Secure and Trusted Application Execution on Embedded Devices

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

Information Security Group, Smart Card Centre, Royal Holloway, University of London, United Kingdom
{k.markantonakis, r.n.akram, mehari.msgna.2011}@rhul.ac.uk

Abstract. Embedded devices have permeated into our daily lives and significant day-to-day mundane tasks involve a number of embedded systems. These include smart cards, sensors in vehicles and industrial automation systems. Satisfying the requirements for trusted, reliable and secure embedded devices is more vital than ever before. This urgency is also strengthened further by the potential advent of the Internet of Things and Cyber-Physical Systems. As our reliance on these devices is increasing, the significance of potential threats should not be underestimated, especially as a number of embedded devices are built to operate in malicious environments, where they might be in the possession of an attacker. The challenge to build secure and trusted embedded devices is paramount. In this paper, we examine the security threats to embedded devices along with the associated prevention mechanisms. We also present a holistic approach to the security and trust of embedded devices, from the hardware design, reliability and trust of the runtime environment to the integrity and trustworthiness of the executing applications. The proposed protection mechanisms provide a high degree of security at a minimal computational cost. Such an agnostic view on the security and trust of the embedded devices can be pivotal in their adoption and trust acquisition from the general public and service providers.

Keywords: Smart Cards, Fault Attacks, Runtime Attacks, Hardware Security, Runtime Security, Trusted Platform, Trusted Execution, Trojans, Counterfeit Products.

1 Introduction

Embedded devices provide a computing environment that is miniaturised to fit in as part of a much larger systems. For example, a smart phone, modern car and aircraft might have number of embedded devices interconnected with each other to perform associated tasks. The deployment of embedded devices is steadily increasing and the advent of the Internet of Things (IoTs) and Cyber Physical Systems (CPS) will make them closely integrated into almost every aspect of our lives.

These embedded devices must provide highly reliable and deterministic services – some of which might even be crucial to the health and safety of an individual or a community. Examples of such deployments can be the embedded devices used in the health sector, vehicles and industrial systems. Therefore, such devices have to not only provide efficient and reliable services in difficult operational environments but also provide a degree of security and trustworthiness. The level of security and trustworthiness

is obviously dictated by the nature of the overall system that the embedded devices are going to integrate with.

Having computational and in certain cases operational restrictions, embedded devices not only have to protect against software based attacks, but also hardware modifications and a combination of the both. Furthermore, such devices are mostly deployed in operational environments where they are easily accessible to the malicious users. Therefore, being small, less powerful and adhering to very stringent performance and economical costs — they are still required to be reliable, secure and trusted. These are the major challenges that embedded devices have to meet.

In addition to the security and reliability of such devices, another important aspect is the genuineness of the device. The genuineness problem has two aspects, which are raised due to the increased demand in outsourcing of the chip/device fabrication to (external/foreign) foundries. These foundries can inject hardware Trojans to the original design of the device. In addition, the foundry can also create counterfeit devices. Therefore, any organisations receiving these devices need to have a high assurance and capability to validate that these devices are legitimate and not tampered with. We term this is a problem of genuineness of the embedded devices.

In this paper, we will investigate the state-of-the-art of the threat and security in the embedded computing field along with examining proposals that cover the embedded device's security, trust and genuineness.

2 Embedded Computing's Security Challenges

In this section, we try to briefly answer two questions, firstly why security and reliability is essential for the embedded devices, and secondly what security threats are posed to embedded devices.

2.1 Rationale for Security Considerations

In the last few decades, embedded devices have proliferated into almost every computing and industrial system. A most common example of the embedded device with which most of the public might be familiar with is "smart cards". These devices are issues to individual users by organisations, so the individuals can access the organisation's services in a secure and reliable manner. Environments these smart cards are deployed in include mobile telecommunication, banking and access control, to name a few. We appreciate further that failure to the respective organisations services might result in potential monetary, reputation and even physical harm in certain cases.

Another example to understand the necessity of the security and reliability of embedded devices can be in automotive industry. Security of the Electronic Control Unit (ECU) plays a crucial role to ensure the safety and reliability of the car. These ECUs are embedded devices used to control different (crucial) operations in a car. In modern cars, there can be 70 ECUs [34] and if any of them can be compromised, the safety of the car and passengers may be at risk.

Embedded devices are tiny electronic chips that perform multitude of tasks in high-tech systems. These devices are present in electronic equipment ranging from microwave oven to high-speed railway systems, nuclear plants, and aeroplanes etc. A

failure of a single device has the potential to damage the overall system. Such failures can lead to disastrous consequences, as revealed by the U.S. Senate Committee on Armed Services that identified suspected components in the CH-46 Sea Knight helicopter, C-17 military transport aircraft, P-8A Poseidon sub hunter and F-16 fighter jet [10]. Multi-million dollar defence equipment reliability might be compromised by a $2 insecure and counterfeit embedded device. As estimated by the Semiconductor Industry Association (SIA) in 2013, the cost of counterfeit embedded devices is at US $7.5 billion per year [11]. The problem is by no means localised only affecting certain areas but a global issue. There are number of high profile cases [10] that came to light that identified sub-standard embedded devices in military equipment because of stringent safety testing. In commercial environments, the problem is perceived to be higher in magnitude than the military.

From the discussion in this section we can conclude that not only the security and reliability of these embedded devices is crucial but also the genuineness. In subsequent sections we will discuss different threats posed to the embedded systems.

2.2 Threat Model for Embedded Devices

In this section, we discuss the threat model in relation to the embedded devices. Embedded devices can be in the hands of an adversary; therefore, he or she has the potential to attack the devices in every conceivable way ranging from hardware intrusions to introducing malicious applications/code. From an adversary's point of view, he or she might target:

1. Hardware platform
2. Permanent data (saved in the devices, which can include cryptographic keys)
3. Runtime data
4. Control flow of the program (i.e. to interfere with the execution of an application)

The above list of potential targets is just a subset of attributes that an adversary might try to focus on during his or her attack. It is by no means an exhaustive list and should be taken as an example of potential targets.

2.3 Hardware Attacks

In these types of attacks, an adversary tries to alter the silicon design of an embedded device. This attack requires a high level of knowledge of the hardware design and specialised equipment that could be used to change the circuit on the silicon. However, these attacks are very powerful as it would be extremely challenging for any software based protection mechanism to provide protection against them.

Furthermore, another facet of the hardware attacks includes the malicious changes to chip design during the manufacturing stage. In this case, when the manufacturing of the devices is outsourced, the foundry can potentially introduce malicious designs that act as hardware Trojans.

