# Design and Analysis of Trusted Computing Platforms

**Dries SCHELLEKENS**

Jury:
Prof. dr. ir. Hugo Hens, chair
Prof. dr. ir. Bart Preneel, supervisor
Prof. dr. ir. Frank Piessens
Prof. dr. ir. Joos Vandewalle
Prof. dr. ir. Ingrid Verbauwhede
Prof. dr. techn. Dieter Gollmann
(Technische Universität Hamburg-Harburg)
Prof. dr.-ing. Ahmad-Reza Sadeghi
(Technische Universität Darmstadt)

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor in Engineering

December 2012

D/2013/7515/4
ISBN 978-94-6018-617-2

# Acknowledgements

I would like to take this opportunity to thank everyone who has supported me during my journey towards this thesis.

First and foremost, I would like to express my gratitude to my promotor Prof. Bart Preneel for giving me the opportunity to do research and pursue a Ph.D. at COSIC, for his guidance and advice, and for carefully reading and correcting this dissertation. I wish to thank Prof. Frank Piessens, Prof. Joos Vandewalle, Prof. Ingrid Verbauwhede, Prof. Dieter Gollmann and Prof. Ahmad-Reza Sadeghi for being members of my jury and Prof. Hugo Hens for chairing it.

I am indebted to my co-authors and the many people I worked with in different projects. In particular I am grateful to Klaus Kursawe and Pim Tulys for the close collaboration during their days at COSIC.

COSIC has been a stimulating working environment and many of my colleagues became friends. Hence, I would like to thank the past and current COSIC members, including An, András, Andreas, Anthony, Antoon, Bartek, Benedikt, Berna, Christophe, Christopher, Claudia, Danny, Dave, Elke, Elmar, Fré, Filipe, Jasper, Jens, Joe, Joris, Junfeng, Gauthier, George, Gregory, Kazuo, Klaus, Koen, Markulf, Markus, Miroslav, Nele, Nessim, Norbert, Orr, Robert, Roel, Roel, Pim, Saartje, Sebastian, Sebastiaan, Stefaan, Stefan, Svetla, Taizo, Thomas, Yoni, and Wim. In particular, I would like to thank Brecht Wyseur, Jan Cappaert, Karel Wouters and Yoni De Mulder for sharing different offices in the ESAT building.

A special thank you also goes to Péla Noë for being the best secretary in the world and much more and for sharing my passion for dogs.

Ik wil ook graag mijn familie en vrienden bedanken voor hun onvoorwaardelijke steun. De laatste maar belangrijkste persoon die ik wil bedanken, is mijn vriendin Sofie voor haar liefde, geduld en steun, en voor de tijd die ze mij gegund heeft om deze thesis te voltooien. Bedankt voor alles!

Dries Schellekens
December 2012

# Abstract

This thesis deals with the analysis and design of trusted computing platforms. Trusted computing technology is a relatively new enabling technology to improve the trustworthiness of computing platforms. With minor changes to the boot process and the addition of a new hardware security component, called TPM (Trusted Platform Module), trusted computing platforms offer the possibility to verifiably report their integrity to external parties (i.e., *remote attestation*) and to bind information to a specific platform (i.e., *sealed storage*).

The first part of this thesis mainly focuses on the analysis of existing trusted computing platforms. We analyze the functionality provided by the specifications of the TCG (Trusted Computing Group) and purely software-based alternatives. Based on this analysis we present an improvement to a software-based attestation scheme: we propose to measure the execution time of a memory checksum function locally (with the time stamping functionality of the TPM) instead of remotely (over the network).

We also study the resilience of trusted computing platforms against hardware attacks. We describe how attacks on the communication interface of the TPM can circumvent the measured boot process. The feasibility of these attacks is investigated in practice. Additionally we explore which operations should be targeted with a side channel attack to extracts the secret keys of a TPM.

The second part of this thesis addresses some of the challenges to implement trusted computing technology on embedded and reconfigurable devices. One of the main problems when integrating a TPM into a system-on-chip design, is the lack of on-chip reprogrammable non-volatile memory. We develop schemes to securely externalize the non-volatile storage of a TPM. One scheme relies a new security primitive, called a reconfigurable physical unclonable function, and another extends the security perimeter of the TPM to the external memory with a cryptographic protocol.

We propose a new architecture to reset the trust boundary to a much smaller scale, thus allowing for simpler and more flexible TPM implementations. The architecture has two distinctive features: the program code is stored outside the coprocessor and only gets loaded in RAM memory when needed, and the architecture is open by allowing to execute arbitrary programs in remotely verifiable manner.

Finally, we study how the TPM can be implemented securely on reconfigurable hardware. This type of implementation is beneficial because it allows for updates of the software as well as of the hardware of the TPM (e.g., the cryptographic coprocessor) in the field. We examine the implementation options on reconfigurable hardware that is currently available commercially. Next, we propose a novel architecture that can measure and report the integrity of configuration bitstreams.

# Samenvatting

Dit proefschrift handelt over de analyse en het ontwerp van vertrouwde computerplatformen. Vertrouwde computerplatformen zijn een relatief nieuwe technologie die de betrouwbaarheid van computersystemen kan verbeteren. Door kleine wijzigingen aan het opstartproces en de toevoeging van een nieuwe hardwarebeveiligingscomponent, TPM (Trusted Platform Module) genoemd, maken vertrouwde computerplatformen het mogelijk om hun integriteit op een verifieerbare manier te rapporteren aan externe partijen (dit is *attestatie op afstand*) en om informatie te koppelen aan een specifiek platform (dit is *verzegelde opslag*).

Het eerste deel van dit proefschrift concentreert zich voornamelijk op de analyse van bestaande vertrouwde computersystemen. We bekijken de functionaliteit die aangeboden wordt door de specificaties van de TCG (Trusted Computing Group) en door puur software-gebaseerde alternatieven. Op basis van deze analyse stellen we een verbetering voor aan een schema voor software-gebaseerde attestatie. Hierbij wordt de uitvoeringstijd van de functie die een checksum over het geheugen berekent, lokaal gemeten (met behulp van de tijdszegelfunctionaliteit van de TPM) in plaats van de meting op afstand (over het netwerk) uit te voeren.

We bestuderen ook in welke mate vertrouwde computerplatformen bescherming bieden tegen hardware-aanvallen. We beschrijven hoe aanvallen op de communicatie-interface van de TPM het geauthentiseerde opstartproces kunnen omzeilen. De praktische haalbaarheid van deze aanvallen wordt onderzocht. Bovendien onderzoeken we op welke bewerkingen nevenkanaalaanvallen zich moeten richten om de geheime sleutels van een TPM te achterhalen.

Het tweede deel van dit proefschrift pakt enkele uitdagingen aan die zich stellen wanneer de technologie van vertrouwde computerplatformen toegepast wordt op ingebedde en herconfigureerbare toestellen. Een van de pijnpunten van de integratie van een TPM in een systeem-op-chip-ontwerp, is het feit dat intern niet-vluchtig geheugen onvoldoende voor handen is. We ontwerpen schema's om de niet-vluchtige opslag van een TPM op een beveiligde manier extern te maken. Eén schema steunt op een nieuw beveiligingsprimitief, dat een herconfigureerbare fysisch onkloonbare functie genoemd wordt. Een andere oplossing breidt de beveiligingsperimeter van de TPM uit naar het extern geheugen met een cryptografisch protocol.

We stellen een nieuwe architectuur voor die de vertrouwensgrens terugzet naar een veel kleinere schaal en die aldus meer eenvoudige en meer flexibele TPM-implementaties toelaat. De architectuur heeft twee onderscheidende kenmerken: de programmacode wordt buiten de coprocessor opgeslagen en enkel in het RAM-geheugen geladen wanneer nodig en de architectuur is open door de uitvoering van willekeurige programma's toe te laten zodat dit op afstand geverifieerd kan worden.

Ten slotte bestuderen we hoe de TPMs op veilige wijze geïmplementeerd kunnen worden op herconfigureerbare hardware. Zo'n implementatie is voordelig omdat het toelaat om zowel de software als hardware van de TPM (bv. de cryptografische coprocessor) bij te werken. We onderzoeken de mogelijkheden tot implementatie op herconfigureerbare hardware die momenteel commercieel beschikbaar is. Daarna stellen we een nieuwe architectuur voor die de integriteit van configuratiebestanden kan meten en rapporteren.

