Secure Application Execution in Mobile Devices

Mehari G. Msgna, Houda Ferradi, Raja Naeem Akram, and Konstantinos Markantonakis

Abstract Smart phones have rapidly become hand-held mobile devices capable of sustaining multiple applications. Some of these applications allow access to services including healthcare, financial and online social networks and are becoming common in the smart phone environment. From a security and privacy point of view, this seismic shift is creating new challenges, as the smart phone environment is becoming a suitable platform for security- and privacy-sensitive applications. The need for a strong security architecture for this environment is becoming paramount, especially from the point of view of Secure Application Execution (SAE). In this chapter, we explore SAE for applications on smart phone platforms, to ensure application execution is undertaken as expected by the application provider. Most of the proposed SAE proposals are based on having a secure and trusted embedded chip on the smart phone. Examples include the GlobalPlatform Trusted Execution Environment, M-Shield and Mobile Trusted Module. These additional hardware components, referred to as secure and trusted devices, provide a secure environment in which the applications can execute security-critical code and/or store data. These secure and trusted devices can become the target of malicious entities; therefore, we need to have a strong framework to validate and secure the code execution on such devices. This chapter discusses how we can provide an assurance that applications executing on such devices are secure by validating the secure and trusted hardware.

Key words: Smart Phone, Apple iOS, Android, Mobile Trusted Manager, GlobalPlatform Trusted Execution Environment, Secure Application Execution.

M. G. Msgna, R. N. Akram & K. Markantonakis, Information Security Group, Smart Card Centre, Royal Holloway, University of London, Egham, UK.
e-mail: `\{mehari.msgna.2011,r.n,akram,k.markantonakis\}@rhul.ac.uk`
e-mail: `m.g.msgna@insa.gov.et`
H. Ferradi, 'Ecole normale supérieure' e-mail: `houda.ferradi@ens.fr`

1 Introduction

Mobile phones have changed the way we communicate and stay in touch with friends and family. This revolution, including ubiquitous voice communication and Short Messaging Services (SMS), was pivotal in the early adoption of mobile devices. However, smart phones went a step further and enabled consumers to carry a powerful computing device in their pocket. This has changed the way we interact with computer technology. With ubiquitous access to the internet and a range of services being designed for the smart phone platform, they have not only inherited the security and privacy issues of traditional computer- and internet-based technology, but also amplified them due to the convergence of services like healthcare, banking, and Online Social Networking (OSN) applications. Therefore, with ever-increasing adoption of smart phones and their role as an integral part of daily life, the challenge is to build such devices in a manner that provides a trusted and secure environment for all sensitive applications. There have been many different proposals on how such an environment can be built, including different architectures such as software protection (including compile time and link time code hardening), application management, and hardware protection (e.g. ARM TrustZone, GlobalPlatform Trusted Execution Environment, Trusted Platform Module, and Secure Elements). In this chapter, we discuss most of the listed examples in order to highlight issues related to Secure Application Execution (SAE) on smart phone platforms. Most of these proposals rely on:

1. Pre-installation secure coding and analysis
2. Post-installation secure application management and analysis
3. Trusted hardware to provide software validation
4. Executing a portion of the activity (usually the most sensitive part of the application code and associated data) on secure and trusted hardware.

A crucial issue regarding the secure execution of sensitive applications on smart phone platforms is: how can we trust the execution environment? Most of the proposals for SAE are based, one way or another, on secure hardware that will provide some level of security, trust and possibly privacy, giving an assurance that the application in execution (at runtime) on such hardware will be secure. This means that the application code executing on the trusted hardware will run without interference from attackers and that each line of code executes as intended by the application developer (or application provider). For a secure and trusted hardware that will not enable a malicious entity to interfere with the execution of an application, we require a runtime protection mechanism. In this chapter, we will discuss such a mechanism, explain its operation and show how it can achieve secure runtime monitoring of sensitive applications on trusted hardware.

1.1 Structure of the Chapter

In section 2, we discuss the smart phone ecosystem and briefly describe the two major smart phone platforms Apple iOS and Google Android. This leads us to a discussion of the SAE frameworks including code hardening, application management and device attestation in section 3. We then discuss proposals for trusted execution environments that are usually based on secure hardware in section 4. In section 5, we address the issue of ensuring that the trusted execution environment will provide a secure and trusted application execution. This section also describes application runtime protection deployed in a trusted execution environment. Finally, in section 6 we provide concluding remarks on SAE for smart phone platforms along with suggestions for future work.

2 Smart Phone Ecosystems

We will first describe the two major smart phone platforms currently in use: Apple's iOS and Google's Android. The subsequent sections introduce the security-related provisions present on these two platforms. It is important to note that Google Android smart phones may feature varying security levels because of the large degree of flexibility left by Google to their hardware manufacturers (design licensees).

2.1 Apple's iOS Ecosystem

This section briefly outlines Apple's security ecosystem. This security ecosystem is meant to prevent insecure or malicious applications from being installed on handsets in the field.

2.1.1 Secure Boot Chain

The secure boot is the process by which Apple ensures that only signed and trusted software is loaded into iOS devices. Amongst other desirable features, this ensures that the lowest levels of software are not tampered with, and allows iOS to run only on validated Apple devices. The secure boot chain encompasses the bootloaders, kernel, kernel extensions, and baseband firmware with each component verifying the next. If any boot process component cannot be loaded or verified correctly, then the boot sequence (also called *boot-up*) is stopped. Each component that is part of the boot process must be signed by Apple. Boot chain sequence:

1. The **Boot ROM** is considered to be an implicitly trusted code embedded within the A5 processor during chip manufacturing. The Boot ROM code contains the

public key of Apple’s Root CA, which is used upon iDevice power-up. This public key allows verification that the Low-Level Bootloader has been signed by Apple before allowing it to load.
2. The **bootloaders** are a piece of software executed whenever the hardware is powered up. There are two bootloader components: the *Low-Level Bootloader (LLB)* and *iBoot*. The LLB is the lowest-level code on an Apple device that can be updated. The LLB runs several firmware setup routines. The bootloaders attempt to verify the iOS Kernel’s signature; if this verification fails the device enters into Recovery Mode (visible to the user as a ”connect to iTunes” mode).
3. The **iOS Kernel** includes an XNU kernel, system modules, services and applications. When the LLB and the iBoot finish their tasks, iBoot verifies and runs the next kernel stage of the iOS. The iOS Kernel is the ultimate authority for allowing the secure boot.

This process was designed by Apple to ensure the integrity of the booting process. Each step is checked cryptographically: this means that each OS component, including the bootloaders, kernel, kernel extensions, and baseband firmware must be signed with a certificate in order to assemble into a *chain of trust*.