As countermeasures to these attacks, the chip designer can include:

1. Smaller circuitry: Reducing the size makes physical attacks more difficult.

2. Hiding the bus: Glue logic and placing bus lines on lower layers of the circuitry of the chip.
3. Scramble bus lines: Communication buses can be scrambled in static, chip-specific or session-specific manner. The scrambling of the communication buses is carried out in order to make the function of individual silicon connections in a communication bus not apparent to the adversary (hidden).
4. Tamper-resistance: Placing sensors to detect physical perturbation and kill the device as a result.

2.4 Attacks on Persistent Storage

Data stored in persistent memory can include sensitive information, including passwords, Personal Identification Number (PIN), and cryptographic keys. In addition to data, proprietary application code and/or algorithm might also be stored on the persistent memory. Therefore, the persistent memory is like a treasure trove for an adversary. There are several potential ways an adversary can read the persistent storage that might include:

- Reading the memory via directly tapping into the storage locations and/or communication buses.
- Exploiting a potential bug/vulnerability in the sandboxing mechanism of the runtime environment, which might lead to a malicious application reading the entire persistent memory.
- Using side channel leakages to infer the data

As a security designer, to protect against potential attacks on the persistent storage, set of comprehensive security countermeasures are required that might include:

1. Encrypted storage/communication buses: To avoid data being read from storage or during transit, the data should be encrypted while in storage and over the communication bus using a hardware based key.
2. Memory read: Allowing only selected instruction in a given condition to access such data and then check their conditions.
3. Side channel protection: Implementing side channel protection techniques that hide the presence of data from the side channel footprint.

2.5 Attacks on Runtime Data

During the execution of an application, several data structures are generated that facilitate the execution. These might include intermediate computation results, function call parameters, return addresses and un/conditional statement parameters. These data structures might contain valuable information for an adversary to compromise the application. Modification to the runtime data can change the behavior of the application execution.

Modification to the runtime data is usually carried out by injecting a fault during the execution of the application. The aim of an adversary during a fault attack is to disrupt

the correct execution of an application by introducing errors. These errors are usually introduced by physical perturbation of the hardware platform on which the application is executing. By introducing errors at a precise instruction, an adversary can circumvent the security measures implemented by the runtime environment. Possible types of faults an adversary can produce are described as below:

1. Precise bit error: In this scenario, an adversary has total control over the timing and locations of bits that needs to be changed
2. Precise byte error: This scenario is similar to the previous one; however, an adversary only has the ability to change the value of a byte rather than a bit.
3. Unknown byte error: An adversary has no control on the timing and byte that it modifies during the execution of an instruction.
4. Unknown error: In this scenario, an adversary generates a fault but has no location and timing control.

From the above list of fault models, the first model adversary can be considered the most powerful. However, for a smart card environment the second scenario (i.e. precise byte error) is the most realistic one. Due to the advances in the smart card hardware and counter-measures against fault attacks (i.e. especially for cryptographic algorithms) it is difficult to have total control of timing and locations of bits to flip [47]. Furthermore, fault attacks require knowledge of the underlying platform and application execution pattern [29]. This is possible to achieve by side-channel analysis [37]. In most processors runtime data is processed as stack items; therefore their protection also works around the stack.

Countermeasures to the attacks on the runtime data include but are not limited to:

1. Stack canaries is a method where the processor inserts canary values into the stack and then check them during operation. If they are changed then there is an attack otherwise execution continues [28].
2. Separation of data and return address stack in this work the authors propose a segregation of the stack memory used for return addresses and other stack items. Then enforce instruction based access to the return addresses [32].
3. Verifying the integrity of an instruction before executing it. A trade-off between the security and the computational cost that the countermeasure adds.
4. Code signing: Verify it before loading it to the processor. However, this doesnt protect the program against runtime attacks. One of the solutions proposed to protect the instructions at runtime is to add an integrated module into the design that hashes the executed instructions and verifies their signature on the fly.

2.6 Notion of Trust and Trustworthiness

The definition of trust, taken from Merriam Webster's online dictionary[1] states that trust is a "belief that someone or something is reliable, good, honest, effective, etc."

Based on this, we generically define digital trust as "a trust based either on past experience or evidence that an entity has behaved and/or will behave in accordance

[1] Website: `http://www.merriam-webster.com/dictionary/trust`

with the self-stated behaviour." The self-stated purpose of intent is provided by the entity and this may have been verified/attested by a third party. The claim that the entity satisfies the self-stated behaviour can either be gained through past interactions (experience) or based on some (hard) evidence like validatable / verifiable properties certified by a reputable third party (i.e. Common Criteria evaluation for secure hardware [2]). This definition is not claimed to be a comprehensive definition for digital trust that encompasses all of its facets. However, this generic definition will be used as a point of discussion for the rest of the paper.

In the real world, trust in an entity is based on a feature, property or association that is entailed in it. In the computing world, establishing trust in a distributed environment also follows the same assumptions. The concept of trusted platforms is based on the existence of a trusted and reliable device that provides evidence of the state of a given system. How this evidence is interpreted is dependent on the requesting entity. Trust in this context can be defined as an expectation that the state of a system is as it is considered to be: secure. This definition requires a trusted and reliable entity called a Trusted Platform Module (TPM) to provide trustworthy evidence regarding the state of a system. Therefore, a TPM is a reporting agent (witness) not an evaluator or enforcer of the security policies. It provides a root of trust on which an inquisitor relies for the validation of the current state of a system.

The TPM specifications are maintained and developed by an international standards group called the Trusted Computing Group (TCG)[1] Today, TCG not only publishes the TPM specifications but also the Mobile Trusted Module (MTM), Trusted Multi-tenant Infrastructure, and Trusted Network Connect (TNC). With emerging technologies, service architectures, and computing platforms, TCG is adapting to the challenges presented by them.

2.7 Trust in Execution Environment

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

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 [49, 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] Trusted Computing Group (TCG) is the culmination of industrial efforts that included the Trusted Computing Platform Association (TCPA), Microsoft's Palladium, later called Next Generation Computing Base (NGSCB), and Intel's LaGrande. All of them proposed how to ascertain

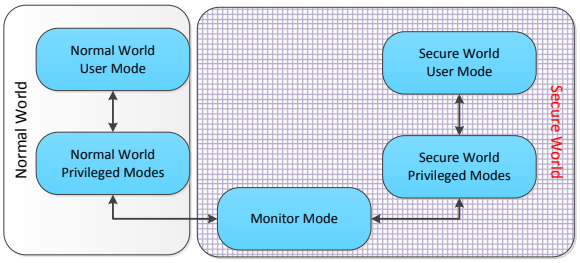

Fig. 1. Generic architectural view of ARM TrustZone

GlobalPlatform Trusted Execution Environment (TEE) The TEE is GlobalPlatform's initiative [4,6,9] 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 [3].