# Contents

# List of Figures

# List of Tables

# List of Acronyms

| | |
|---|---|
| **3DES** | Triple DES |
| **ACPI** | Advanced Configuration Power Interface |
| **AE** | Authenticated Encryption |
| **AES** | Advanced Encryption Standard |
| **AID** | Application Identifier |
| **AIK** | Attestation Identity Key |
| **AMT** | Active Management Technology |
| **API** | Application Programming Interface |
| **ASIC** | Application Specific Integrated Circuit |
| **BGA** | Ball Grid Array |
| **BIOS** | Basic Input/Output System |
| **BTE** | Bitstream Trust Engine |
| **CA** | Certification Authority |
| **CBC** | Cipher Block Chaining |
| **CC** | Common Criteria |
| **CCA** | Common Cryptographic Architecture |
| **CCM** | Counter with CBC-MAC |
| **CE** | Consumer Electronics |
| **CMOS** | Complementary Metal Oxide Semiconductor |

| | |
|---|---|
| **CPU** | Central Processing Unit |
| **CRC** | Cyclic Redundancy Check |
| **CRTM** | Core Root of Trust for Measurement |
| **CTR** | Counter |
| **CWC** | Carter-Wegman + Counter |
| **DAA** | Direct Anonymous Attestation |
| **D-CRTM** | Dynamic Core Root of Trust for Measurement |
| **DDR** | Double Data Rate |
| **DES** | Data Encryption Standard |
| **DMA** | Direct Memory Access |
| **DoS** | Denial-of-Service |
| **DRAM** | Dynamic Random Access Memory |
| **DRM** | Digital Rights Management |
| **EAL** | Evaluation Assurance Level |
| **ECB** | Electronic Code Book |
| **EEPROM** | Electrically Erasable Programmable Read-Only Memory |
| **eID** | Electronic Identity |
| **EK** | Endorsement Key |
| **EM** | Electromagnetic |
| **EMV** | Europay, MasterCard, Visa |
| **FCR** | Firmware Configuration Register |
| **FIB** | Focused Ion Beam |
| **FPGA** | Field Programmable Gate Array |
| **FSB** | Front Side Bus |
| **FTE** | Firmware Trust Engine |
| **GCM** | Galois/Counter Mode |

| | |
|---|---|
| **GPIO** | General Purpose Input/Output |
| **HCR** | Hardware Configuration Register |
| **HDL** | Hardware Description Language |
| **HEK** | Hardware Endorsement Key |
| **HMAC** | Hash-based Message Authentication Code |
| **I2C** | Inter-Integrated Circuit |
| **ICAP** | Internal Configuration Access Port |
| **IC** | Integrated Circuit |
| **IEC** | International Electrotechnical Commission |
| **IMA** | Integrity Measurement Architecture |
| **I/O** | Input/Output |
| **IOMMU** | Input/Output Memory Management Unit |
| **IP** | Intellectual Property |
| **IPC** | Inter Process Communication |
| **IRQ** | Interrupt Request |
| **ISA** | Industry Standard Architecture |
| **ISO** | International Organization for Standardization |
| **IV** | Initialization Vector |
| **JTAG** | Joint Test Action Group |
| **LPC** | Low Pin Count |
| **LQFP** | Low profile Quad Flat Package |
| **LRPUF** | Logically Reconfigurable Physical Unclonable Function |
| **LUT** | Look-Up Table |
| **MAC** | Message Authentication Code |
| **MCH** | Memory Controller Hub |
| **ME** | Management Engine |

| | |
|---|---|
| **MLC** | Multi-Level Cell |
| **MLTM** | Mobile Local Owner Trusted Module |
| **MMU** | Memory Management Unit |
| **MRTM** | Mobile Remote Owner Trusted Module |
| **MPWG** | Mobile Phone Work Group |
| **MTM** | Mobile Trusted Module |
| **MTP** | Multiple-Time Programmable |
| **NFC** | Near Field Communication |
| **NGSCB** | Next-Generation Secure Computing Base |
| **NVM** | Non-Volatile Memory |
| **OAEP** | Optimal Asymmetric Encryption Padding |
| **OCB** | Offset CodeBook |
| **OFB** | Output Feedback |
| **OTP** | One-Time Programmable |
| **PC** | Personal Computer |
| **PCI** | Peripheral Component Interconnect |
| **PCM** | Phase Change Memory |
| **PCMCIA** | Personal Computer Memory Card International Association |
| **PCR** | Platform Configuration Register |
| **PIX** | Proprietary Application Identifier Extension |
| **PKCS** | Public Key Cryptography Standards |
| **POK** | Physically Obfuscated Key |
| **PQFP** | Plastic Quad Flat Package |
| **PROM** | Programmable Read-Only Memory |
| **PUF** | Physical Unclonable Function |
| **RAM** | Random Access Memory |

| | |
|---|---|
| **RFID** | Radio Frequency Identification |
| **RID** | Registered application provider Identifier |
| **RNG** | Random Number Generator |
| **ROM** | Read-Only Memory |
| **RPUF** | Reconfigurable Physical Unclonable Function |
| **RSA** | Rivest Shamir Adleman |
| **RTC** | Real-Time Clock |
| **RTM** | Root of Trust for Measurement |
| **RTR** | Root of Trust for Reporting |
| **RTS** | Root of Trust for Storage |
| **S-CRTM** | Static Core Root of Trust for Measurement |
| **SE** | Secure Element |
| **SHA-1** | Secure Hash Algorithm 1 |
| **SIM** | Subscriber Identity Module |
| **SLC** | Single-Level Cell |
| **SMBus** | System Management Bus |
| **SMI** | System Management Interrupt |
| **SMM** | System Management Mode |
| **SML** | Stored Measurement Log |
| **SoC** | System-on-Chip |
| **SPI** | Serial Peripheral Interface |
| **SRAM** | Static Random Access Memory |
| **SRK** | Storage Root Key |
| **STM** | SMM Transfer Mode |
| **SVM** | Secure Virtual Machine |
| **TCB** | Trusted Computing Base |

| | |
|---|---|
| **TCG** | Trusted Computing Group |
| **TCPA** | Trusted Computing Platform Alliance |
| **TEAS** | Timed Executable Agent System |
| **TIS** | TPM Interface Specification |
| **TOCTOU** | Time-of-Check Time-of-Use |
| **TPM** | Trusted Platform Module |
| **TrEE** | Trusted Execution Environment |
| **TRNG** | True Random Number Generator |
| **TSM** | Trusted Service Manager |
| **TSN** | Tick Session Nonce |
| **TSS** | TCG Software Stack |
| **TSSOP** | Thin Shrink Small Outline Package |
| **TXT** | Trusted Execution Technology |
| **USB** | Universal Serial Bus |
| **UTC** | Universal Time Clock |
| **VM** | Virtual Machine |
| **VMM** | Virtual Machine Monitor |
| **VPN** | Virtual Private Network |

# Chapter 1

# Introduction

## 1.1 Background on Trusted Computing

As today's software is becoming more and more mobile and inherently networked, and its tasks get increasingly critical, mechanisms should be in place to establish trust relationships between computing platforms. For instance, in online banking the bank wants be assured that a financial transaction is generated by a legitimate client of the bank and not by malware that has infected the client's computer. Similarly, providers of digital content such as music, movies and e-books want to check whether a so-called Digital Rights Management (DRM) system is properly installed on the consumer's platform. The DRM software typically restricts the usage of the digital content; e.g., the content can only be played on a certain number of computers or media players, for a limited number of times or during a specific time period. In online games "misbehaving" users must be identified. The usage of bots that automate certain actions in the game, or the installation of cheat software that gives the user advantages over the other players (e.g., viewing through walls) must be detected. As a final example, it would be desirable in Virtual Private Network (VPN) solutions to grant remote access to a corporate network over the public Internet not only based on user credentials (e.g., password, digital signature, biometrics), but also on the verification of the platform's integrity.

For all these applications, it is clear that only legitimate, untampered client applications should be granted access to a service. Hence, an authorized entity wants to be able to both identify a remote platform and verify whether its software is running untampered. In the literature this process is often called *remote attestation.* If tampering is detected, the verifier will want to disconnect the client from the network, stop the service to this particular client, or even force that client application to halt its execution.

