Physical System Security, CPSS@AsiaCCS, Xi'an, China, May 30, 2016*, pages 70–81. ACM, 2016.

[72] R. P. Lee, K. Markantonakis, and R. N. Akram. Provisioning software with hardware-software binding. In *Proceedings of the 12th International Conference on Availability, Reliability and Security, Reggio Calabria, Italy, August 29 - September 01, 2017*, pages 49:1–49:9. ACM, 2017.

[73] R. P. Lee, K. Markantonakis, and R. N. Akram. Ensuring secure application execution and platform-specific execution in embedded devices. *ACM Transactions on Embedded Computing Systems*, 18(3):26:1–26:21, Apr. 2019.

[74] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. Meltdown. *ArXiv e-prints*, Jan. 2018.

[75] W. Liu, Z. Zhang, M. Li, and Z. Liu. A trustworthy key generation prototype based on DDR3 PUF for wireless sensor networks. *Sensors*, 14(7):11542–11556, 2014.

[76] G. Lowe. Casper: A compiler for the analysis of security protocols. *Journal of Computer Security*, 6(1-2):53–84, 1998.

[77] P. Maene. BlockCiphers, 2017.

[78] P. Maene and I. Verbauwhede. Single-cycle implementations of block ciphers. In T. Güneysu, G. Leander, and A. Moradi, editors, *Lightweight Cryptography for Security and Privacy - 4th International Workshop, LightSec 2015, Bochum, Germany,*

September 10-11, 2015, Revised Selected Papers, volume 9542 of *Lecture Notes in Computer Science*, pages 131–147. Springer, 2015.

[79] R. Maes. *Physically Unclonable Functions - Constructions, Properties and Applications*. Springer, 2013.

[80] R. Maes, P. Tuyls, and I. Verbauwhede. Intrinsic PUFs from flip-flops on reconfigurable devices. In *Proceedings of the 3rd Benelux Workshop on Information and System Security, WISSec 2008, Eindhoven, The Netherlands, November 13-14, 2008*, 2008.

[81] R. Maes and I. Verbauwhede. Physically unclonable functions: A study on the state of the art and future research directions. In A. Sadeghi and D. Naccache, editors, *Towards Hardware-Intrinsic Security - Foundations and Practice*, Information Security and Cryptography, pages 3–37. Springer, 2010.

[82] A. Maiti and P. Schaumont. Improved ring oscillator PUF: an FPGA-friendly secure primitive. *Journal of Cryptology*, 24(2):375–397, 2011.

[83] MAOSCO Ltd. Securing Smart Meters with MULTOS - Technical Overview, 2014.

[84] MAOSCO Ltd. Guide to Generating Application Load Units, 2015.

[85] MAOSCO Ltd. MULTOS explained, 2017.

[86] K. Markantonakis and K. Mayes. An overview of the GlobalPlatform smart card specification. *Information Security Technical Report*, 8(1):17 – 29, 2003.

[87] P. Marwedel. *Embedded System Design*. Kluwer, 2003.

[88] P. Marwedel. *Embedded System Design - Embedded Systems Foundations of Cyber-Physical Systems, Second Edition*. Embedded Systems. Springer, 2011.

[89] S. Mauw and S. Piramuthu. A PUF-based authentication protocol to address ticket-switching of rfid-tagged items. In A. Jøsang, P. Samarati, and M. Petrocchi, editors, *Security and Trust Management - 8th International Workshop, STM 2012, Pisa, Italy, September 13-14, 2012, Revised Selected Papers*, volume 7783 of *Lecture Notes in Computer Science*, pages 209–224. Springer, 2012.

[90] K. Mayes. An Introduction to Smart Cards. In K. Mayes and K. Markantonakis, editors, *Smart Cards, Tokens, Security and Applications*, chapter 1, pages 1 – 30. Springer Nature, Gewerbestrasse 11, 6330 Cham, Switzerland, 2017.

[91] S. Meier, B. Schmidt, C. Cremers, and D. A. Basin. The TAMARIN prover for the symbolic analysis of security protocols. In N. Sharygina and H. Veith, editors, *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, volume 8044 of *Lecture Notes in Computer Science*, pages 696–701. Springer, 2013.

[92] Microsoft Inc. Internet of Things (IoT) healthcare solutions, 2018.

[93] T. Morris. Trusted Platform Module. In H. C. A. van Tilborg and S. Jajodia, editors, *Encyclopedia of Cryptography and Security, 2nd Ed.*, pages 1332–1335. Springer, 2011.

[94] K. Munro. Making children's toys swear, 2015.

[95] L. H. Newman. The worst hacks of 2017, 2017.

[96] P. Oltermann. German parents told to destroy doll that can spy on children, 2017.

[97] Z. S. Paral and S. Devadas. Reliable and efficient PUF-based key generation using pattern matching. In *HOST 2011, Proceedings of the 2011 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST), 5-6 June 2011, San Diego, California, USA*, pages 128–133. IEEE, 2011.

[98] R. Patel, B. Borisaniya, A. Patel, D. R. Patel, M. Rajarajan, and A. Zisman. Comparative analysis of formal model checking tools for security protocol verification. In N. Meghanathan, S. Boumerdassi, N. Chaki, and D. Nagamalai, editors, *Recent Trends in Network Security and Applications - Third International Conference, CNSA 2010, Chennai, India, July 23-25, 2010. Proceedings*, volume 89 of *Communications in Computer and Information Science*, pages 152–163. Springer, 2010.

[99] Postscapes Labs. Internet of Things market size, 2015.

[100] S. Rasoolzadeh and H. Raddum. Cryptanalysis of PRINCE with minimal data. In D. Pointcheval, A. Nitaj, and T. Rachidi, editors, *Progress in Cryptology - AFRICACRYPT 2016 - 8th International Conference on Cryptology in Africa, Fes, Morocco, April 13-15, 2016, Proceedings*, volume 9646 of *Lecture Notes in Computer Science*, pages 109–126. Springer, 2016.