3 Trust in the Underlying Hardware

In the early days of computing systems security was almost virtually associated with the software. However, the commercial and economic conditions of late have forced hardware manufacturers to outsource their production process to countries with cheaper infrastructure cost. While this significantly reduces the integrated circuit production cost, it also makes it much easier for an attacker to compromise their supply chain and replace them with unoriginal or malicious ones. Such items could be counterfeits or hardware Trojans. This threat to the IC supply chain is already a cause for alarm in some countries [12, 40]. For this reason, some governments have been subsidising few high-host local foundries for producing ICs used in military applications [31]. However, this is not affordable solution for most of the developing countries.

Counterfeits Counterfeiting at a global stage covers almost everything that is made or manufactured, from spare parts to clothing to prescription drugs. In contrast to other counterfeit items, the ramifications of a counterfeit IC device failure in an electronic system are more than just inconvenience or a minor loss of money. According to [27], the number of counterfeit incidents has increased from 3,868 in 2005 to 9,356 in 2008. These incidents can have the following ramifications; (a) original IC providers incur an

trust in a device's state in a distributed environment. These efforts were combined in the TCG specification that resulted in the proposal of TPM.

irrecoverable loss due to the sale of often cheaper counterfeit components, (b) low performance of counterfeit 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; this could in turn harm the reputation of authentic providers, and (c) unreliability of defective devices could render the integrated systems that unknowingly use the parts unreliable; this potentially affects the performance of weapons, airplanes, cars or other crucial applications that use the fake components [38].

Hardware Trojans Hardware Trojans are malicious circuitry implanted in an IC. The malicious circuit can be inserted for different reasons, such as stealing sensitive information, IP reverse engineering or spying on the user. One way of implanting a Trojan into an IC is by compromising the supply chain of ICs and adding the Trojan mask into the original design. Trojan circuits are designed to be very difficult, nearly impossible, to detect by purely functional testing. They are designed to monitor for specific but rare trigger conditions; for instance specific bit patterns on received data or the bus. Once triggered the actions of the Trojan could be leaking secrets, creating glitches to compromise the security of larger electronic equipments or simply disabling the circuit. For example, a simple yet deadly Trojan in RSA [46] could be to inject a fault into the CRT inversion step during RSA signature computation that could lead to the compromise of the RSA keys [25].

3.1 Countermeasure

Counterfeit ICs and hardware Trojans could be designed to be hard to detect by purely functional testing. However, in the real world ICs leak information about their internal state unintentionally. This leakage comes as a power consumption or electromagnetic emissions caused by a varying electric current flowing through the IC's circuitry. This leakage can be recorded and analysed to adequately detect counterfeits and hardware Trojans. For instance in [48], a gate-level passive hardware characterisation of an IC was proposed to identify defective ICs. However, the gate-level characteristics are dependent on ageing, temperature and supply voltage instability. The authors use the negative bias temperature instability model proposed in [26] to calculate the original characteristics of aged ICs. In another proposal [13], power consumption of a device was proposed for detecting hardware Trojans implanted in ICs. In this paper process variation noise modelling (constructed using genuine ICs) is used for detecting ICs with Trojan circuits through statistical analysis. In this section we discuss how same leakage can be used to verify the integrity of control flow jumps and instructions integrity before the IC is integrated into a security critical environments.

We implemented these techniques on the ATMega164 processor. This processor has 130 instructions used for transferring data, performing arithmetic and logic operations. To simplify the experiment we removed redundant instructions. The processor is powered up by a +5V power supply and running at a 4MHz clock speed. Leroy WaveRunner 6100A [39] is used to measure the power traces.

Control Flow Verification An application is a combination of basic blocks. A basic block is a linear sequence of executable instructions with only one entry point and one

exit point [23]. After executing one basic block the processor jumps into another basic block based on the branching instruction executed at the end of the current basic block. In this paper we refer to basic blocks as states.

To reconstruct the state sequence that a device followed during the execution of a program from its side channel leakage we modelled the device as a *Hidden Markov Model (HMM)* [30, 44]. A *Markov Model* is a memoryless system with a finite number of hidden states. It is called memoryless because the next state depends only on the current state. In such a model the states are not directly observable. However, there has to be (at least) one observable output of the process that reveals partial information about the state sequence that the device has followed. Fig. 2, illustrates a *Markov Process* with five hidden states (i.e A to E).

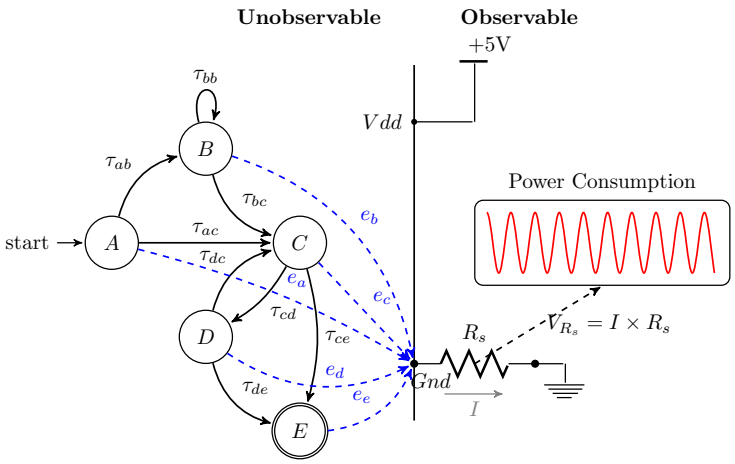

Fig. 2. A Markov model representing a device executing a program with five states (A, B, C, D and E). The power consumption is the observable output that reveals partial information about the state sequence of the device.

To build the HMM of our test program we collected 1000 traces for each state and computed all the necessary parameters. We have also pre-computed the possible valid control flow jumps of the program. At runtime we collected the power consumption of the program without any prior information which path the device followed to execute the program. From the trace we recovered the control flow jumps using the HMM and Viterbi algorithm [35]. We repeated this experiment multiple times and successfully verified the control flow jumps. Details of the technique and experiment results are presented in [42].