### 1.1.1 Closed Platforms

In closed systems, communicating platforms have an a priori trust relationship. The client platform is assumed to only run the legitimate software of the service provider and cryptographic keys to access the service can be embedded inside the device. Typical examples are Consumer Electronics (CE) devices such as DVD players and recorders, portable media players, satellite TV receivers, digital TV set-top boxes, and game consoles. Often the user of such device has an incentive to modify the original software or extract the embedded keys; for instance to play a DVD with a foreign region code, to watch pay TV for free, or to play a pirate copy of a computer game.

Typically the integrity of code executing on a closed platform does not have to be verified remotely as no software interface is provided to install malicious modified code. The fact that a device has access to the correct cryptographic keys is believed to offer sufficient evidence that the service provider is communicating with an authentic platform. Therefore there is an implicit trust relationship. The closeness of the platform's software forces attackers to resort to hardware attacks on the platform. Consequently numerous security mechanisms are commonly implemented in hardware: e.g., the initial boot loader of the platform is stored in Read-Only Memory (ROM) and only starts authorized code (i.e., signed by the device manufacturer), cryptographic keys are stored in a tamper resistant module such as a smart card, and the communication and memory buses of the platform are physically and/or cryptographically protected against eavesdropping and tampering.

### 1.1.2 Open Platforms

On open platforms such as the Personal Computer (PC) an adversary has total control over all the software including the operating system. The adversary can remotely compromise the platform through a security vulnerability, but he can also have local control of the platform if he is trying to attack an application on his own machine. The latter is for instance the case when a PC user attempts to circumvent a DRM system. Moreover, adversaries with local access can perform hardware attacks, such as using DMA to read and/or alter the main computer memory.

Establishing a secure execution environment in such conditions is a big challenge. Many enabling technologies has been researched in this area. Aucsmith [12] introduced that concept of *tamper resistant software* that has built-in integrity checks to detect tampering of its code, and Horne et al. [52] and Chang and Atallah [52] presented improved implementations of Aucsmith's concept. Typically tamper resistant software is complemented with *obfuscation* techniques [63, 64, 179, 307] that complicate the reverse engineering of the binary executable and hence make it more difficult to understand how to circumvent a tamper detection mechanism. Finally, *white-box cryptography* [309] aims to hide cryptographic keys into applications, either in a large collection of lookup tables, as proposed by Chow et al. [60, 61], or in executable code, as proposed by Michiels and Gorissen [199]. The scheme of Michiels and Gorissen is a form of tamper resistant software, as code modifications will alter the key and consequently cripple the functionality of the application.

However, when these software techniques are used to protect standalone, non-networked applications, their security is limited. Obfuscation makes the reverse engineering process more time consuming but not impossible, and most proposals for a white-box block cipher have been broken [26, 112, 139, 310]. Tamper resistant software typically calculates a checksum on its code and checks whether the checksum corresponds with an expected value. In offline applications this expected value has to be stored inside the software and the decision whether tampering has occurred, has to be taken locally by the client software itself. Wurster et al. [292, 308] showed that self-checking software can be attacked with hardware support and Tan et al. [270] observed that the tamper response mechanism is often a weak point.

Networked applications suffer less from these issues. The integrity checksums do not have to be present in the client software and the comparison between the runtime and the pre-computed checksum can be performed remotely by the service provider, that is not under control of an attacker. Additionally the service provider can periodically replace the client application with a new version, containing a different cryptographic key and/or obfuscated in another way. This *code replacement*, which was proposed in the work of Ceccato et al. [48, 49], can be used to limit the time an adversary has to reverse engineer a version of the application.

An adversary that has complete control over an untrusted platform, also has control over its input and output network traffic. This makes it difficult for a remote verifier to be assured of communicating with a particular environment on a given platform. The attacker can forward the remote attestation protocol from a tampered platform to an honest platform. Similarly, he can compromise the platform immediately after the attestation protocol has verified the integrity of the platform.

In addition, the verifier has to determine whether the software is running directly on the operating system of the platform or in a simulator, emulator or virtual machine. So-called genuinity tests were developed by Kennell and Jamieson [148] to verify whether software is running on specific hardware. These tests leverage detailed knowledge about the processor of the untrusted platform and are slow to execute on other processors or to simulate. In practice however, the proposed solution turns out to be flawed, as shown by Shankar et al. in [240].

The Pioneer system proposed by Seshadri et al. [235, 236, 237] establishes whether software on an untrusted host is untampered by calculating a checksum over its runtime memory image. If the resulting checksum is not reported within a defined time frame, the verifier assumes that the checksum function itself has been altered; the timing information helps to detect the overhead caused by modifications to the checksum functionality and redirection of the network flow. The proposed solution was first proposed for embedded systems with a low-end microcontroller [238] and later for legacy PC systems. The scheme relies on strong assumptions on the underlying hardware; e.g., the processor must not be overclocked or the size of the memory must not be increased.

Alternatively, one can limit the impact of tampering by moving critical code away from untrusted platforms. Zhang and Gupta [320] introduced *software splitting* as a technique for protecting software from piracy. The core idea of this technique is to remove small but essential components from an application and place them either on a secure server on the Internet or on a secure coprocessor (see Section 1.1.3). Dvir et al. [85] developed a non-blocking software splitting technique and Ceccato et al. [46, 47] formulated a framework to identify which portions of the client code should be moved to the server. All these schemes are a form of *server side execution*.

### 1.1.3 Secure Coprocessor

The software-based attestation schemes proposed for open platforms will never give the same confidence level as the hardware mechanisms of a closed platform. Therefore, in the nineties the idea arose to add a *secure coprocessor* to the open PC platform [248, 249, 312, 313]. This coprocessor offers a closed execution environment next to the untrustworthy legacy operating system. The security mechanisms of closed platforms are applied: the coprocessor only executes authenticated code and physical shielding provides hardware tamper resistance.

Even if plenty of research has been done on secure coprocessors, their commercial success is limited to banking networks. IBM is the main manufacturer of off-the-shelf secure coprocessor products that are freely programmable. The IBM 4758 [86] was a PCI card with a 486 processor, a cryptographic engine, and battery-backed RAM for non-volatile storage and it ran a proprietary operating system called CP/Q++ that supports custom applications. Its successors, the IBM 4764 and 4765, use a PowerPC processor and embedded Linux as operating system. The IBM secure coprocessor family supports *outbound authentication* [246, 247]: the ability of coprocessor applications to authenticate themselves to remote parties. IBM also provides an Application Programming Interface (API) called Common Cryptographic Architecture (CCA), which can be used to protect banking transactions, but Bond [32] demonstrated a number of flaws in this API.

In some sense the latest generation of *smart card* meets the definition of secure coprocessor. Traditionally smart cards have been constrained in processing power and storage capacity and a dedicated smart card reader was needed. Hence the application of classical cards has been limited to some specific tasks, such as data and entity authentication (e.g., SIM card), identification and digital signatures (e.g., eID card) and financial payments (e.g., EMV credit card). However, the latest generation of smart cards is getting a high speed interface (i.e., USB), a high density non-volatile memory (i.e., Flash memory) and a more powerful microprocessor.

With the appropriate cryptographic techniques the coprocessor can establish a secure communication channel to a remote entity through the network connection of the untrustworthy host computer. This channel can be used to report the identity and integrity of code executing in the secure environment, and to update and configure its software components. However, it is less straightforward to establish a secure path to a human operator of the open platform: the input of the user (e.g., key strokes entered on the keyboard or mouse movement) and the output to the user (e.g., information displayed on the computer screen) can be intercepted and manipulated by malicious software on the PC. For this reason, some applications (e.g., in the banking world) mandate the usage of a card reader with pin pad and display or cards with display and OK button.

### 1.1.4 Trusted Computing Platforms

In the nineties academic researchers proposed architectures to improve the trustworthiness of the PC bootstrap process. All assume the Basic Input/Output System (BIOS), which acts as initial boot loader of the PC platform, to be immutable and use it as trust anchor for a secure bootstrap. Arbaugh introduced the concept of chaining layered integrity checks [7, 8]. Each software component loaded during the boot process (starting from the BIOS) checks the integrity of the next component (by verifying a digital signature) before passing control to it. He also defined a mechanism for automatic recovery of corrupt or invalid bootstrap components [9]. This proposal effectively turns the PC into a closed platform as it restricts the software that can be booted.

Groß [115, 129] defined a secure bootstrap architecture that supports remote attestation. The platform contains a unique asymmetric key pair signed by the hardware manufacturer and the operating system is signed by the operating system producer. During startup the integrity of the operating system is checked by verifying the digital signature, and the platform signs the identity of the operating system with its private key yielding a boot certificate. During operation the integrity of the platform can be remotely verified with a cryptographic challenge-response protocol that transfers the boot and hardware certificate. A very similar solution was already presented earlier by Gasser et al. in [108].