2.1.2 Hardware Encryption Features

The cryptographic operations that we have just described require modular exponentiations and hashings of long data streams (executable code). These tasks are resource-consuming and require efficient hardware processing. iOS provides access to a specific API that, besides allowing the system to access such computational resources, also allows developers to add custom cryptographic operations to their applications. Such hardware acceleration features:

1. An AES [37] engine implemented on the DreamFactory Mobile Application” (DMA) path between the flash storage and the main system memory.
2. An SHA-1 [35] API allowing high-speed hashing for integrity check purposes.
3. iOS devices have two fuse-protected (non-erasable) device keys burnt into the processor during manufacturing. These keys, which are only accessible by the AES crypto-engine, are:
   a. The **User ID (UID) key**: a 256-bit AES key unique to each device. The UID key is used to bind data to a given device.
   b. The **Group ID (GID) key**: a 256-bit AES key common to all processors using Apple A5 chips. the GID key is used if required by Apple to install and restore software [39].

2.1.3 Data Security

To secure the data stored in flash memory, Apple has constructed a data protection mechanism in conjunction with the hardware-based encryption. The data protection mechanism allows the interaction of the device with incoming phone calls, which are treated as incoming events from identified sources (called IDs) and includes a remote wipe function, passcodes, and data protection, which are briefly described as follows:

1. The **Remote wipe feature** allows the device owner to sanitize the device if it is stolen or if too many passcode attempts fail. Remote wiping can be initiated *via* MDM (Mobile Device Management), Exchange, or iCloud.
2. The **Passcode** serves two purposes: it protects the device's accessibility by locking it and provides entry to the encryption/decryption keys stored on board. This ensures that certain sensitive files can be decrypted only upon successful passcode presentation by the user.
3. In addition to the above features, Apple devices have also methods for collecting and distilling entropy for creating new encryption keys on the fly. As attested by many Apple patent applications, methods range from collecting application data to the monitoring of device movements and hashing them into random information.
4. The **Key chain data protection** uses a hardware encryption accelerator, shipped with all 3GS or newer iPhone devices. The accelerator can encrypt selected sensitive data fields that many apps need to handle (*e.g.* passwords, credentials and keys). Data stored in the keychain is logically zoned to prevent applications from accessing confidential information belonging to other applications. An application developer can therefore easily manage his application's data by simply declaring it as private to his application.

2.1.4 Sandboxing

The kernel of iOS is the XNU kernel [40]. XNU is the OS kernel initially developed for the Mac OS X operating system and subsequently released as free open source software. The security model for iOS is therefore very similar to that of the Mac OS, where code is secured using signatures, sandboxing and entitlement checking. Sandboxing is a software environment where codes are isolated from each other, and where an applications access to resources is controlled by the OS. Each application has access to its own files, preferences, and network resources. The camera, GPS and other system resources on the device are accessible through an interface of abstract classes [38].

2.1.5 Application Code Signing (Vetting)

To guarantee the integrity of data stored in the mobile device, code signing (or vetting) is a process allowing the application developer to certify that an application was created by him. Once an application is signed, the system can detect any changes (be these accidental or malicious) in the signed code. Apple signs all components in the boot process (*e.g.* the bootloaders, kernel, kernel extensions, and baseband firmware). Signatures are required for all programs running on the device regardless of whether these are Apple codes or third-party applications (*e.g.* Safari). Thereby iOS avoids loading unsigned codes or applications that may be malicious.

2.2 The Android Ecosystem

Because Linux is at the heart of Android [41], most Linux security concepts also apply to Android.

2.2.1 Sandboxing

Android inherits a permission model from the Linux kernel that provides data isolation based on UIDs and GIDs. Therefore, each user has an assigned UID and one or more GIDs. To enforce data confidentiality, Android uses two concepts that permit users to access only files that they own:

1. The **Discretionary Access Control** (DAC) concept is a Linux mechanism allowing only the device owner to access her own files [44]
2. The **Mandatory Access Control** (MAC) is an OS protection mechanism that constrains the ability to access or perform certain operations on specific objects or targets. Generally, the MAC is used to control access by applications to system resources [44].

To differentiate one user from another or one user group from another, each application within a Linux system is given a UID and a GID. Each file's access rules are specified for three sets of subjects: user, group and everyone. Each subject set has valid or invalid permissions to read, write and execute a file. To restrict file access to owners only, the Android kernel sandbox uses UIDs and GIDs to enforce DAC.

2.2.2 Applications permissions

By default an Android application has no specific permissions to access mobile resources. This means that the application cannot do anything that would adversely impact [43]. However, application developers can add permissions to their applications using tags in the `AndroidManifest.xml` file. These tags allow developers to describe the functionality and the requirements of a target Android application and thereby adapt security to increase functionality. For example, an application that needs to monitor incoming SMS messages would specify:

```
uses-permission android:name="android.permission.RECEIVE_SMS"
```

2.2.3 Application Code Signing

Android requires that all apps be digitally signed with a certificate before they can be installed [42]. This functionality is used to:

- Identify the code's author,
- Detect if the application has changed, and
- Establish trust between applications

However, applications signed using the same digital signature can grant each other permission to access signature-based APIs. Such applications can also run in the same process if they share user IDs, allowing access to each other's code and data.

3 Secure Application Execution (SAE)

In this section, we briefly discuss existing secure application frameworks for smart phone or embedded platforms.

3.1 Code Hardening

A program code is a group of executable processor instructions designed to achieve a desired output. During program execution each instruction performs a certain operation. These instructions can be individually targeted by an attacker to force the processor into generating a faulty output. An example of such an attack is a fault injection attack, where the attacker uses equipment such as laser perturbations and clock manipulators to induce a fault [34]. This type of attack can be prevented by manipulating the code in such a way that either ⓐ makes it impossible for the attacker to locate and target these instructions, or ⓑ enables the code to detect induced faults during execution of the program. This code protection process is known as *code hardening*. Yet another code hardening technique is *obfuscation*. Obfuscation is defined in the Oxford Dictionary as "*making something obscure, unclear and unintelligible*" [**?**]. In a software development context, obfuscation is the deliberate act of creating a source and/or machine code that is difficult for other programmers to understand and manipulate. Program developers may deliberately obfuscate code to conceal its purpose or logic in order to prevent tampering. Because the attacker does not know exactly what each instruction does, it becomes harder to inject faults into specific software functions. A further common method for avoiding faults is *redundancy*. Redundancy in this context involves duplicating critical code parts. The main principle behind this technique is that induced faults are detected by executing the duplicate codes and checking whether execution results match or not. If both codes generate identical results, then the execution is considered fault-free; otherwise, execution is terminated. The redundant code may be inserted either into the source code or into the machine code. In the case of source-level injection, source code has to pass through a tool, called a source-to-source rewriter, which essentially inserts redundancy by duplicating selected statements. Source-to-source rewriters, however, suffer from major drawbacks. Firstly, modern compilers are equipped with code optimisation tools. One such tool is the Common Subexpression Elimination (CSE) tool [15], which removes redundant expressions/statements. During compilation the CSE searches for identical expressions and removes them. One of the great advantages of CSE is that it reduces the program size and speed by removing duplicated codes, but this risks undoing the security protection provided by redundant code execution. To ensure that sufficient redundancy survives the CSE and remains in the generated code, the source-to-source rewriter inserts either; i) Non-optimised and non-analysed code by disabling the CSE or ii) a code that is complex enough to withstand the compiler optimisation and analysis process. Secondly, source-to-source rewriters are very dependent on the language and the compiler being used. Hence, they need to be redeveloped (ported) for every programming language. In other words, neither the protection nor the minimal performance overhead can be ported between compilers and languages. As a result of these drawbacks, it still remains a challenge to guarantee the presence of only the necessary degree of redundancy with acceptable performance overheads. It can be very difficult or in some cases impossible to have redundant source code statements that i) will survive compiler optimisation, and ii) will not limit the compiler's existing analysis and optimisation scope. To avoid source-to-source rewriter problems, in certain cases redundancy is inserted into the binary code of the program. Such tools are known as link-time rewriters. These rewriters do not suffer from the drawbacks of source-to-source code rewriters. However, they do suffer from a lack of high-level semantic information such as symbol and type information. This lack of information limits the precision and scope of protection provided by binary code rewriters. The best example of a binary rewriter is Diablo [21].