Verifying Integrity of Executed Instructions The first step in our verification is the construction of instruction-level side channel templates using few identical processors. During verification, the verifying device records the processor's power consumption

waveform while executing the application and extracts the executed instructions by matching it against the pre-constructed templates. The extracted information together with the pre-computed signatures are then used to verify the integrity of the software component using *RSA signature screening* algorithm [24].

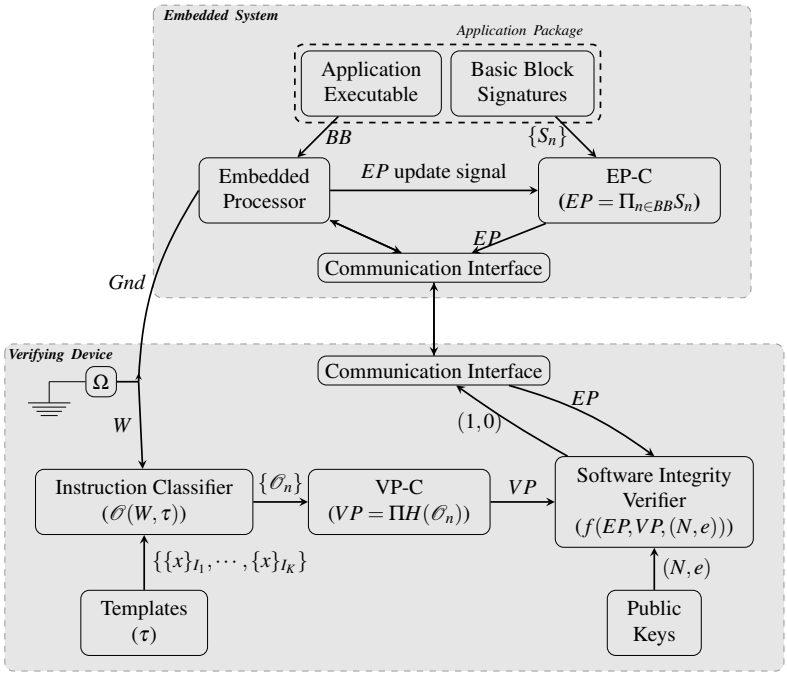

Fig. 3. Software integrity verification block diagram

As shown in the diagram (Fig. 3), the embedded system has the embedded parameter calculator (EP-C), embedded processor and the application package which includes the application executable and the basic block signatures. The EP-C is a special module that calculates the product of two large numbers. It can be implemented in hardware or software; although, hardware would be preferable for performance reasons. The embedded processor is the core that executes the software component of the embedded system (application executable). After the execution of every basic block the EP-C updates its parameter (EP) by multiplying it with the basic block's signature.

The verifying device has the templates, the instruction classifier, the verifier parameter calculator (VP-C) and the software integrity verifier. The templates are constructed ahead of time using identical processors and then installed into the verifying device's non-volatile memory. How these templates are installed into the verifying device is beyond the scope of this paper. The instruction classifier uses these templates to extract the executed instructions from the processor's power consumption waveform (W).

The power consumption waveform is measured as a voltage drop across a shunt resistor connecting the embedded processor's ground and the verifying device's ground voltage. The VP-C uses the output of the classifier to compute the verifying device's parameter. Finally, the software integrity verifier uses the output of the EP-C and VP-C to verify the software using *RSA signature screening* algorithm. Details of the template construction, instruction classification and software integrity verification processes are discussed [43].

The templates of selected instructions are created from 2500 traces collected by executing the instructions using different conditions; such as data processed, memory locations and registers. Finally, using these templates we successfully verified the integrity of executed instructions of a sample PIN verification program. Full detail of the verification techniques and experimental results are presented in [43].

4 Trusted Platform and Execution for Embedded Devices

The Trusted Computing Group is currently looking into the concept of trusted platform for embedded devices. Although, there is no specification made public at the time of writing this paper. However, we have proposed a similar trusted platform for smart cards and we will be discussing it in subsequent sections.

4.1 Trusted Environment & Execution Manager (TEM)

This section discusses the architecture of a Trusted Environment & Execution Manager (TEM) specifically for smart cards, and highlights how the TEM differs from a typical TPM not only in architectural but also in operational context.

4.2 Architecture

The TEM is illustrated as a layer between the smart card hardware and the runtime environment. This illustration provides a semantic view of the architecture and does not imply that all communication between the runtime environment and the hardware goes through the TEM.

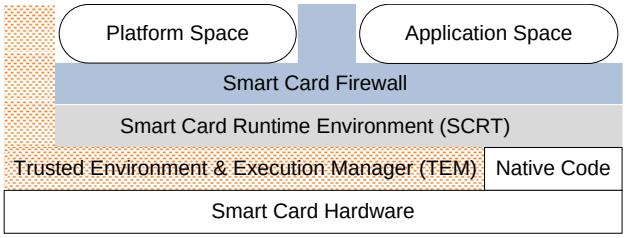

Fig. 4. Smart Card Architecture in with TEM

If general TPM requirements are analysed [5], the basic building blocks in the hardware required to build a TPM chip are already available on smart cards. Therefore, most

of the functionality of the TEM would be implemented in software and it would not impose any additional hardware requirement on the host platform. The detailed TEM architecture is shown in Figure 5.

Figure 5 depicts native code and smart card hardware as complementary components of the TEM. This is because the TEM does not need separate hardware for its operations. It will utilise the existing services provided by the smart card hardware. To avoid duplicating the code, the TEM uses the native code implementation of cryptographic services like encryption/decryption, digital signature and random number generation.

Interface The interface manages communication with external entities that can either be on-card or off-card entities. Any request that the interface receives is interpreted: if it is a well-formed request and the requesting entity is authorised to do so, then the interface will redirect the request to the intended module in the TEM. The interface during the interpretation of the request will enforce the access policy of the TEM as defined by the access control module (discussed in section 4.2). To manage these relationships with the authorised entities, the TEM should have a mechanism to establish the relationship in the first place. Therefore, at the time of installation of an application, a binding (symmetric key) is generated between the downloaded application and the TEM. For all subsequent communications, the application would use this key when requesting the TEM [16]. The protocol that establishes this binding is managed by the interface and the binding is stored in the key/certificate manager and corresponding access privilege in access control module.

Attestation Handler During the application installation process, both an application and a smart card platform would need to verify each other's current state to gain assurance of their trustworthiness. An application can only request attestation for either itself or the respective platform. It cannot request attestation for other applications on the smart card concerned. However, to facilitate the application sharing mechanism [14] an application can issue an authorisation token. The attestation handler will then provide the attestation of the token-issuing application to the requesting application [15].