The main initiative for a new generation of computing platforms was taken by the Trusted Computing Platform Alliance (TCPA), a consortium of most major IT companies, and its successor the Trusted Computing Group (TCG). This initiative opted for a different approach that respects the openness of the PC platform. A TCG enabled platform reliably measures the software components that get loaded during startup by calculating their cryptographic hash and records these measurements in a hardware security module, the Trusted Platform Module (TPM). This approach is called *authenticated boot*, *measured boot* or *trusted boot*. Measured boot does not impose restrictions on the operating system that the platform can boot, as the TPM merely operates as a logging device that does not actively intervene in the bootstrap process. This means that the platform can start into an arbitrary but verifiable state. After startup, the platform state can be reported to a remote entity with an attestation protocol or it can be used to securely bind secrets to a specific platform configuration in a process commonly referred to as *sealed storage*. The former enables service providers to restrict access to a network service based on the measured platform configuration and identity.

The remote attestation provided by TCG platforms has a number of issues which limit practical deployment. Firstly, in its original form the TCG attestation process posed some privacy concerns, which are partially addressed by the Direct Anonymous Attestation (DAA) protocol [38, 42] of the TPM 1.2 specification. Secondly, binary measurement of the platform configuration has scalability issues because managing the multitude of possible configurations can be troublesome, and allows for discrimination of certain configurations. Lastly, attestation of individual applications [226] necessitates a secure operating system. We will provide an analysis of the TCG remote attestation functionality in Section 2.1.

The initial focus of the TCG was on the open PC platform, resulting in the specification of a TPM. Originally the TCG envisioned that this TPM would be generic enough to be used in a large variety of computing platforms such as servers, mobile phones, computer peripherals, etc. However, it turns out that other platforms have slightly different security requirements. In particular, for mobile phones and embedded devices, which are historically more closed, it is desirable that the platform is halted before untrusted software is started, like in the research of Arbaugh. Therefore the TCG Mobile Phone Work Group (MPWG) published the specification for a Mobile Trusted Module (MTM) and proposed a reference architecture. The specification distinguishes between local and remote owner trusted modules, defines a subset of TPM commands that have to be implemented, and describes mobile specific commands, e.g., to implement *secure boot* in a standardized way.

### 1.1.5 Compatibility with Legacy Operating System

Pure software approaches for remote attestation, that rely on timed execution of a checksum function, have a number of limitations. It is impossible to uniquely identify the platform, creating an opportunity for proxy attacks that forward the attestation protocol from a tampered platform to an honest platform. To determine the expected execution time of the checksum computation, detailed knowledge about the processor of the untrusted platform is needed. The adversary will be tempted to replace the processor with a faster one such that the extra computing cycles can be used to tamper with the checksum function. The expected execution time can be unreliable because the verifier has to make a worst case assumption on the network latency, which can be rather unpredictable on the Internet.

Meanwhile, more than 600 million computers equipped with TPMs have been sold today, but their functionality is hardly used. The main reason for this is the lack of software support. If legacy operating systems such as Windows and Linux are used on a TCG platform, the chain of trust can be easily subverted, e.g., by loading a malicious device driver or by exploiting a kernel level security vulnerability. A solution that is often proposed to increase the trustworthiness of the PC platform while maintaining backward compatibility, is the usage of a Virtual Machine Monitor (VMM) or hypervisor [101, 102, 160, 181]. In this way a security critical application can run on a dedicated Virtual Machine (VM) isolated from the VM that hosts the legacy operating system. The integrity of the application VM and the hypervisor can be verified with a remote attestation protocol. Trusted virtualization layers have been researched and developed, for instance in Microsoft's Next-Generation Secure Computing Base (NGSCB) project [96, 212], the German EMSCB project [225] and the European OpenTC project [160], but are not yet commercially available.

Given the shortcomings of software-based attestation schemes and the lacking software support for TCG platforms we proposed a hardware-assisted software solution in [229, 230]. In particular we improved the Pioneer scheme by using the time stamping functionality provided by the TPM. Our solution only relies on a secure bootloader, instead of a secure operating system or a trusted virtualization layer. We will discuss this scheme in detail in Chapter 2.

Another approach is taken in the work of McCune et al. [190, 192, 193, 194]. They propose to use the late launch capability offered by AMD's Secure Virtual Machine (SVM) extensions and Intel's Trusted Execution Technology (TXT) [113, 114] in order to create a strongly isolated execution environment that can be remotely verified. The Trusted Computing Base (TCB) for this proposal is very small and hence the resulting solution potentially provides a strong level of assurance. This scheme has strong hardware requirements, i.e., a x86 processor with SVM/TXT and a TPM, and incurs significant performance overhead due to its frequent use of slow TPM operations. In 2010 McCune et al. overcame the performance issue by building a tiny hypervisor that includes a fast and minimized virtual/software TPM [191].

For more background on secure bootstrapping and remote attestation for commodity computers, we recommend the extensive survey of Parno et al. [209, 210].

## 1.2 Thesis Outline and Contributions

This section outlines the structure of the thesis and details the personal contributions. The thesis is organized in seven chapters.

**Chapter 1: Introduction.** The first chapter provides a brief background on trusted computing platforms. We also outline a summary and the contributions of each chapter separately.

**Chapter 2: Remote Attestation.** In Chapter 2 we provide an introduction to the main TCG specifications. We analyze the attestation functionality provided by the TCG and purely software-based attestation techniques. After this analysis, we present a new scheme for remote attestation that combines an existing software-based attestation scheme with the time stamping functionality of the TPM. This scheme was developed in collaboration with Wyseur and it is published in [229, 230].

**Chapter 3: Hardware Attacks.** In Chapter 3 we analyze the resilience of trusted computing platforms against hardware attacks. We mainly focus on the analysis and manipulation of the TPM's communication interface. This research, which includes experimental results on an Atmel 1.1b TPM, was done under supervision of Kursawe and the initial findings are published in [165]. In this chapter we also discuss how TPMs can be attacked theoretically with a side channel attack.

**Chapter 4: Non-Volatile State Protection.** In Chapter 4 we investigate how the non-volatile state of a TPM can be protected in external non-volatile memory. We provide a generic framework for non-volatile state protection and present the concept of PUF-based key storage. Next, we introduce reconfigurable Physical Unclonable Functions (PUFs) as a new security primitive and discuss how they can be utilized in non-volatile state protection schemes. Finally, we describe how the security perimeter of a TPM can be extended to an external non-volatile memory module with a cryptographic protocol. The research on reconfigurable PUFs, which is presented in [163], is joint work with Kursawe, Škorić and Tuyls, who came up with the concept of a reconfigurable PUF when working at Philips Research, and Sadeghi. The author of this thesis is responsible for the scheme to protect the persistent state of a TPM with a reconfigurable PUF and for the idea to create a logically reconfigurable PUF with a static PUF and embedded non-volatile memory. The work on authenticated external non-volatile memory was performed under supervision of Tuyls and it was published in [228].

**Chapter 5: Flexible TPM Architecture.** In Chapter 5 we introduced a new architecture for a secure coprocessor called $\mu$TPM, that allows simpler and more flexible TPM implementations. In order to minimize the hardware resources of the $\mu$TPM architecture, the program code of the processor is stored in external non-volatile memory and only gets loaded in internal memory when needed. The $\mu$TPM architecture was developed in collaboration with Kursawe. This chapter is an extended version of [164].

**Chapter 6: Reconfigurable Trusted Computing.** In Chapter 6 we discuss how the techniques from Chapter 4 can be used to protect the persistent state of a trusted module on currently available FPGAs. This research is published, in part, in [228]. We also describe a novel FPGA architecture that defines a root of trust to measure and report the integrity of partial bitstreams. The research on this architecture, which is presented in [87], was initially started by Eisenbarth, Güneysu, Paar, Sadeghi and Wolf from Ruhr-Universität Bochum.

The author of this thesis contributed at a later stage by helping to refine and improve the architecture.

**Chapter 7: Conclusions and Future Work.** In Chapter 7 we summarize the most important findings of this thesis and propose a number of future research directions.