3.2 Device Attestation

Device Attestation is a technique allowing a verifying entity $\mathbb{V}$ to check that the hardware, the firmware and/or the software of a proving entity $\mathbb{P}$ are genuine. $\mathbb{V}$ is called the *verifier* (or the challenger or the authenticator) whereas $\mathbb{P}$ is called the *prover* (or the attestator). In this section, we will distinguish two common device attestation method variants. The distinction between the two methods, called *remote attestation* and *local attestation*, is based on $\mathbb{V}$'s location and on $\mathbb{V}$'s access to $\mathbb{P}$.

3.2.1 Remote Attestation:

This concept was first promoted by the Trusted Computing Group (TCG) and implemented in the Trusted Platform Module (TPM) specifications [9]. In most modern telecommunication services remote attestation is widely used for authentication and is referred to as Direct Anonymous Attestation (DAA) [11]. DAA is a method by which one $\mathbb{P}$ can prove to another $\mathbb{V}$ that a given secret statement is true without revealing any information about $\mathbb{P}$'s secret apart from the fact that the statement is indeed true. This attestation is the means by which trusted software proves to remote parties that it is trustworthy, thereby confirming that a remote server is communicating with authentic software rather than malware. For instance, a remote bank server could use DAA to ensure that the banking application in a particular OS has not been changed. At present, there are several ways to provide a Secure Element (SE) to allow the storage of a root-of-trust for mobile devices. The best known implementations are FreeScale's i.MX53 and Texas Instruments M-Shield. There are three SE categories:

- Embedded SEs, generally used to provide security for Near Field Communication (NFC);
- Embedded hardware SEs,
- and Removable hardware SEs implemented in form-factors such as smart-cards and secure SD cards.

To allow the regeneration of security by replacing the system's SE in case of breach, the most popular SEs are tamper-resistant smartcards. In this section we will only consider tamper-resistant security chips implementing remote attestation available on trusted hardware modules such as TPMs. Remote attestation is a technique allowing $\mathbb{P}$ to attest to $\mathbb{V}$ that $\mathbb{P}$s hardware, firmware and software are genuine. Remote attestation allows a remote entity $\mathbb{V}$ to reach a level of trust concerning the integrity of a second remote entity $\mathbb{P}$. Because $\mathbb{P}$ and $\mathbb{V}$ are at a distance from each other, cryptographic keys must be used to convince $\mathbb{P}$ and $\mathbb{V}$ that information is being exchanged between them and not between one of the parties and an opponent. The remote authentication process breaks down into two steps. The first step, called "*integrity measurement*" involves the measurement of different system parameters by $\mathbb{P}$. $\mathbb{P}$ might collect information such as BIOS, OS and kernel execution times, system memory consumption, installed software, user access patterns, version numbers, stack level usage, and data hash imprints. This information μ can be monitored under nominal conditions or under randomly chosen working conditions. μ can be a system response to a challenge sent by $\mathbb{V}$ (e.g. compress file X using the built-in compression function Y and measure the stack user imprint during the compression process), jointly chosen with $\mathbb{P}$, or can result from the processing and monitoring of user-generated activity. After this data collection phase, $\mathbb{P}$ and $\mathbb{V}$ execute a *remote attestation protocol.* This protocol is a public-key cryptographic interaction secure against man-in-the-middle attacks, by which the $\mathbb{P}$ and $\mathbb{V}$ check each other's knowledge of respective secret keys and create a session key allowing $\mathbb{P}$ to safely transmit μ to $\mathbb{V}$. The μ allows $\mathbb{V}$ to ascertain that $\mathbb{P}$ is malware-free. When $\mathbb{V}$ is convinced that μ matches good values (either known or re-computed), $\mathbb{V}$ issues a digital signature. The detailed description of remote attestation protocols falls beyond the goal of this introductory discussion, but is briefly summarised as follows. Both parties use public-key key exchange, public-key encryption and signatures to create a secure channel through which μ will later transit. To prevent malware from emulating $\mathbb{P}$'s behavior, secret operations and measurement data storage do not take place in $\mathbb{P}$'s open hardware region but in $\mathbb{P}$'s TPM $\mathbb{P}_{\text{TMP}}$ whose public-keys are certified by some certification authority $\mathbb{A}$. A specific register bench of $\mathbb{P}_{\text{TMP}}$, called the Platforms Configuration Register (PCR), is devoted to measurement data storage. Very schematically, an application App running on $\mathbb{P}$ starts by generating public/private encryption and signature key-pairs and submits these keys to $\mathbb{P}_{\text{TMP}}$ to be certified. $\mathbb{P}_{\text{TMP}}$ hashes App, signs its digest and gets an Attestation Identity Key (AIK) (see section 4.6) that $\mathbb{P}_{\text{TMP}}$ returns to $\mathsf{App}.\mathsf{App}$ uses $\mathbb{P}$'s communication stack to send the AIK and $\mathbb{P}_{\text{TMP}}$'s certificates to $\mathbb{V}$, which checks that $\mathbb{P}$ is indeed known to $\mathbb{V}$ (*i.e.* present in the database of devices with which $\mathbb{V}$ is entitled to communicate) and that all aforementioned signatures and certificates are correct. If this succeeds, $\mathbb{P}$ and $\mathbb{V}$ establish a secure channel and start communicating measurement information. In general, the attestation protocol described above is usually run twice, with $\mathbb{P}$ and $\mathbb{V}$ switching roles. For added security, the code of $\mathbb{V}$ that validates the measurements is not published but is provided as a *cloud-based security-service* that does not expose (to potential attackers) the models that reflect $\mathbb{P}$'s structure [45]. Cloud-based verifiers also have the advantage of allowing a complete (memory and time consuming) virtual emulation of relatively complex $\mathbb{P}$'s so as to infer their expected measurements quickly and accurately.