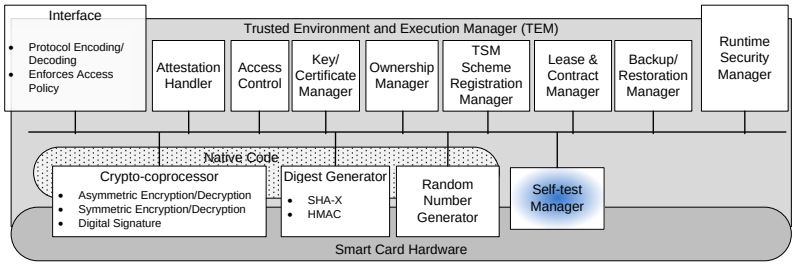

Fig. 5. Trusted Platform Module for Smart Card Architecture

Access Control At the time of application installation, the Service Provider (SP) involved would request attestation of the card platform. However, no information regarding any of the other applications installed on the card would be provided to the SP at this stage. Once the application is installed, it can request attestation only for itself and not for any other applications. These restrictions are required to avoid privacy issues like application scanning attacks [15].

Key & Certificate Manager The key & certificate manager manages the keys and certificates that a TEM stores in the non-volatile memory (EEPROM [45]). Contrary to the general TPM architecture, there are no migratable keys in the TEM. The TEM signature key pair and certificate is the permanent key and certificate (it can be considered as the endorsement key in the general TPM architecture). Besides managing the keys and certificates, it also generates them. Therefore, it is a combination of key generation and non-volatile memory components of the general TPM.

The key & certificate manager stores the evaluation certificates which are provided by the respective applications. Therefore, when an application requests attestation, the TEM does not return the hash value of the application. In fact, it returns an evaluation of whether the current state complies with the state for which the evaluation certificate was issued. Therefore, the decision whether an application is trustworthy or not is actually made by the TEM. If the evaluation fails, then depending upon the application or platform policy it might either block the application or delete it (and inform the cardholder and respective SP).

Ownership Manager This component manages the ownership of a smart card. When a smart card is acquired by a user either from a card manufacturer or a card supplier, it is under the default ownership of the card manufacturer/supplier. The user then initiates the ownership acquisition process that requires the user to provide personal information (i.e. name and date of birth) and their Card Management Personal Identification Number (CM-PIN). The TEM will then generate a signature key pair specific to the cardholder along with a certificate that will also include the user information. Although this key is assigned to the cardholder, it will be protected by the TEM.

TSM Scheme Registration Manager This module is optional and it facilitates Competitive Architecture for Smart Cards. For further details please refer to [17].

Lease & Contract Manager An SP would lease its application to a smart card (cardholder) and the card would assure that it would abide by the SP's Application Lease Policy (ALP) The lease contract is signed by the TEM with the user's signature key and as these keys are stored/restrict access only to the TEM, the signing and storage of the contracts are on the TEM. The cardholder can retrieve these contracts after providing the CM-PIN if he/she needs to. Similarly, individual applications can also retrieve their own contracts from the TEM repository.

Backup/Restoration Manager A cardholder may download multiple applications onto her smart card. If she loses her smart card, she will lose access to all of the applications (and related services). One possible approach can be to acquire a new card and then manually install all the applications again. However, another approach could be that a user creates a backup of the installed applications and restores the backup to a new smart card, if required. This backup mechanism is credential-based (a token issued by the SPs and not the actual application) and it is stored securely at a remote location [20]. When users lose their smart cards, they only need to get a new smart card and then initiate the restoration process, which will take each credential from the backup and initiate the application download process with respective SPs. The restoration process can also request the respective SPs to block (revoke the lease) their application(s) installed on the stolen/lost device.

Self-test Manager For security validation, the TEM implements a validation mechanism that is divided into two parts: tamper-evidence and reliability assurance. Smart cards are required to be tamper-resistant devices [45] and for this purpose card manufacturers implement hardware-based tamper protections. The tamper-evidence process verifies whether the implemented tamper-resistant mechanisms are still in place and effective. The reliability assurance process, on the other hand, verifies that the software part of the smart card that is crucial for its security and reliability has not been tampered with.

A TEM tamper-evidence process should provide the properties listed below:

1. Robustness: On input of certain data, it always produces the associated output.
2. Independence: When the same data is input to a tamper-evidence process on two different devices, it outputs different values.
3. Pseudo-randomness: The generated output should be computationally difficult to distinguish from a pseudo-random function.
4. Tamper-evidence: An invasive attack to access the function should cause irreversible changes, which render the device unusable.
5. Assurance: The function can provide assurance (either implicitly or explicitly) to independent (non-related) verifiers. It should not require an active connection with the device manufacturer to provide the assurance. The assurance refers to the current hardware and software state as it was at the time of third party evaluation.

For the TEM tamper-evidence process there are several candidates including: active (intelligent) shield/mesh [45]; Known Answer Test (KAT) [1], hard-wired HMAC key, attestation based on PRNG [36]; and Physically Unclonable Function (PUF) [33]. Two algorithms that provide tamper-evidence and reliability based upon PUF and PRNG based validation mechanisms are discussed in [22] and [21], respectively.

Runtime Security Manager The purpose of the runtime security manager is to enforce the security counter-measures defined by the respective platform. To enforce the security counter-measures, the runtime security manager has access to the heap area (e.g. method area, Java stacks) and can be implemented as either a serial or a parallel mode.

A serial runtime security manager will rely on the execution engine of the Java Card Virtual Machine (JCVM) [8] to perform the required tasks. This means that when an execution engine encounters instructions that require an enforcement of the security policy, it will invoke the runtime security manager that will then perform the checks. If successful the execution engine continues with execution, otherwise, it will terminate. A parallel runtime security manager will have its own dedicated hardware (i.e. processor) support that enables it to perform checks simultaneously while the execution engine is executing an application. Having multiple processors on a smart card is technically possible [45]. The main question regarding the choice is not the hardware, but the balance between performance and latency.