# Chapter 2

# Remote Attestation

A number of applications require verification of software executing on a remote platform. Trusted computing platforms promise to solve this problem, but large scale deployment of this technology is limited because there are scalability issues and lacking software support. On the other hand, timed execution of code checksum calculations offers a solution on legacy platforms, but cannot provide strong security assurance as it solely relies on software mechanisms.

In this chapter we analyze the attestation functionality provided by the TCG and purely software-based attestation techniques. Next we present a new solution, which we presented in [229, 230], that uses the time stamping functionality of the TPM and a modified bootloader to enhance an existing timed execution scheme.

## 2.1 Attestation with Trusted Computing Platforms

Trusted computing initiatives intend to solve some of today's security problems of the underlying computing platforms through hardware and software changes. The main initiative for a new generation of computing platforms is the TCG, a consortium of most major IT companies. The TCG sees itself mainly as a standard body[1] and it does not provide any infrastructure to fully utilize the technology. Only in 2009 the TCG announced a certification program to

[1] In 2009 the TPM specifications were approved as an ISO/IEC standard, namely ISO/IEC 11889.

test the correctness of implementations.[2] The TCG specifications define three components that form a Trusted Platform.

1. The core component of a TCG platform is a hardware module called Trusted Platform Module (TPM) or Mobile Trusted Module (MTM). This component will be explained in more detail in Section 2.1.1.

2. The second component is called Core Root of Trust for Measurement (CRTM), and is the first code that the platform executes when it is booted. In a PC, this is the first part of the BIOS, which cannot be flashed or otherwise be modified. New-generation PCs with SVM/TXT support have the ability to measure and start a hypervisor after the legacy operating system has booted; this measured launch routine is known as the Dynamic Core Root of Trust for Measurement (D-CRTM) whereas the BIOS boot block is known as the Static Core Root of Trust for Measurement (S-CRTM).

3. To compensate for the lack of functionality in the TPM, the TCG specifies a TCG Software Stack (TSS), which facilitates some of the complex, but non-critical functionality and provides standard interfaces for high-level applications.

### 2.1.1 Trusted Platform Module

The TPM is a smart card like hardware module that was originally envisioned to be platform agnostic. However, in practice, the specification is primarily designed for the PC platform and therefore the TCG later on made a specification for a hardware module more tailored for advanced mobile devices such as smart phones and tablets, called MTM. The MTM specification adds some mobile specific functionalities and declares (mandatory) TPM features optional in order to minimize the footprint of the module [91].

The TPM has to be securely bound to the rest of the platform. In a PC the binding is accomplished by implementing the functionality with a dedicated discrete chip and by mounting it on the motherboard or by integrating the TPM into the chipset. Some of the first discrete TPMs were installed on a separate daughterboard plugged into the motherboard and hence a logical binding mechanism was required to guarantee that the module could not be

[2] At the moment of writing only two products, the Infineon TPM and the latest STMicroelectronics TPM, have been certified. This TCG certification program was presumably created because independent testing by Sadeghi et al. [223] revealed non-compliance bugs in some early TPM products.

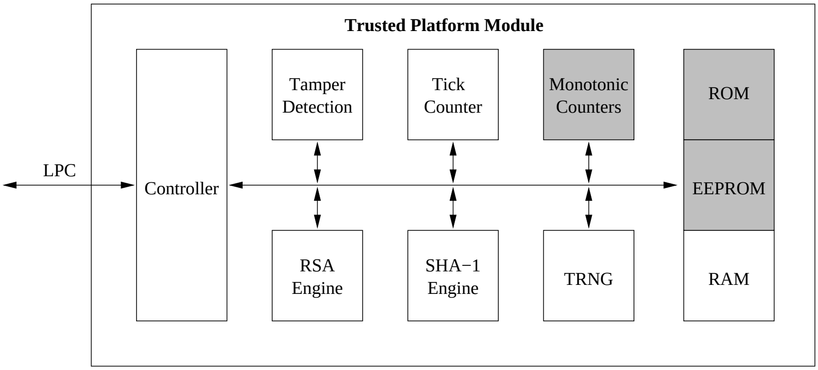


Figure 2.1: Simplified architecture of TPM 1.2.

removed and replaced with a different TPM. Nowadays the TPM chip is soldered directly onto the motherboard, establishing a physical binding.

For the MTM specification various implementation options exist, especially because a mobile phone will contain multiple trusted modules for different stakeholders (e.g., device manufacturer and cellular operator). The MTM can be implemented in hardware as a separate dedicated chip or integrated into existing chips [72], or as software running on the main processor, possibly in a higher privileged mode [90, 297]. If the platform has to support multiple execution engines, software/virtual trusted modules can run in isolated domains provided by a microkernel or hypervisor [24, 231, 234, 319].

Both TCG modules can be implemented with similar hardware, namely a microcontroller, a cryptographic coprocessor (supporting RNG, RSA, SHA-1, and HMAC), read-only memory for firmware and certificates, volatile memory and non-volatile memory. Figure 2.1 gives a schematic overview of the internal architecture of a TPM version 1.2. The trusted module communicates with the central microprocessor of the platform over an I/O bus. The Low Pin Count (LPC) bus is the standardized interface for PCs to communicate with a TPM. Some manufacturers also provide a TPM variant for embedded systems that has an Inter-Integrated Circuit (I2C) or System Management Bus (SMBus) interface (see Table 3.1).

The trusted module needs volatile memory for temporary data. This includes key slots to load keys that are stored outside the trusted module, information (e.g., nonces) about authorization sessions, and a set of so-called Platform Configuration Registers (PCRs) that are used to store measurements (i.e., hash values) about the platform configuration. The content of these registers can only[3] be modified using the irreversible process known as "extending":

$$PCR_{\text{new}} = \mathcal{H}(PCR_{\text{old}}||M)\,,$$

with $PCR_{\text{old}}$ the previous register value, $PCR_{\text{new}}$ the new value, $M$ a new measurement, $\mathcal{H}$the cryptographic hash function Secure Hash Algorithm 1 (SHA-1) and || denoting the concatenation of values. The operation has several benefits: (a) it is computationally infeasible to find two different measurement values $M$ that yield the same extended PCR value, (b) it preserves the order in which the measurements are recorded in the register (e.g., extending $M_1$ before $M_2$ results in a different value than extending $M_1$ after $M_2$), and (c) the operation allows to store an unlimited number of measurements in a single PCR value. In the 1.2 version of the TCG specifications SHA-1 is still used as hash algorithm. Although the theoretical collision attacks on SHA-1 do not pose immediate concerns for the PCR extension operation, the TPM 2.0 specification will support additional hash functions, including SHA-2 and Whirlpool.

The non-volatile memory is used to securely store the trusted module's persistent state, that includes cryptographic keys, authorization data and monotonic counters. The TPM contains two important long-term asymmetric keys:

1. The Endorsement Key (EK) uniquely identifies each TPM. During production this key is generated externally and programmed in the TPM by the manufacturer or alternatively it is generated inside the TPM (with the TPM_CreateEndorsementKeyPair command). The manufacturer may provide a certificate on the EK, however in practice Infineon and STMicroelectronics currently are the only manufacturers shipping endorsement certificates with their TPMs. This lack of endorsement certificates implies that it is not straightforward to distinguish a genuine hardware TPM from a software emulator. Optionally the TPM may support a mechanism to revoke the EK and create a new key (using the TPM_RevokeTrust and TPM_CreateRevocableEK command respectively). However this is only sensible if the owner is prepared to certify the new key himself and if the platform is required only to be trusted by parties that trust the certification (e.g., within a corporation).

2. The Storage Root Key (SRK) is uniquely created inside the TPM, when ownership over the TPM is taken, and acts as the root of the tree of storage keys. The TPM_TakeOwnership operation also generates a secret random value known as *tpmProof* which the TPM uses to identify encrypted blobs that it creates. The SRK (and *tpmProof*) can be changed by revoking (with

[3] The version 1.2 specification introduced a number of PCRs that can be reset (using TPM_PCR_Reset) by higher privileged (determined by locality) code.

TPM_OwnerClear) and re-taking ownership, but this process destroys all existing keys maintained by the TPM and hence it is probably only done when a platform is decommissioned.

Other data included in the persistent state include the owner's authorization data (i.e., password), the content of monotonic counters, volatile state information that is temporally stored with the TPM_SaveState command, and additional status information (e.g., the number of failed authorization attempts used to prevents a dictionary attack against the owner's password).