3.2.2 Local Attestation:

Without strict control over the boot process of an operating system, unauthorised software can be introduced via the boot sequence or the BIOS and attack the platform in devastating ways [48, 46, 47]. In comparison to remote attestation, which is done by a remote $\mathbb{V}$, in local attestation, the verification is a sort of security self-test, allowing a platform $\mathbb{P}+\mathbb{V}$ to protect itself from infection. This is achieved using a variety of techniques ranging from code hash values to real-time hardware watchdogs. Because it is impossible to enforce security if the boot process is compromised, local attestation tries to carefully ascertain that that the boot process does not feature any anomalous warning signs. This is done in two steps: **Authenticated Boot:** Upon power-up the system is executed and measured to infer μ. Then the PCRs are initialized with μ using a specific PCR extension function that updates (extends) the PCR value using the formula:

$$\text{PCR}_{i+1} \leftarrow \text{SHA1}(\text{PCR}_i|\mu_{i+1})$$

where the vertical bar | stands for data concatenation. Upon reboot, the system will re-perform all the measurements $\mu_1,\ldots,\mu_\ell$ (where ℓ is the number of PCR extensions done so far, and ascertain that at least the previously measured features remain unaltered. **Secure Boot:** The previous accumulation idea can also be applied to the "measurements" μ_i consisting of the digital signatures of the various software components present in the platform. These signatures are recorded in the TPM upon software installation. Here the accumulation process is not a simple hash but a signature accumulation and screening process [49, 51, 50] allowing accumulation of ℓ individual RSA signatures $s_1,\ldots,s_\ell$ into one ”global” RSA signature s checked together:

$$s = \prod_{i=1}^{\ell} s_i \bmod n$$

Note that when program k is uninstalled, s must be corrected by updating it as follows $s \leftarrow s \times s_k^{-1} \bmod n$. The above assumes that the code implementing the signature verification is itself secure. To that end, a small part of $\mathbb{P}$'s code is saved in an inerasable (immutable) ROM space called the *boot-ROM*. The boot-ROM measures the OS loader's components and, upon successful measurement, hands over execution to the OS loader. The OS loader will measure the OS and again, hand over execution to the OS and its drivers, which, ultimately, will measure the applications installed in the platform. All these measurements are done by verifying a tree of digital signatures where the ancestor is the measuring code and where the offspring are measured codes. When the process ends, the entire platform has been attested and is considered as having reached an *approved configuration*.

4 Trusted Execution Environment

In this section we briefly introduce some of the proposals for a secure and trusted application execution and data storage.

4.1 Mobile Trusted Module (MTM)

The growth of mobile computing platforms has encouraged the Trusted Computing Group (TCG) to propose the Mobile Trusted Module (MTM). In this section, we briefly discuss the MTM architecture and its operations along with how this differs from the TPM. In section 4.6, we will discuss the proposed TPM MOBILE whose origins lie in the TPM v1.2. The ecosystems of mobile computing platforms (e.g mobile phones, tablets, and PDAs) are fundamentally different from traditional computing platforms. Therefore, the architecture of the MTM has some features from the TPM specification, but it introduces new features to support its target environment. The main changes introduced in the MTM that make it different from the TPM specification are:

1. The MTM is required not only to perform integrity measurements during the device boot up sequence, but also to enforce a security policy that prevents the system from initiating securely if it does not meet the trusted (approved) state transition.

2. The MTM does not have to be in the hardware; it is considered as a functionality, which can be implemented by device manufacturers as an add-on to their existing architectures.
3. The MTM specification supports parallel installations of MTMs associated with different stakeholders.

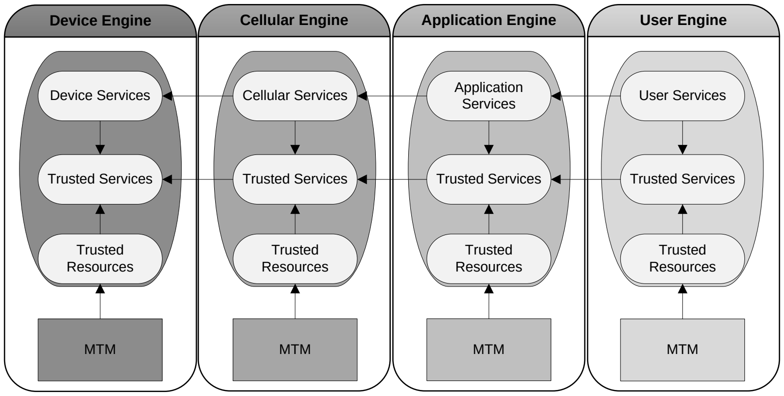

Fig. 1: Possible (Generic) Architecture of Mobile Trusted Platform

The MTM specification [6] is dynamic and scalable to support the existence of multiple MTMs interlocked with each other, as shown in Figure 1. The MTM refers to them as engines, where each of these engines is under the control of a stakeholder. Stakeholders may include the device manufacturer (Device Engine), the mobile network operator (Cellular Engine), the application provider (Application Engine), and the user (User Engine); as illustrated in Figure 1. Each engine is an abstraction of trusted services associated with a single stakeholder. Therefore, on a mobile platform there can be a single hardware that supports the MTM functionality and is accessed by different engines. Each mobile platform abstract engine supports:

1. Provision to implement trusted and non-trusted services (normal services) associated with a stakeholder.
2. Self-test to ascertain the trustworthiness of its own state.
3. Storage of Endorsement Key (EK) (which is optional in MTM) and/or Attestation Identification Keys (AIKs).
4. Key migration.

We can further dissect abstract engines as components of different services as shown in Figure 2. The non-trusted services in an engine cannot access the trusted resources directly. They have to use the Application Programming Interfaces (APIs) implemented by the trusted services. The trusted resources, including reporting, verification and enforcement, are new concepts that are introduced in the MTM specifications. The MTM measurement and storage services shown in Figure 2 are