[101] S. Rasoolzadeh and H. Raddum. Faster key recovery attack on round-reduced PRINCE. In A. Bogdanov, editor, *Lightweight Cryptography for Security and Pri-*

vacy - 5th International Workshop, LightSec 2016, Aksaray, Turkey, September 21-22, 2016, Revised Selected Papers, volume 10098 of *Lecture Notes in Computer Science*, pages 3–17. Springer, 2016.

[102] N. Routley. Visualizing the trillion-fold increase in computing power, 2017.

[103] R. Rutt. Energy saving tips: Can a smart meter really cut the cost of my bills?, 2017.

[104] M. Safkhani, N. Bagheri, and M. Naderi. Security analysis of a PUF based RFID authentication protocol. *IACR Cryptology ePrint Archive*, 2011:704, 2011.

[105] M. Satyanarayanan. Pervasive computing: vision and challenges. *IEEE Personal Commun.*, 8(4):10–17, 2001.

[106] M. D. F. Schlag and S. Trimberger, editors. *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays, FPGA 2002, Monterey, CA, USA, February 24-26, 2002*. ACM, 2002.

[107] D. C. Schmidt. Middleware for real-time and embedded systems. *Commun. ACM*, 45(6):43–48, 2002.

[108] H. Schmit and V. Chandra. FPGA switch block layout and evaluation. In Schlag and Trimberger [106], pages 11–18.

[109] C. Shepherd, G. Arfaoui, I. Gurulian, R. P. Lee, K. Markantonakis, R. N. Akram, D. Sauveron, and E. Conchon. Secure and trusted execution: Past, present, and future - A critical review in the context of the internet of things and cyber-physical systems. In *2016 IEEE Trustcom/BigDataSE/ISPA, Tianjin, China, August 23-26, 2016*, pages 168–177. IEEE, 2016.

[110] E. Simpson and P. Schaumont. Offline hardware/software authentication for reconfigurable platforms. In L. Goubin and M. Matsui, editors, *Cryptographic Hardware and Embedded Systems - CHES 2006, 8th International Workshop, Yokohama, Japan, October 10-13, 2006, Proceedings*, volume 4249 of *Lecture Notes in Computer Science*, pages 311–323. Springer, 2006.

[111] Statista. Number of Internet of Things (IoT) devices connected worldwide in 2017 and 2018, by type (in millions), 2018.

[112] G. E. Suh and S. Devadas. Physical unclonable functions for device authentication and secret key generation. In *Proceedings of the 44th Design Automation Conference, DAC 2007, San Diego, CA, USA, June 4-8, 2007*, pages 9–14. IEEE, 2007.

[113] D. Sullivan, O. Arias, L. Davi, P. Larsen, A. Sadeghi, and Y. Jin. Strategy without tactics: policy-agnostic hardware-enhanced control-flow integrity. In *Proceedings of the 53rd Annual Design Automation Conference, DAC 2016, Austin, TX, USA, June 5-9, 2016*, pages 163:1–163:6. ACM, 2016.

[114] Symantec Corporation. Internet security technical report (ISTR). Technical report, 03 2018.

[115] Symantec Security Response. Mirai: what you need to know about the botnet behind recent major DDoS attacks, 2016.

[116] S. Tajik, E. Dietz, S. Frohmann, J. Seifert, D. Nedospasov, C. Helfmeier, C. Boit, and H. Dittrich. Physical characterization of arbiter pufs. In L. Batina and M. Robshaw, editors, *Cryptographic Hardware and Embedded Systems - CHES 2014 - 16th International Workshop, Busan, South Korea, September 23-26, 2014. Proceedings*, volume 8731 of *Lecture Notes in Computer Science*, pages 493–509. Springer, 2014.

[117] H. Theiling. Extracting safe and precise control flow from binaries. In *7th International Workshop on Real-Time Computing and Applications Symposium (RTCSA 2000), 12-14 December 2000, Cheju Island, South Korea*, pages 23–30. IEEE Computer Society, 2000.

[118] T. Toyofuku, T. Tabata, and K. Sakurai. Program obfuscation scheme using random numbers to complicate control flow. In T. Enokido, L. Yan, B. Xiao, D. Kim, Y. Dai, and L. T. Yang, editors, *Embedded and Ubiquitous Computing - EUC 2005 Workshops, EUC 2005 Workshops: UISW, NCUS, SecUbiq, USN, and TAUES, Nagasaki, Japan, December 6-9, 2005, Proceedings*, volume 3823 of *Lecture Notes in Computer Science*, pages 916–925. Springer, 2005.

[119] Trusted Computing Group. Part 1 Design Principles. In *TPM Main Specification*. TCG, 2011.

[120] J. Turley. Embedded processors by the numbers, 1999.

[121] V. van der Leest and P. Tuyls. Anti-counterfeiting with hardware intrinsic security. In E. Macii, editor, *Design, Automation and Test in Europe, DATE 13, Grenoble,*

France, March 18-22, 2013, pages 1137–1142. EDA Consortium San Jose, CA, USA / ACM DL, 2013.

[122] M. Weiser. Ubiquitous computing, 1996.

[123] Xilinx, Inc. PicoBlaze 8-bit Microcontroller, 2011.

[124] Xilinx, Inc. Spartan-6 FPGA SP601 Evaluation Kit, 2018.

[125] J. Zhang, Y. Lin, Y. Lyu, and G. Qu. A PUF-FSM binding scheme for FPGA IP protection and pay-per-device licensing. *IEEE Trans. Information Forensics and Security*, 10(6):1137–1150, 2015.