## 2.1.2 TCG Functionality

### Authorization

The TPM contains a comprehensive authorization scheme because several entities may have a relationship with one platform. Most importantly, it differs between the owner (the entity who bought the platform), the user who has physical access to the machine, and normal users who may have special rights on, for example, cryptographic keys administered by the TPM. With few exceptions, the owner is the entity with all rights to the TPM, who can, however, give up rights in favor of other users, which he then cannot revoke; in the TCG specifications this process is known as *delegation*.

An authorization secret called *AuthData* is associated with every TPM key and it can be used to limit access to that key (i.e., even for the owner of the platform). Demonstration of ownership of or authorization to use the key is done by accompanying a TPM command with an Hash-based Message Authentication Code (HMAC) of the command parameters, keyed with the *AuthData* or a shared secret derived from the *AuthData*; the response to an authorized command is also accompanied by an HMAC of the response parameters. A good overview of the TCG authorization protocols is given in [55].

The authorization secret may be weak (i.e., containing low entropy) and hence it may be guessable. Therefore the TCG mandates TPM manufacturers to implement a dictionary attack mitigation scheme; after a number of authorization failures the TPM will for instance exponentially increase the time between authorization attempts.[4]

Chen and Ryan have identified two flaws in the TCG authorization protocols. Firstly, in certain circumstances offline dictionary attacks on low-entropy

[4] The TPM owner can reset the dictionary attack mitigation scheme using the TPM_ResetLockValue command.

authorization secret are possible [54], effectively circumventing the online mitigation scheme. Secondly, sharing of authorization data between users allows a TPM impersonation attack that completely breaks the security of the TPM storage functions [55]. The TCG endorses the practice of sharing of authorization data and for instance Windows Vista applies it by setting the SRK password to a "well-known" value (all-zeros). However, the TCG explicitly states that in this particular scenario the confidentiality of authorization data can be protected with a so-called transport session (see Section 3.2.4).

### Key Management

To reduce the amount of non-volatile memory needed inside the TPM, only one key, namely the SRK, needs to be permanently stored inside the TPM. Other keys maintained by the TPM can be "wrapped" (encrypted) under the SRK or by another storage key that is already maintained by the TPM. These wrapped keys are maintained outside the TPM by the TSS, which typically stores the keys on hard disk. This allows the TPM to maintain a virtually unlimited number of keys, at the price that it gives up the control over the lifetime of keys – neither can the TPM revoke individual keys itself (save of the SRK, which then destroys all keys maintained by the TPM). Nor can the TPM prevent the operating system from destroying keys maintained by the TPM.

The two main commands to manage the key hierarchy are TPM_CreateWrapKey and TPM_LoadKey2. The TPM_CreateWrapKey command takes as argument (a pointer to) the parent key, generates a new key, and returns the generated key. The two parts of the newly created key are exported in a different way: the public part of the key pair is exported in plaintext, whereas the private part is encrypted/wrapped under the parent key. Before a TPM key can be used, it must be loaded using TPM_LoadKey2. This command takes as argument the key blob, decrypts the wrapped private key and stores it in volatile memory, and returns a handle, i.e., a pointer to the loaded key. Since the operation involves a decryption with the parent key, this key must be loaded in the TPM beforehand and its key handle is provided as argument to the command. The SRK is permanently loaded and has a well-known handle value. It is left to the TSS on the host platform to properly manage which keys are currently loaded in the TPM and what their corresponding handles are.

The TCG defines the following main types of keys:

- **Storage keys** are used to wrap other keys in the TPM's protected storage and hence form the inner nodes of the key tree.

- **Binding keys** are used to encrypt secret data (using TPM_Bind and TPM_UnBind). Typically they protect symmetric keys that the host platform uses to encrypt arbitrary sensitive information.
- **Signing keys** are used to sign arbitrary data (e.g., using TPM_Sign). They are used for signing operations only and form the leaves of the key tree.
- **Identity keys**, also known as Attestation Identity Keys (AIKs), are special signing keys used for attestation to prove that data originated in a genuine TPM (see below). They are always direct children of the SRK.
- **Legacy keys** are keys that have been created outside the TPM. They can be used for encryption and signing operations, and are useful for interoperability with existing systems.

Besides the type, TPM keys have various other properties, such as authorization data to restrict access, a particular platform configuration to which the key is bound (see below) or migration type. The TPM keys can be migratable, non-migratable or certified migratable. A non-migratable key may not leave the TPM at all; the specification does suggest an optional *maintenance* mechanism to move the entire content of one TPM to another, but this mechanism is rather complex and thus not supported by most implementations. If a key is to be migrated, authorization from the TPM owner is required.

### Integrity Measurement

The initial platform state is "measured"[5] by computing cryptographic hashes of all software components loaded during the boot process. Figure 2.2 shows the case of a TCG-compliant PC. The task of the CRTM is to measure (i.e., compute a hash of) the code and parameters of the BIOS and extend the first PCR register with this measurement (using the TPM_Extend operation explained above). Next, the BIOS will measure the binary image of the bootloader before transferring control to the bootloader, which in its turn measures the operating system. The PCRs represent an accumulated measurement of the history of

[5] The term "measurement" is normally defined as the process or the result of determining the ratio of a physical quantity (e.g., length, time, temperature) to a unit of measurement (e.g., meter, second, degree Celsius). The term "integrity measurement" does not comply with this literal definition because the integrity of platform is not a quantity that can be "measured"; i.e., it is impossible to make a distinction between more and less integrity. The TCG defines integrity measurement as "*the process of obtaining metrics of platform characteristics that affect the integrity (trustworthiness) of a platform, and putting digests of those metrics in shielded locations (called PCRs).*"

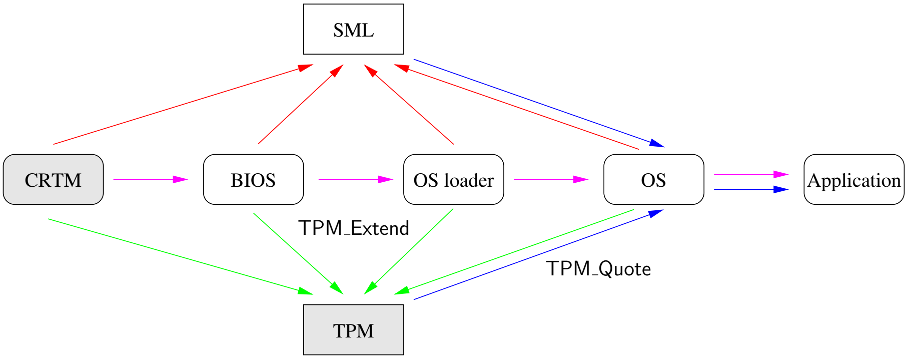


Figure 2.2: Integrity measurement during boot process of TCG-compliant PC.

all code that has executed from the power-up of the platform. In this way a chain of trust can be established from the CRTM to the operating system and potentially even to individual applications.

### Integrity Reporting

The TCG attestation allows to report the current platform configuration ($PCR_0$, ..., $PCR_n$) to a remote party. It is a challenge-response protocol, where an anti-replay challenge provided by the remote party and the current value of chosen PCRs are digitally signed with an AIK (using the TPM_Quote command). If needed, a Stored Measurement Log (SML), describing the measurements that lead to a particular PCR value, can be reported as well. The AIKs act as *pseudonyms* of the EK which uniquely identifies a TPM.

A trusted third party called Privacy Certification Authority (CA) is used to create a certificate on the public part of the AIKs.[6] The TPM_MakeIdentity command is used to create a new AIK and obtain the public part. This public AIK, the public EK and the endorsement certificate are sent to the privacy CA, who checks the endorsement certificate, signs a certificate for the AIK, and encrypts the AIK certificate with a session key, which is encrypted with the EK. The TPM_ActivateIdentity command decrypts the session key and releases it to the user software. Finally, the software uses the session key to decrypt the AIK certificate.

[6] At the moment of writing two experimental privacy CAs exists: http://www.privacyca.com was created by Hal Finney, one of the developers of the open source TSS TrouSerS, whereas http://privacyca.iaik.tugraz.at is hosted by the IAIK research group of TU Graz [215].

Version 1.2 of the TCG specification defines a cryptographic protocol called DAA [38] to eliminate the need for a Privacy CA, as it can potentially link different AIKs of the same TPM.

TCG technology also supports the concept of sealing, which enables to cryptographically bind certain data or keys to a certain platform configuration. The TPM_Seal commands takes as argument a key handle, the data to be encrypted, and information about PCRs to which the data should be bound, and returns a sealed blob. The TPM will only "unseal"/release this data if a given configuration is booted (using TPM_Unseal). This can be considered as an implicit form of attestation: an application can seal a secret in the TPM and, if the application is able to unseal this secret, the platform is known to be in a specific state.