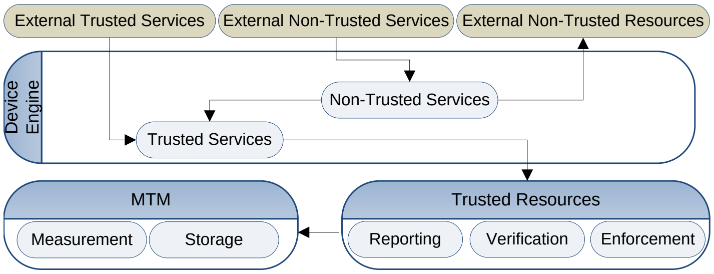

Fig. 2: Generic Architecture of an Abstract Engine

similar to the TPMs discussed in previous sections. The MTM specification defines two variants of the MTM profile depending upon who is the owner of a particular MTM. They are referred as Mobile Remote Ownership Trusted Modules (MRTMs) and Mobile Local Ownership Trusted Modules (MLTMs). The MRTM supports the remote ownership, which is held either by the device manufacturer or the mobile network operator, while the MLTM supports the user ownership. The roots of trust in the MTM include those discussed in TPM, including Root of Trust for Storage (RTS), Root of Trust for Measurement (RTM), and Root of Trust for Reporting (RTR); however, the MTM introduces two new roots of trust known as Root of Trust for Verification (RTV) and Root of Trust for Enforcement (RTE). During MTM operations on a trusted mobile platform, we can logically group different roots of trust; for example, RTM and RTV are grouped together to perform an efficient measure-verify-extend operation illustrated in Figure 5. Similarly, RTS and RTR can be grouped together to deal with secure storage and trustworthiness of the mobile platform. The MTM operations as shown in Figure 5 begin when a process starts execution, and they are:

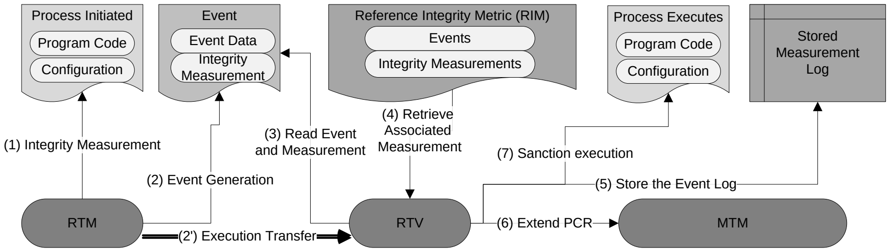

Fig. 3: MTM Measurement and Verification Process

1. The RTM performs integrity measurements on the initiated process.
2. The RTM registers an event that includes the event data (application/process identifier) and associated integrity measurement. The RTM then transfers the execution to the RTV.

3. The RTV reads the event registered by the RTM.
4. The RTV then searches the event details via the Reference Integrity Metric (RIM). This includes the trusted integrity measurements associated with individual events, populated by the engine owner. This operation makes the MTM different from the TPM, as the latter does not make any decision regarding the trustworthiness of the application or process. However, MTM does so via the comparison performed by the RTV to verify that the integrity measurement performed by the RTM matches the one stored in the RIM. If the integrity measurement does not match, the MTM terminates the execution or disables the process. If the verification is successful then it proceeds with steps 5 and 6 along with sanctioning the execution (step 7).
5. The RTV registers the event in the measurement logs. These logs give the order in which the measurements were made to generate the final (present) value of the associated PCR.
6. The RTV extends the associated PCR value that is stored in the MTM.
7. If verification is successful, the execution of the process is sanctioned.

4.2 M-Shield

Texas Instruments has designed the M-Shield as a secure execution environment for the mobile phone market [5]. Unlike ARM TrustZone, the M-Shield is a standalone secure chip, and it provides secure execution and limited non-volatile memory. It also has internal memory (on-chip memory) to store runtime execution data [13] and this makes it less susceptible to attacks on off-chip memory or communication buses[1] [12].

4.3 ARM TrustZone

Similar to the MTM, the ARM TrustZone also provides the architecture for a trusted platform specifically for mobile devices. The underlying concept is the provision of two virtual processors with hardware-level segregation and access control [14, 7]. This enables the ARM TrustZone to define two execution environments described as Secure world and Normal world. The Secure world executes the security- and privacy-sensitive components of applications and normal execution takes place in the Normal world. The ARM processor manages the switch between the two worlds. The ARM TrustZone is implemented as a security extension to the ARM processors (e.g. ARM1176JZ(F)-S, Cortes-A8, and Cortex-A9 MPCore) [7], which a developer can opt to utilise if required.

[1] The memory or communication buses mentioned are between a TPM and other components on a motherboard, rather than the on-chip memory and communication buses.

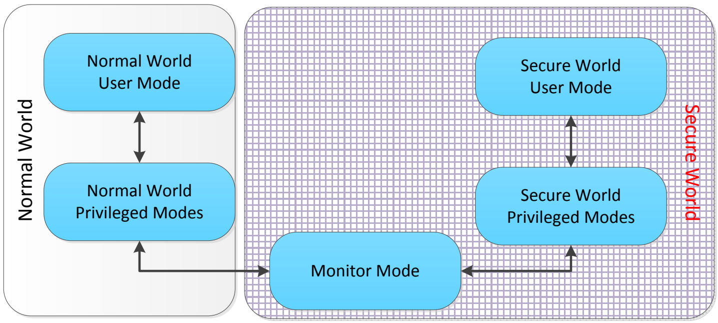

Fig. 4: Generic architectural view of ARM TrustZone

4.4 *GlobalPlatform Trusted Execution Environment (TEE)*

The TEE is GlobalPlatform's initiative [3, 4, 8] for mobile phones, set-top boxes, utility meters, and payphones. GlobalPlatform defines a specification for interoperable secure hardware, which is based on GlobalPlatform's experience in the smart card industry. It does not define any particular hardware, which can be based on either a typical secure element or any of the previously discussed tamper-resistant devices. The rationale for discussing the TEE as one of the candidate devices is to provide a complete picture. The underlying ownership of the TEE device still predominantly resides with the issuing authority, which is similar to GlobalPlatform's specification for the smart card industry [1].

4.5 *Secure Elements*

A secure element is an electronic chip which can securely store and execute programs. Examples are the Universal Integrated Circuit Card (UICC), the Embedded Secure Element, and Secure Memory Cards. Secure elements available on most of the Google Android supported devices conform to the Java Card specifications [2]. A generic framework to use the secure elements (or even Subscriber Identity Module: SIM card) is shown in Figure 5 and discussed below. An application installed on a smart phone platform can have a collaborative application available on the secure element. The collaborative application has the responsibility for executing and storing security- and/or privacy-preserving functionality and data. The application, once installed and executing on the smart phone platform, provides a feature-rich interface to the user, while communicating with collaborative applications when required.