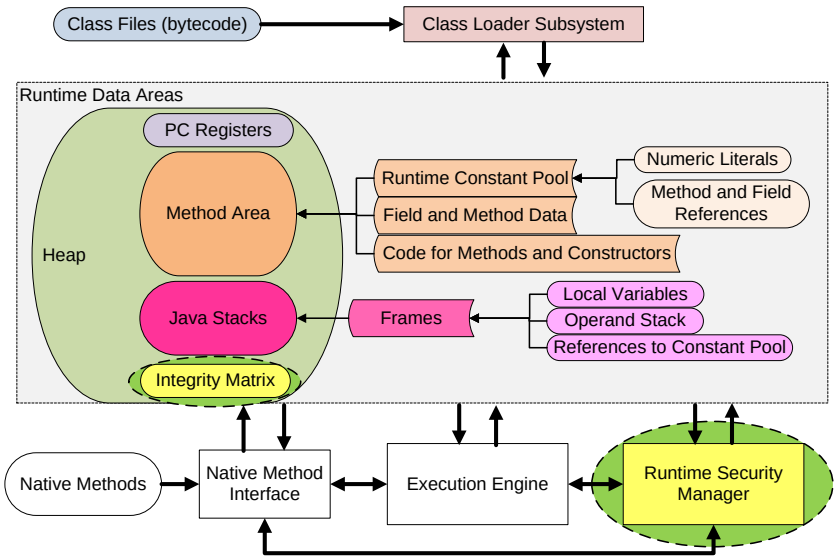

Fig. 6. Runtime Security Manager Integration with Java Card Runtime Environment

Performance, as the name suggests, is concerned with computational speed, whereas latency deals with the number of instructions executed between an injected error and the point at which it is detected. For example, if during the execution of an application 'A', at instruction A4 a malicious user injects an error, which is detected by the platform security mechanism at instruction A7 of the application, the latency is three (i.e. 7-4=3). A point to note is that the lower the latency value the better the protection mechanism, as it will catch the error quickly. Therefore, theoretically we can assume that a serial runtime security manager will have low performance but also a low latency value, while a parallel runtime security manager will have a good performance measure but a higher latency value.

It is obvious that the implementation of additional components like runtime security managers will also incur additional economic costs (i.e. increase in the price of a smart card. The security measures that could be enforced are:

1. Operand Stack Integrity: In this we XOR the value pushed on to the operand stack with the top value of the integrity stack and the results are pushed back on to the integrity stack. When a value is popped from the operand stack, we will XOR the popped value from the top value in the integrity stack (where n is the top value of the stack). If the result is same as the value n-1 in the integrity stack, the execution continues, if not, then it is interrupted by the runtime security manager.
2. Control Flow Analysis: Authorised execution flows are generated oncard at the time of application installation. Later when application is executing, only the authorised execution flow is allowed to go ahead. Any violation would render the application blocked and may lead of it being deleted from the device.
3. Bytecode Integrity: Each basic block of an application code has an associated integrity value. When the basic block is fetched to the runtime memory, the integrity value is verified.

Preliminary Results For evaluation of proposed counter-measures, we have selected four sample applications. Two of the applications selected are part of the Java Card development kit distribution: Wallet and Java Purse. The other two applications are the implementation of our proposed mechanisms that include the offline attestation algorithm [21] and STCP_{SP} protocol [18].

4.3 Latency Analysis

As discussed before, latency is the number of instructions executed after an adversary mounts an attack and the system becomes aware of it. Therefore, in this section we analyse the latency of proposed counter-measures under the concept of serial and parallel runtime security managers that are listed in Table 1 and discussed subsequently.

Table 1. Latency measurement of individual countermeasure

Counter-measures	**Serial RSM**	**Parallel RSM**
Operand Stack Integrity	0 + i	3 + i
Permitted Execution Path Analysis	0	$3(C_n)$
Bytecode Integrity	0	0

In case of the operand stack integrity, the serial runtime security manager finds the occurrence of an error (e.g. fault injection) with latency "0+i", where 'i' is the number of instructions executed before the manipulated value reaches the top of the operand stack. For example, consider an operand stack with values V_1, V_2, V_3, V_4, and V_5, where V_5 is the value on the top. If an adversary changes the value of V_3 by physical perturbation, then the runtime security manager will not find out about his change until the value is popped out of the stack. Therefore, the value of 'i' depends upon the number of instructions that will execute until the V_3 reaches the top of the operand stack and JCVM pops it out. Similarly, the latency value in case of the operand stack integrity for the parallel runtime security manager is "3+i", where '3' is the number of instructions required to perform a comparison on a pop operation. The latency value of the parallel

runtime security manager is higher than the serial. This has to do with the fact that while parallel runtime security manager is applying the security checks the JCVM does not need to stop the execution of subsequent instructions.

Regarding the control flow analysis, the serial runtime security manager has a latency of zero where the parallel runtime security manager has latency value of "3(C_n)", where the value C_n represents the number of legal jumps in the respective execution flow set. The value '3' represents the number of instructions required to execute individual comparison.

A notable point to mention here is that all latency measurements listed in the Table 2 are based on the worst-case conditions. Furthermore, it is apparent that it might be difficult to implement a complete parallel runtime security manager. To explain our point, consider two consecutive jump instructions in which the parallel runtime security manager has to perform control flow analysis. In such situation, there might be a possibility that while the runtime security manager is still evaluating the first jump, the JCVM might initiate the second jump instruction. Therefore, this might create a deadlock between the JCVM and parallel runtime security manager - we consider that either JCVM should wait for the runtime security manager to complete the verification, or for the sake of performance the runtime security manager might skip certain verifications. We opt for the parallel runtime security manager that will switch to the serial runtime security manager mode - restricting the JCVM to proceed with next instruction until the runtime security manager can apply the security checks. This situation will be further explained during the discussion on the performance measurements in the next section.

4.4 Performance Analysis

To evaluate the performance impact of the proposed counter-measures we developed an abstract virtual machine that takes the bytecode of each Java Card applet and then computes the computational overhead for individual countermeasure. When a Java application is compiled the java compiler (`javac`) produces a class file. The class file is Java bytecode representation, and there are two possible ways to read class files. We can either use a hex editor (an editor that shows a file in hexadecimal format) to read the Java bytecodes or better utilise the `javap` tool that comes with Java Development Kit (JDK). In our practical implementation, we opted for `javap` as it produces the bytecode representation of a class file in human-readable mnemonics as represented in the JVM specification [41]. We used `javap` to produce the mnemonic bytecode representation; the abstract virtual machine takes the mnemonic bytecode representation of an application and searches for push, pop, and jump (e.g. method invokes) opcodes. Subsequently, we calculated the number of extra instructions required to be executed in order to implement the counter-measures discussed in previous sections.

To compute the performance overhead, we counted the number of instructions an application has and how long the application takes to execute on our test Java Cards (e.g. C1 and C3). After this measurement, we have associated costs based on additional instructions executed for each JCVM instruction and calculated as an (approximate) increase in the percentage of computational overhead and listed in Table 2.