### 2.1.3 Application Level Attestation

TCG attestation is designed to provide remote verification of the complete platform configuration, which consists of all software loaded since startup of the platform. However, establishing a chain of trust to individual programs is not straightforward in practice.

#### Operating System Requirements

The operating system needs to measure the integrity of all privileged code it loads (i.e., kernel modules), because these can be used to subvert the integrity of the kernel. Traditionally loadable kernel modules or device drivers are used to inject kernel backdoors. However, legacy operating systems are monolithic, too big and too complex to provide a sufficiently small TCB [192] and hence they are often prone to security vulnerabilities. Therefore legacy operating systems cannot guarantee a chain of trust beyond the bootloader. This is why trusted computing initiatives rely on a microkernel such as seL4 [130, 151], a hypervisor such as Xen [16, 202], or a combined microkernel-hypervisor such as NOVA [260], OKL4 microvisor [131] or PikeOS to achieve both security and backward compatibility. If the platform has hardware support for virtualization, the overhead of the hypervisor will be limited.

#### Load-Time Binary Attestation

A first approach to attest individual programs is to directly apply the TCG (i.e., load-time binary) attestation on all userland components. This approach is applied in the Integrity Measurement Architecture (IMA) of Sailer et al. [226]. On the creation of user level processes, the kernel measures the executable code loaded into the process (i.e., the original executable and shared libraries) and this code can subsequently measure security sensitive inputs that its loads (e.g., arguments, configuration files, shell scripts). All these measurements are stored in a PCR register and the SML.

In its basic form TCG attestation has some shortcomings. First, binary attestation is not scalable because a huge number of possible configurations exist. Every new version of a component will have a different binary and hence produces a different hash value. The verifier of a remote attestation process has to maintain a huge database of measurements in order to determine whether the reported configuration is trustworthy.

According to England [95], a typical Windows installation loads two hundred or more drivers from a known set of more than 4 million. Steffen [259] on the other hand reports that around 1200 files are measured by the IMA scheme during startup of a Linux desktop and for each Linux version more than 10 000 reference measurements must be stored in the attestation database. In [50] Cesena et al. investigated the scalability of TCG attestation and they conclude that between 1000 and 3700 measurements are recorded on a Linux desktop platform, depending on the configuration of IMA. They also point out that the Fedora 14 distribution consists of more than 22 000 packages, containing 2.9 million files in total.

Lastly, load-time attestation provides no runtime assurance as there can be a big time difference between integrity measurement (i.e., startup of the platform) and integrity reporting. The platform could have been compromised since it has been booted. This is sometimes referred to as a Time-of-Check Time-of-Use (TOCTOU) attack.

### Hybrid Attestation Schemes

To overcome some of the shortcomings of binary attestation, more flexible attestation mechanisms have been proposed in the literature.

The BIND scheme of Shi et al. [241] provides fine-grained attestation by not verifying the complete memory content of an application, but only the piece of the code that will be executed. Furthermore it allows to include the data that the code produces in the attestation data. The solution requires the attestation service to run in a more privileged execution environment and the integrity of the service is measured using the TPM.

In [121] the concept of semantic remote attestation is proposed by Haldar et al. This is also a hybrid attestation scheme, where a virtual machine is attested by the TPM and the trusted virtual machine will certify certain semantic properties of the running program.

Property-based attestation [216, 224] takes a similar approach where "properties" of the platform and/or applications are reported instead of hash values of the binary images. Sadeghi and Stüble [224] define a platform property as a quantity that describes an aspect of the behavior of that platform with respect to certain requirements. A platform property could for instance state that it strictly isolates processes from each other or that it is complies with privacy laws. One practical proposal is to use delegation-based property attestation: a certification agency certifies a mapping between properties and configurations and publishes these property certificates [161].

## 2.2 Software-based Attestation on Legacy Platforms

In this section we present two software-based attestation solutions that offer an alternative for TCG attestation on legacy platforms that are not (yet) equipped with a TPM. They rely on the timed execution of a checksum function: the Pioneer scheme of [235, 236, 237] and the Timed Executable Agent System (TEAS) solution of Garay and Huelsbergen [99].

### 2.2.1 Checksum Functions

A widely implemented technique in software tamper resistance is the use of checksum functions (e.g., in software guards [52]). These functions read the software code, compute a hash value and check whether the value corresponds with an expected value that was pre-computed. If the values do not match, the software is assumed to be tampered with and an appropriate response must be taken; in an offline scenario the software will typically be stopped and in a networked application the tampered software is no longer granted access to a network service.

Note that the hash function used does not necessarily have to satisfy all requirements of a cryptographic hash function. It must provide second preimage resistance, such that the software cannot be tampered with in a meaningful way and still yield the same hash value. In order to protect against insiders, the function must also resist collision attacks.

In [292, 308] Wurster et al. describe a generic memory copy attack on check functions. This attack tries to distinguish if code instructions are interpreted/executed or if they are read (i.e., when they are used as input to a checksum function). Hence, tamper detection can be fooled when reading of code is redirected to an untampered copy, although a tampered copy is executed. Wurster et al. analyze how the Memory Management Unit (MMU) of modern process architectures can be used to facilitate the attack.

Two techniques to detect memory copy attacks have been proposed. A first approach is the accurate measurement of the execution time of the checksum function. Memory copy attacks introduce some levels of indirection, which imply extra computations that slow down the execution, and this behavior can be detected.

A second option that is proposed by Giffin et al. in [109], is the usage of self-modifying code to detect a memory copy attack. If the verification function modifies itself, only the clean (i.e., untampered) memory copy, where memory reads/writes are pointed to, will be updated. Doing so, a verifier can notice that the execution, i.e., running the unmodified tampered copy, has not been changed, and thus detect the attack.

## 2.2.2 Pioneer

In [238] Seshadri et al. describe a remote attestation solution for embedded devices, without the need for any hardware changes (e.g., the addition of a TPM). Later, they proposed an adapted solution for legacy PC systems, called Pioneer [236, 237]. The Pioneer scheme consists of a two-stage challenge-response protocol. First, the verifier obtains an assurance that a verification agent is present inside the operating system on the untrusted host. Next, this verification agent reports the integrity of the executable that the verifier is interested in, similar to TCG attestation.

### Protocol Description

The detailed steps of the Pioneer protocol are depicted in Figure 2.3.

1. The verifier invokes the verification agent $V$ on the untrusted host by sending a challenge $n$, and starts timing its execution: $t_1 \leftarrow t_{\text{current}}$.
2. This challenge is used as a seed for a pseudo-random walk through the memory of the verification agent. Based on this walk, a checksum is computed: $c \leftarrow \mathsf{cksum}(n, V)$.

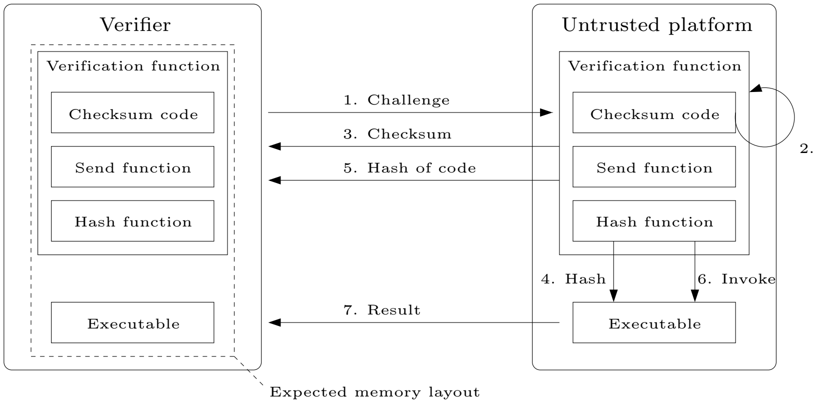


Figure 2.3: Schematic overview of Pioneer protocol.

3. The verification agent reports the checksum $c$ to the verifier. The verifier can now check the integrity of the verification agent by verifying that two conditions are satisfied:

   (a) The checksum must correspond with the value that the verifier has calculated on its own local copy of the verification agent.