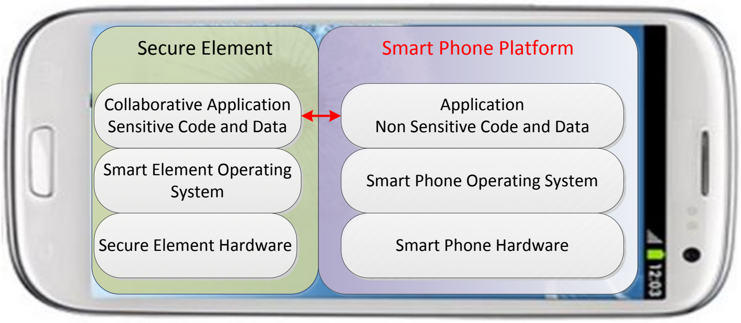

Fig. 5: Generic architectural view of Secure Element-based Framework

4.6 TPM MOBILE

The TPM chip, whose specification is defined by the Trusted Computing Group [10], is known as a hardware root-of-trust into the trusted computing ecosystem. Currently it is deployed to laptops, PCs, and mobiles and is produced by manufacturers including Infineon, Atmel and Broadcom. At present, the TPM is available as a tamper-resistant security chip that is physically bounded to the computers motherboard and controlled by software running on the system using well-defined commands. The TPM MOBILE with Trusted Execution Environment has recently emerged; its origin lies in the TPM v1.2 a with some enhancements for mobile devices [9] . The TPM provides:

1. The **Roots of trust** include hardware/software components that are intrinsically trusted to establish a chain of trust that ensures only trusted software and hardware can be used (see the Mobile Trusted Module (MTM) section).
2. The **Platform Configuration Register ”PCR”** in the most modern TPM includes 24 registers. It is used to store the state of system measurements. These measurements are represented normally by a cryptographic hash computed from the hash values (SHA-1) of components (applications) running on the platform. PCRs cannot be written directly; data can only be stored by a process called extending the PCR.
3. The **RSA keys**: There are two types of RSA keys that TPM generates and which are considered as *root keys* (they never leave the TPM):

    a. **Endorsement Key (EK)**: This key is used in its role as a *Root of Trust for Reporting*. During the installation of an owner in the TPM, this key is generated by the manufacturer with a public/private key pair built into the hardware. The public component of the EK is certified by an appropriate CA, which assigns the EK to a particular TPM. Thus, each individual TPM has a unique platform EK. For the private component of the EK, the TPM can sign assertions about the trusted computer's state. A remote computer can verify that those assertions have been signed by a trusted TPM.

b. **Storage Root Key (SRK)**: This key is used to protect other keys and data via encryption.
c. **Attestation Identity Keys (AIKs)**: The AIK is used to identify the platform in transactions such as platform authentication and platform attestation. Because of the uniqueness of the EK, the AIK is used in remote attestation by a particular application. The private key is non-migratable and protected by the TPM and the public key is encrypted by a storage root key (or other key) outside the TPM with the possibility to be loaded into the TPM. The security of the public key is bootstrapped from the TPM's EK. The AIK is generally used for several roles: signing/reporting user data; storage (encrypting data and other keys); and binding (decrypting data, used also for remote parties).

4.7 Overseeing the Overseer

In the proposals discussed in this chapter, the burden of SAE is moved to the trusted execution environment in one way or another. The security and reliability of the trusted execution environment has to be not only adequate, but in certain cases provable. We need to build a trusted environment that can ensure all application code being executed on it will be protected from any runtime tampering by a resourceful malicious entity. In the subsequent sections, we address the security and reliability of the trusted execution environment and how we can ensure a trusted application execution.

5 Remaining Security Challenges of Application Execution

Before we begin a detailed discussion of this section we summarise what has been presented so far. Mobile devices are composed of hardware, operating system and applications. Techniques such as code hardening and centralised vetting try to protect mobile applications from malicious intruders by inserting clues, such as redundant statements, into the application executables and verifying them centrally before they are installed. However, they cannot protect against attacks targeting the operating system or the underlying hardware. To counteract such attacks, device attestation is needed. Device attestation such as TPM ensures only that the necessary components of the operating system are started securely during device booting. However, this does not provide any protection against attacks that can take place after the operating system is up and running. Manufacturers try to tackle this challenge by using various techniques including MTM, ARM TrustZone, M-Shield and GlobalPlatforms TEE. The common theme of these protections is to provide a secure execution space for critical applications, for example PIN verification, which is segregated from the main execution space. In spite of all the efforts to secure embedded systems, there still remain significant threats/vulnerabilities that can be exploited by dedicated attackers. The questions one may ask regarding the security of embedded systems are:

1. How do we make sure that the hardware processor is secure and free of malicious entities such as hardware Trojans?
2. If we only execute selected applications/programs inside the secure zone, what happens to the other applications?

In the subsequent sections we discuss these security challenges and their possible solutions.

5.1 Pre-deployment/Post-production Device Verification

Recent economic conditions have forced embedded system manufacturers to outsource their production processes to cheaper cost structure countries. While this significantly reduces the total production cost, it also makes it much easier for an attacker to compromise the supply chain for components used in critical business and military applications, and replace them with defective components. This threat to the electronic components supply chain is already a cause for alarm in some countries [18, 20]. For this reason, some governments have been subsidising high-cost local foundries to produce components used in military applications [19]. However, this is not an affordable solution for most developing countries and commercial applications. According to [23], the incidence of defective components increased from 3,868 in 2005 to 9,356 in 2008. Such defective electronic components have at least the following ramifications: 1) original component providers incur an irrecoverable loss due to the sale of often cheaper counterfeit components; 2) low performance of defective products (that are often of lower quality and/or cheaper older generations of a chip family) affects the overall efficiency of the integrated systems that unintentionally use them, which could in turn harm the reputation of authentic providers; 3) unreliability of defective devices could render the integrated systems that unknowingly use the parts unreliable, potentially affecting the performance of weapons, airplanes, cars or other crucial applications [24]; and 4) untrusted defective components may have intentional malware or some backdoor for spying, remotely controlling critical objects and/or leaking secret information. These ramifications and their growing presence in the market make them important problems to address. Traditionally, the integrity of software codes on personal computers and other platforms is verified by using hash values and digital signatures. The software developer hashes the entire software code and signs it with his private key. Later, the person using it verifies the signature before installing it. This scheme, however, suffers from a major drawback in embedded devices. The reason is that hashing the entire program memory is often impossible due to the read protection countermeasures that embedded systems implement. In certain scenarios the software developer can provide users with the source code so the users can manually check it before using it. However, for commercial and intellectual property reasons this is not possible in many cases. Sometimes it may be necessary to verify the hardware before installing the software. In this case the easiest way of doing it would be to verify the integrity of the netlist[1] of the target device. A definition of the term netlist can also be found in [36]. However, as with software codes this is generally impossible as companies do not reveal such information for commercial and intellectual property reasons. Therefore, to verify an embedded system before it is deployed/inserted into larger electronic equipment we need to find alternative but reliable methods that can help us verify its integrity. In [28] Paul et al. demonstrated that side channel information, such as power consumption, carries information about the data processed at runtime. Furthermore, it is possible that the same information can reveal much more information about the internal state of the device than just runtime data. In [30, 29, 31] the authors demonstrated that the side channel leakage of an embedded device contains information about the executed instructions and that it can be used to partially reverse engineer them. In [32] Mehari et al. have improved the recognition rate to fully reverse engineer them. The authors of [25] also demonstrate that side channel information can be used to spot additional modules that were not part of the original design of the device, such as hardware Trojans. From the above work it is reasonable to conclude that side channel information can be effectively used to verify embedded devices before they are deployed. In the context of embedded system software integrity verification, Mehari et al. have demonstrated its possibility in their paper [33].