For each application, the counter-measures have different computational overhead values because they depend upon how many times certain instructions that invoke the

Table 2. Performance measurement (percentage increase in computational cost)

Applications	Serial RSM	Parallel RSM
Wallet	+29%	+22%
Java Purse	+30%	+26%
Offline Attestation [22]	+27%	+23%
$STCP_{SP}$ [19]	+39%	+33%

counter-measures are executed. Therefore, the computational overhead measurements in Table 2 can only give us a measure of how the performance is affected in individual cases - without generalising for other applications.

In this section we discussed the smart card runtime environment by taking the Java Card as a running example. The JCRE was described with its different data structures that it uses during the execution of an application. Subsequently, we discussed various attacks that target the smart card runtime environment and most of these attacks based on perturbation of the values stored by the runtime environment. These perturbations are called fault injection, which was defined and mapped to an adversary's capability in this chapter. Based on these recent attacks on the smart card runtime environment, we proposed an architecture that includes the provision of a runtime security manager. We also proposed various counter-measures and provided the computational cost imposed by these counter-measures. No doubt, counter-measures that do not change the core architecture of the Java virtual machine, will almost always incur extra computational cost. Therefore, we concluded in this chapter that a better way forward would be to change the architecture of the Java virtual machine. However, in the context of this paper we showed that current architecture can be hardened at the cost of a computational penalty.

5 Conclusions

In this paper, we have briefly highlighted the security, trust and genuineness requirements for the embedded devices. These devices are becoming ever present in our daily life and reliance on them is going to increase in coming years. Therefore, utmost efforts have to be invested into their security and reliability in order to provide a safe and efficient service to the users. In this paper, we discussed some of the threat vectors that an adversary can use to compromise these devices. Furthermore, we also discussed the associated countermeasures along with some state-of-the-art protection mechanism. In this paper, we have also detailed a few of our proposal to provide security, reliability, trust and genuineness. The field of embedded computing still faces a number of challenges, effectively making it an exciting domain for security research to investigate and be innovative.

References

1. FIPS 140-2: Security Requirements for Cryptographic Modules. Online, May 2005.

2. Common Criteria for Information Technology Security Evaluation, Part 1: Introduction and General Model, Part 2: Security Functional Requirements, Part 3: Security Assurance Requirements,, August 2006.
3. GlobalPlatform: GlobalPlatform Card Specification, Version 2.2,, March 2006.
4. . GlobalPlatform Device: GPD/STIP Specification Overview. Specification Version 2.3, GlobalPlatform, August 2007.
5. Trusted Computing Group, TCG Specification Architecture Overview. revision 1.4, The Trusted Computing Group (TCG), Beaverton, Oregon, USA, August 2007.
6. GlobalPlatform Device Technology: Device Application Security Management - Concepts and Description Document Specification. Online, April 2008.
7. . ARM Security Technology: Building a Secure System using TrustZone Technology. White Paper PRD29-GENC-009492C, ARM, 2009.
8. Java Card Platform Specification: Classic Edition; Application Programming Interface, Runtime Environment Specification, Virtual Machine Specification, Connected Edition; Runtime Environment Specification, Java Servlet Specification, Application Programming Interface, Virtual Machine Specification, Sample Structure of Application Modules, May 2009.
9. GlobalPlatform Device Technology: TEE System Architecture. Specification Version 0.4, GlobalPlatform, October 2011.
10. Inquiry into Counterfeit Electronic Parts in the Department of Defense Supply Chain. Online, September 2012.
11. Winning the Battle Against Counterfeit Semiconductor Products. Online, August 2013.
12. Defense Advanced Research Projects Agency. Darpa baa06-40, a trust for integrated circuits, Visited, May 2013. `https://www.fbo.gov/index?s=opportunity&mode=form&id=db4ea611cad3764814b6937fcab2180a&tab=core&_cview=1`.
13. D. Agrawal, S. Baktir, D. Karakoyunlu, P. Rohatgi, and B. Sunar. Trojan detection using ic fingerprinting. In *Security and Privacy, 2007. SP '07. IEEE Symposium on*, pages 296–310, 2007.
14. Raja Naeem Akram, Konstantinos Markantonakis, and Keith Mayes. Firewall Mechanism in a User Centric Smart Card Ownership Model. In Dieter Gollmann, Jean-Louis Lanet, and Julien Iguchi-Cartigny, editors, *Smart Card Research and Advanced Application, 9th IFIP WG 8.8/11.2 International Conference, CARDIS 2010*, volume 6035/2010 of *LNCS*, pages 118–132, Passau, Germany, April 2010. Springer.
15. Raja Naeem Akram, Konstantinos Markantonakis, and Keith Mayes. Application-Binding Protocol in the User Centric Smart Card Ownership Model. In Udaya Parampalli and Phillip Hawkes, editors, *the 16th Australasian Conference on Information Security and Privacy (ACISP)*, LNCS, pages 208–225, Melbourne, Australia, July 2011. Springer.
16. Raja Naeem Akram, Konstantinos Markantonakis, and Keith Mayes. Cross-Platform Application Sharing Mechanism. In Huaimin Wang, Stephen R. Tate, and Yang Xiang, editors, *10th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (IEEE TrustCom-11)*, Changsha, China, November 2011. IEEE Computer Society.
17. Raja Naeem Akram, Konstantinos Markantonakis, and Keith Mayes. Building the Bridges – A Proposal for Merging different Paradigms in Mobile NFC Ecosystem. In Shengli Xie, editor, *The 8th International Conference on Computational Intelligence and Security (CIS 2012)*, Guangzhou, China, November 2012. IEEE Computer Society.
18. Raja Naeem Akram, Konstantinos Markantonakis, and Keith Mayes. A Secure and Trusted Channel Protocol for the User Centric Smart Card Ownership Model. In *12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (IEEE TrustCom-13)*, Melbourne, Australia, 2013. IEEE Computer Society.