5.2 Runtime Security Protection

As discussed in the previous sections, several techniques have been proposed and deployed to secure embedded systems. However, they still remain vulnerable to a range of attacks. This is true partly because security was not the main criterion of the original processor design and it has not changed much since then. On some occasions researchers have tried to address this problem by integrating security into the processor design. Integrated hardware protections are implemented by hardware manufacturers. One of the hardware protections is hardware block redundancy, in which selected or all hardware blocks on the embedded chip are implemented more than once. During runtime the program is executed by all blocks and a comparator compares the results. In [34], several varieties of this protection are discussed in detail. Figure 6 illustrates a simple duplicate of a hardware block. The decision block either resets the device or invokes a specifically designed reaction algorithm when a fault is detected by the comparator. In another approach Arora et al. [26] demonstrated that the control flow jumps and instruction integrity of embedded programs

[1] a list of logic gates and a textual description of their interconnections which make up an electronic circuit.

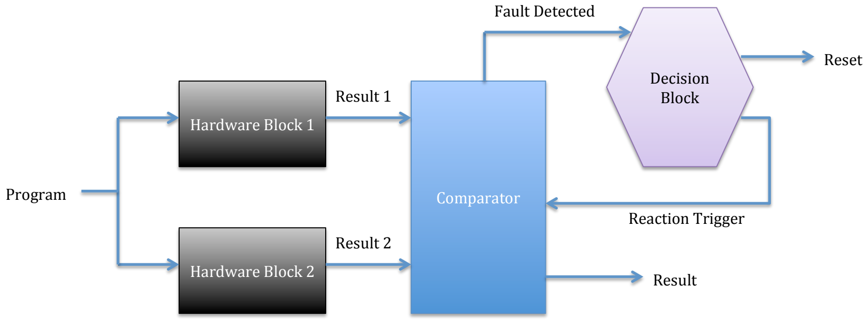

Fig. 6: A simple hardware redundancy with two identical blocks.

can be verified on the fly with minimal overhead by integrating a security module into the core design. They discussed the idea that if the security attributes of the program can be extracted during compilation, then these attributes can be used at runtime to ensure the program is behaving correctly. If problems are detected, either shutdown or reset signals will be generated. Krutartha et al. [27] discussed a similar approach, designing a security module as part of the processor code, to secure multiprocessors against code injection attacks. They introduced a new module called the monitor processor that monitors communication between the individual processors. In an N-processor core the individual processors communicate with each other through a First In First out (FIFO) queue structure. In their approach the program is broken down to basic blocks. The basic blocks are then attributed with two FIFO instructions that notify the monitor processor of the start and finish of each basic block. Although these approaches show some success in securing embedded system application at runtime, they are not exhaustively researched to be a viable commercial solution.

6 Conclusion

In this chapter, we discussed the importance of SAE for mobile devices, especially smart phones. The chapter also briefly described the Apple iOS and Google Android ecosystem, along with how they secure not only their respective platforms but also the applications running on them. Different proposals put forward to provide a secure and trusted environment for execution and data-storage of sensitive applications were discussed. These proposals included MTM, TPM MOBILE, M-Shield, GlobalPlatform TEE, ARM TrustZone and Secure Elements. One thing in common to most of these proposals is a secure and trusted hardware-based execution environment that serves the smart phone platform and applications. These trusted execution environments have to protect the application and its sensitive data from tampering during application execution. We therefore discussed potential runtime security protection systems that ensure an application executes without interference and/or tampering from any external entity (malicious or otherwise). Our next research direction in attempting to secure embedded applications is designing and integrating a hardware module into the core processor that is capable of protecting program attributes, such as control flow jumps, runtime data and executed instructions. Other important issues that we will attempt to solve are IP protection and hardware attestation issues in embedded environments.

Acknowledgement

Mehari G. Msgna is sponsored by the Information Network Security Agency, Addis Ababa, Ethiopia.

References

1. GlobalPlatform: GlobalPlatform Card Specification, Version 2.2, (2006)
2. Java Card Platform Specification; Application Programming Interface, Runtime Environment Specification, Virtual Machine Specification (2006). URL `http://java.sun.com/javacard/specs.html`
3. GlobalPlatform Device: GPD/STIP Specification Overview. Specification Version 2.3, GlobalPlatform (2007)
4. GlobalPlatform Device Technology: Device Application Security Management - Concepts and Description Document Specification. Online (2008)
5. M-Shield Mobile Security Technology: Making Wireless Secure. Whilte Paper, Texas Instruments (2008). URL `http://focus.ti.com/pdfs/wtbu/ti_mshield_whitepaper.pdf`
6. TCG Mobile Trusted Module Specification. Online (2008)
7. ARM Security Technology: Building a Secure System using TrustZone Technology. White Paper PRD29-GENC-009492C, ARM (2009)
8. GlobalPlatform Device Technology: TEE System Architecture. Specification Version 0.4, GlobalPlatform (2011)
9. Trusted Platform Module Main Specification.
10. Tusted Computing Group. http://www.trustedcomputinggroup.org
 Online (2011). URL `http://www.trustedcomputinggroup.org/resources/tpm_main_specification`
11. Brickell, E., Camenisch, J., Chen, L.: Direct Anonymous Attestation. In: Proceedings of the 11th ACM Conference on Computer and Communications Security, CCS '04, pp. 132–145. ACM, New York, NY, USA (2004). DOI 10.1145/1030083.1030103. URL `http://doi.acm.org/10.1145/1030083.1030103`
12. Halderman, J.A., Schoen, S.D., Heninger, N., Clarkson, W., Paul, W., Calandrino, J.A., Feldman, A.J., Appelbaum, J., Felten, E.W.: Lest we remember: cold boot attacks on encryption keys. In: Proceedings of the 17th conference on Security symposium, pp. 45–60. USENIX Association, Berkeley, CA, USA (2008)
13. Kostiainen, K., Ekberg, J.E., Asokan, N., Rantala, A.: On-board credentials with open provisioning. In: Proceedings of the 4th International Symposium on Information, Computer, and

Communications Security, ASIACCS '09, pp. 104–115. ACM, New York, NY, USA (2009). DOI http://doi.acm.org/10.1145/1533057.1533074