19. Raja Naeem Akram, Konstantinos Markantonakis, and Keith Mayes. A Secure and Trusted Channel Protocol for the User Centric Smart Card Ownership Model. In *12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (IEEE TrustCom-13)*, Melbourne, Australia, July 2013. IEEE Computer Society.
20. Raja Naeem Akram, Konstantinos Markantonakis, and Keith Mayes. Recovering from Lost Digital Wallet. In F. G. Marmol Y. Xiang and S. Ruj, editors, *The 4th IEEE International Symposium on Trust, Security, and Privacy for Emerging Applications (TSP-13)*, Zhangjiajie, China, November 2013. IEEE CS.
21. Raja Naeem Akram, Konstantinos Markantonakis, and Keith Mayes. Remote Attestation Mechanism based on Physical Unclonable Functions. In C. Ma J. Zhou and J. Weng, editors, *The 2013 Workshop on RFID and IoT Security (RFIDsec'13 Asia)*, Guangzhou, China, November 2013. IOS Press.
22. Raja Naeem Akram, Konstantinos Markantonakis, and Keith Mayes. Remote Attestation Mechanism for User Centric Smart Cards using Pseudorandom Number Generators. In S. Qing and J. Zhou, editors, *5th International Conference on Information and Communications Security (ICICS 2013)*, Beijing, China, November 2013. Springer.
23. Frances Allen. Control flow analysis. In *Proceedings of a symposium on Compiler optimization*, pages 1–19, New York, NY, USA, July 1970. ACM.
24. Mihir Bellare, Juan A. Garay, and Tal Rabin. Fast batch verification for modular exponentiation and digital signatures. In Kaisa Nyberg, editor, *EUROCRYPT*, volume 1403 of *Lecture Notes in Computer Science*, pages 236–250, Espoo, Finland, May 31 - June 4 1998. Springer.
25. Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. On the importance of checking cryptographic protocols for faults (extended abstract). In *Advances in Cryptology - EUROCRYPT '97, International Conference on the Theory and Application of Cryptographic Techniques, Konstanz, Germany, May 11-15, 1997, Proceeding*, pages 37–51, 1997.
26. S. Chakravarthi, A.T. Krishnan, V. Reddy, C.F. Machala, and S. Krishnan. A comprehensive framework for predictive modeling of negative bias temperature instability. In *Reliability Physics Symposium Proceedings, 2004. 42nd Annual. 2004 IEEE International*, pages 273–282, 2004.
27. U.S. Department Of Commerce. Defense industrial base assessment: Counterfeit electronics. Technical report, Bureau of Industry and Security, Office of Technology Evaluation, January, 2010. `http://www.bis.doc.gov/defenseindustrialbaseprograms/osies/defmarketresearchrpts/final_counterfeit_electronics_report.pdf`.
28. Crispin Cowan, Calton Pu, Dave Maier, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th Conference on USENIX Security Symposium - Volume 7*, SSYM'98, pages 5–5, Berkeley, CA, USA, 1998. USENIX Association.
29. Marc Éluard and Thomas Jensen. Secure Object Flow Analysis for Java Card. In *CARDIS'02: Proceedings of the 5th conference on Smart Card Research and Advanced Application Conference*, pages 11–11, Berkeley, CA, USA, 2002. USENIX Association.
30. A. Fink. *Markov Models for Pattern Recognition*. Springer, 2008.
31. Defense Science Board Task Force. High performance microchip supply, Visited, May 2013. `http://www.acq.osd.mil/dsb/reports/ADA435563.pdf`.
32. Mike Frantzen and Mike Shuey. Stackghost: Hardware facilitated stack protection. In *Proceedings of the 10th Conference on USENIX Security Symposium - Volume 10*, SSYM'01, Berkeley, CA, USA, 2001. USENIX Association.
33. Blaise Gassend, Dwaine Clarke, Marten van Dijk, and Srinivas Devadas. Silicon Physical Random Functions. In *Proceedings of the 9th ACM conference on Computer and communications security*, CCS '02, pages 148–160, New York, NY, USA, 2002. ACM.

34. Olaf Henniger, Ludovic Apvrille, Andreas Fuchs, Yves Roudier, Alastair Ruddle, and Benjamin Weyl. Security requirements for automotive on-board networks. In *Intelligent Transport Systems Telecommunications,(ITST), 2009 9th International Conference on*, pages 641–646. IEEE, 2009.
35. David Forney Jr. The Viterbi Algorithm: A personal history. *CoRR*, abs/cs/0504020, 2005.
36. Rick Kennell and Leah H. Jamieson. Establishing the genuinity of remote computer systems. In *Proceedings of the 12th conference on USENIX Security Symposium - Volume 12*, pages 21–21, Berkeley, CA, USA, 2003. USENIX Association.
37. Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential Power Analysis. In *CRYPTO '99: Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology*, pages 388–397, London, UK, 1999. Springer-Verlag.
38. F. Koushanfar, A.-R. Sadeghi, and H. Seudie. EDA for secure and dependable cybercars: Challenges and opportunities. In *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, pages 220–228, 2012.
39. Teledyne LeCroy. Teledyne LeCroy website, Visited February 2013. `http://www.teledynelecroy.com`.
40. Joseph I. Lieberman. The national security aspects of the global migration of the u.s. semiconductor industry, Visited, May 2013. `http://www.fas.org/irp/congress/2003_cr/s060503.html`.
41. Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley Longman, Amsterdam, 2 edition, April 1999.
42. Mehari Msgna, Konstantinos Markantonakis, and Keith Mayes. The b-side of side channel leakage: Control flow security in embedded systems. In *Security and Privacy in Communication Networks - 9th International ICST Conference, SecureComm 2013, Sydney, NSW, Australia, September 25-28, 2013, Revised Selected Papers*, pages 288–304, 2013.
43. Mehari Msgna, Konstantinos Markantonakis, David Naccache, and Keith Mayes. Verifying software integrity in embedded systems: A side channel approach. In *Constructive Side-Channel Analysis and Secure Design - 5th International Workshop, COSADE 2014, Paris, France, April 13-15, 2014. Revised Selected Papers*, pages 261–280, 2014.
44. Lawerence Rabiner. A tutorial on Hidden Markov Models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, Feb 1989.
45. W. Rankl and W. Effing. *Smart Card Handbook*. John Wiley & Sons, Inc., New York, NY, USA, third edition, 2003.
46. Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, 1978.
47. Ahmadou A. Sere, Julien Iguchi-Cartigny, and Jean-Louis Lanet. Automatic Detection of Fault Attack and Countermeasures. In *Proceedings of the 4th Workshop on Embedded Systems Security*, WESS '09, pages 71–77, New York, NY, USA, 2009. ACM.
48. Sheng Wei, Ani Nahapetian, and Miodrag Potkonjak. Robust passive hardware metering. In *International Conference on Computer-Aided Design (ICCAD)*, pages 802–809. IEEE, November 7-10 2011.
49. Peter Wilson, Alexandre Frey, Tom Mihm, Danny Kershaw, and Tiago Alves. Implementing Embedded Security on Dual-Virtual-CPU Systems. *IEEE Design and Test of Computers*, 24:582–591, 2007.