14. Wilson, P., Frey, A., Mihm, T., Kershaw, D., Alves, T.: Implementing Embedded Security on Dual-Virtual-CPU Systems. IEEE Design and Test of Computers 24, 582–591 (2007)
15. Steven S. Muchnick: Advanced Compiler Design and Implementation Morgan Kaufmann (1997), pp. 378–396.
16. Hagai Bar-El, Hamid Choukri, David Naccache and Michael Tunstall and Claire Whelan: The Sorcerer's Apprentice Guide to Fault Attacks In: IACR Cryptology ePrint Archive, 2004.
17. Jonas Maebe, Ronald De Keulenaer, Bjorn De Sutter and Koen De Bosschere: Mitigating Smart Card Fault Injection with Link-Time Code Rewriting: A Feasibility Study Financial Cryptography, pp. 221–229, 2013.
18. Defense Advanced Research Projects Agency: DARPA BAA06-40, A TRUST for Integrated Circuits Visited, September 2014.
19. Defense Science Board Task Force: High Performance Microchip Supply URL `http://www.acq.osd.mil/dsb/reports/ADA435563.pdf` Visited, September 2014.
20. Joseph I. Lieberman: The national security aspects of the global migration of the U.S. semiconductor industry URL `http://www.fas.org/irp/congress/2003\_cr/s060503.html` Visited, September 2014.
21. Diablo: Diablo Is A Better Link-Time Optimizer URL `https://diablo.elis.ugent.be/` Visited, October 2014.
22. Oxford Dictionaries: Definition of obfuscate URL `http://www.oxforddictionaries.com/definition/english/obfuscate`
23. U.S. Department Of Commerce: Defense Industrial Base Assessment: Counterfeit Electronics Bureau of Industry and Security, Office of Technology Evaluation, Visited January, 2010, URL `http://www.bis.doc.gov/defenseindustrialbaseprograms/osies/defmarketresearchrpts/final\_counterfeit\_electronics\_report.pdf`
24. Koushanfar, F., Sadeghi, A.-R. and Seudie, H.: EDA for secure and dependable cybercars: Challenges and opportunities In: Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE, pp. 220–228, 2012.
25. D. Agrawal and S. Baktir and D. Karakoyunlu and P. Rohatgi and B. Sunar: Trojan Detection using IC Fingerprinting In: Security and Privacy, 2007. SP '07. IEEE Symposium on, pp. 296–310, 2007.
26. Arora, D., Ravi, S. and Raghunathan, A. and Jha, N.K.: Secure embedded processing through hardware-assisted run-time monitoring In: Design, Automation and Test in Europe, 1, pp. 178–183 (2005). DOI 10.1109/DATE.2005.266
27. Patel, K., Parameswaran, S. and Seng Lin Shee: Ensuring secure program execution in multiprocessor embedded systems: A case study In: IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS), pp. 57–62 (2007).
28. Paul Kocher, Joshua Jaffe and Benjamin Jun: Differential Power Analysis In: Michael J. Wiener, *editors*, CRYPTO '99, 1666 of LNCS, pp. 388–397, 1999. Santa Barbara, California USA. Springer.
29. Dennis Vermoen, Marc F. Witteman and Georgi Gaydadjiev: Reverse Engineering Java Card Applets Using Power Analysis In: Damien Sauveron and Constantinos Markantonakis and Angelos Bilas and Jean-Jacques Quisquater, *editors*, WISTP, 4462 of Lecture Notes in Computer Science, pp. 138–149, 2007. Heraklion, Crete, Greece. Springer.
30. Jean-Jacques Quisquater and David Samyde: Automatic Code Recognition for Smartcards Using a Kohonen Neural Network In: CARDIS, November 21-22, 2002. San Jose, CA, USA. USENIX.
31. Thomas Eisenbarth, Christof Paar and Björn Weghenkel: Building a Side Channel Based Disassembler In: Marina L. Gavrilova and Chih Jeng Kenneth Tan and Edward D. Moreno, *editors*, Transactions on Computational Science, 6340 of Lecture Notes in Computer Science, pp. 78–99, 2010. Springer.

32. Mehari Msgna, Konstantinos Markantonakis and Keith Mayes: Precise Instruction-Level Side Channel Profiling of Embedded Processors In: 10th International Conference Information Security Practice and Experience (ISPEC), pp. 129–143. Fuzhou, China, May 5-8, 2014. DOI 10.1007/978-3-319-06320-1_11
33. Mehari Msgna, Konstantinos Markantonakis, David Naccache and Keith Mayes: Verifying Software Integrity in Embedded Systems: A Side Channel Approach In: 5th International Workshop Constructive Side-Channel Analysis and Secure Design (COSADE), pp. 261–280. Paris, France, April 13-15, 2014. DOI 10.1007/978-3-319-10175-0_18
34. Hagai Bar-El, Hamid Choukri, David Naccache, Michael Tunstall, and Claire Whelan: The Sorcerer's Apprentice Guide to Fault Attacks In: IACR Cryptology ePrint Archive, Volume 2004 (2004). URL http://eprint.iacr.org/2004/100
35. What is SHA-1 https://en.wikipedia.org/wiki/SHA-1
36. Netlist Definition. Xilinx http://www.xilinx.com/itp/xilinx10/help/iseguide/mergedProjects/constraints_editor/html/ce_d_netlist.htm
37. What is SHA-1 https://en.wikipedia.org/wiki/SHA-1
38. iOS Security Sandbox white paper https://www.cs.auckland.ac.nz/courses/compsci702s1c/lectures/rs-slides/6-iOS-SecuritySandbox.pdf
39. URL https://www.apple.com/privacy/docs/iOS_Security_Guide_Oct_2014.pdf
40. http://en.wikipedia.org/wiki/XNU
41. http://en.wikipedia.org/wiki/Android
42. http://developer.android.com/tools/publishing/app-signing.html
43. http://developer.android.com/guide/topics/security/permissions.html
44. What is MAC/DAC https://www.internetsociety.org/sites/default/files/02_4.pdf
45. http://www.tclouds-project.eu/downloads/factsheets/tclouds-factsheet-07-attestation.pdf
46. T. Zeller, The ghost in the CD; Sony BMG stirs a debate over software used to guard content, The New York Times, c1, November 14, 2005.
47. http://en.wikipedia.org/wiki/CIH_(computer_virus)
48. Vanessa Gratzer, David Naccache, Alien vs. Quine, the Vanishing Circuit and Other Tales from the Industrys Crypt, Advances in Cryptology - EUROCRYPT 2006, Lecture Notes in Computer Science Volume 4004, 2006, pp 48-58
49. Benoît Chevallier-Mames, David Naccache, Pascal Paillier, David Pointcheval, How to Disembed a Program? Cryptographic Hardware and Embedded Systems - CHES 2004, Lecture Notes in Computer Science Volume 3156, 2004, pp 441-454
50. Mihir Bellare, Juan A. Garay, Tal Rabin, Fast batch verification for modular exponentiation and digital signatures, Advances in Cryptology EUROCRYPT'98, Lecture Notes in Computer Science Volume 1403, 1998, pp 236-250
51. Josh Benaloh, Michael de Mare, One-way accumulators: a decentralized alternative to digital signatures, EUROCRYPT '93 Workshop on the theory and application of cryptographic techniques on Advances in cryptology, Pages 274-285, Springer-Verlag.