   (b) The fingerprint of the verification agent must be delivered in time ($t_2 \leftarrow t_{\text{current}}$), i.e., the verifier knows an upper bound on the expected execution time of the checksum calculation:

   $$t_2 - t_1 < \Delta t_{\text{expected}} = \Delta t_{\text{cksum}} + \Delta t_{\text{network}} + \delta t\,,$$

   with $\Delta t_{\text{cksum}}$ the expected execution time of the checksum function, $\Delta t_{\text{network}}$ the network delay, and $\delta t$ some margin.

4. The verification agent computes a cryptographic hash of the executable $E$ as a function of the original nonce: $h \leftarrow \mathcal{H}(n||E)$.

5. This hash is sent to and verified by the verifier. Again, the verifier needs to independently perform the same computation on a local copy of the executable.

6. The verification agent invokes the application $E$ and transfers control to it.

### Checksum Function

When an adversary attempts to produce a correct checksum while running tampered code, this should be detectable due to an execution slowdown. In Pioneer, the Program Counter value and/or the Data Pointer value are incorporated into the checksum computation in order to cause a measurable execution slowdown, when a memory copy attack is deployed. Because an adversary needs to forge these values as well, this will lead to an increase in execution time.

The design of the checksum function cksum() in Pioneer is subject to several constraints:

- The checksum function should be optimal in execution time. If an adversary would be able to optimize the checksum function, he would gain time to perform malicious actions.
- To maximize the adversary's overhead, the checksum function will read the memory in a pseudo-random traversal. This prevents the adversary from predicting the memory reads beforehand. The challenge $n$ seeds the pseudo-random traversal.
- The execution time of the checksum function must be predictable. Hence, Pioneer needs to run in supervisor mode and with interrupts disabled. For this reason, the proof-of-concept implementation of Pioneer by the authors runs inside a network interface driver of the Linux kernel.

### Shortcomings

The security of the Pioneer solution relies on three important assumptions.

First, the verifier needs to know the exact hardware configuration of the untrusted platform, including the Central Processing Unit (CPU) model, clock speed and memory latency, in order to compute the expected untampered execution time. If an adversary is able to replace or overclock the CPU, he could influence the execution time. Hence in the Pioneer system, it is assumed that the hardware configuration is known by the verification entity and cannot be changed.

Secondly, an adversary could act as a proxy, and ask a faster computing device to compute the checksum on his behalf. To avoid this, in the Pioneer protocol, it is assumed that there is an authenticated communication channel between the verification entity and the untrusted execution platform.

Finally, a general problem that remains is the network latency. Hence Pioneer assumes that the verification entity is located on the same local network as the untrusted execution platform. Consequently the network latency is less probabilistic than on the public Internet.

### 2.2.3 Timed Executable Agent System

Garay and Huelsbergen also rely on the time execution of a verification agent in their TEAS [99]. Contrary to Pioneer, TEAS issues a challenge that is an obfuscated executable program potentially computing any checksum function. Hence, the verification agent is mobile in TEAS, while Pioneer uses a single fixed verification function invoked with a random challenge.

The motivation is that an attacker has to reverse engineer the obfuscated and unpredictable agent (i.e., to gather information on the checksum function used) and that he has to do this within the expected time, in order to fool the verification entity. It should be noted that the verification entity still has to keep track of execution time to detect hardware assisted memory copy attacks.

The TEAS solution is a generic framework for remote attestation without hardware support. However, as no (proof-of-concept) implementation exists and as the authors did not consider the memory copy attack of Wurster et al., it is difficult to judge the practical security of this approach.

## 2.3 Local Execution Time Measurement with TPMs

In [229, 230] we proposed a new variant of the Pioneer scheme that relies on the time stamping feature of TPMs. The core idea of our proposal is to locally measure the execution time of the checksum function, instead of timing the execution remotely on the verifier. Additionally, the usage of a TPM enables to identify on which platform the software is executing. In [232, 233] Sadeghi et al. propose an alternative software-based attestation scheme that binds the software to the platform by means of a PUF, instead of with a TPM.

In this section we first describe the time stamping feature of TPMs and next how this functionality can be used to enhance software-based attestation schemes that rely on timed execution. The basic version of our scheme does not require extensive trusted computing software support, because we do not use the integrity measurement functionality of the TPM. The only requirement is a TCG-enabled BIOS to startup the TPM and a TPM device driver to use the time stamping functionality. In Section 2.4 we will describe some enhancements to the basic scheme, that utilize a trusted bootloader.

### 2.3.1 TPM Time Stamping

Time stamping is one of the features in version 1.2 of the TPM specification that was not supported by TPM 1.1b. The TPM can create a time stamp on a blob:

$$TS \leftarrow \mathsf{sign}_{SK}(blob||t||TSN)\,,$$

with $SK$ a signature or identity key, $blob$ the digest to stamp, $t$ the current time and $TSN$ a nonce determined by the TPM. The time stamp $TS$ does not include an actual Universal Time Clock (UTC) value, but rather the number of timer ticks that the TPM has counted since startup of the platform. Therefore the functionality is sometimes called *tick stamping* instead of time stamping. It is the responsibility of the caller to associate the ticks to an actual UTC time, which can be done in a similar way as in online clock synchronization protocols.

#### Tick Session

The TPM counts ticks from the start of a timing session, which is identified with the Tick Session Nonce (TSN). On a PC, the TPM may use the clock of the LPC bus as timing source, but it may also have a separate clock circuit (e.g., with an internal crystal oscillator or with an oscillator circuit). At the beginning of a tick session, the tick counter is reset to 0 and the session nonce $TSN$ is randomly generated by the TPM. The beginning of a timing session is platform dependent. In a PC desktop power is continually available to the TPM by using power from the wall socket, while in a mobile platform power may be unavailable when the platform is in a suspend or sleep mode. In laptops, the clock of the LPC bus can be stopped to save power, which could imply that the tick counter is stopped as well. Consequently it depends on the platform whether the TPM will have the ability to maintain the tick counter across power cycles or in different power modes on a platform.

#### Tick Counter Resolution

According to the specification the tick counter has to have a maximum resolution of 1 $\mu$s, and a minimum resolution of 1 ms. Our initial experiments show that the Infineon TPM has a resolution of 1 ms and that the Atmel TPM clearly violates the TCG specification [282]. Subsequential invocations of the TPM_GetTicks command on the Atmel TPM give a tick count value that is incremented with one; so effectively its tick counter behaves as a monotonic counter and not as a clock![7] This is not the first instance of non-compliance of TPM implementations with the TCG specification [223].

### 2.3.2 Improved Pioneer Protocol

We propose to improve the Pioneer protocol by employing the tick stamping functionality described above (see Figure 2.4).

1. The verifier $V$ sends a challenge $n$ to the verification agent $A$.
2. The verification agent uses the TPM to create a tick stamp on this challenge: $TS_1 \leftarrow \mathsf{sign}_{SK}(n||t_1||TSN_1)$. The result $TS_1$ is sent to the verifier.
3. The verification agent uses $TS_1$ as seed for the pseudo-random walk through its memory, resulting in a fingerprint: $c \leftarrow \mathsf{cksum}(TS_1, V)$.
4. The calculated checksum gets time stamped by the TPM as well: $TS_2 \leftarrow \mathsf{sign}_{SK}(c||t_2||TSN_2)$. This result $TS_2$ gets reported to the verifier.
5. The verifier can now verify the integrity of the verification agent by performing the following steps:
    (a) Verify the two signatures $TS_1$ and $TS_2$. At this stage the untrusted platform can be uniquely identified.
    (b) Check if $TSN_1 = TSN_2$. This enables the verifier to determine whether the TPM has been reset by a platform reboot or a hardware attack (see Chapter 3).
    (c) Extract the tick counters $t_1$ and $t_2$ from the time stamps and check whether $t_2 - t_1$ corresponds with the expected execution time of the checksum function:

    $$t_2 - t_1 < \Delta t_{\text{expected}} = \Delta t_{\text{cksum}} + \Delta t_{\text{sign}} + \delta t\,,$$

    with $\Delta t_{\text{cksum}}$ the expected execution time of the checksum function, $\Delta t_{\text{sign}}$ the TPM signing duration, and $\delta t$ a bound on the latency between the operations and the time to generate a TPM tick stamp.

[7] This behavior is valid in an older revision (64) of the 1.2 specification, where the TPM only needs to guarantee that "*the clock value will increment at least once prior to the execution of any command.*" Sending other commands between two TPM_GetTicks requests, confirms that the tick counter of the Atmel TPM increments after every command.

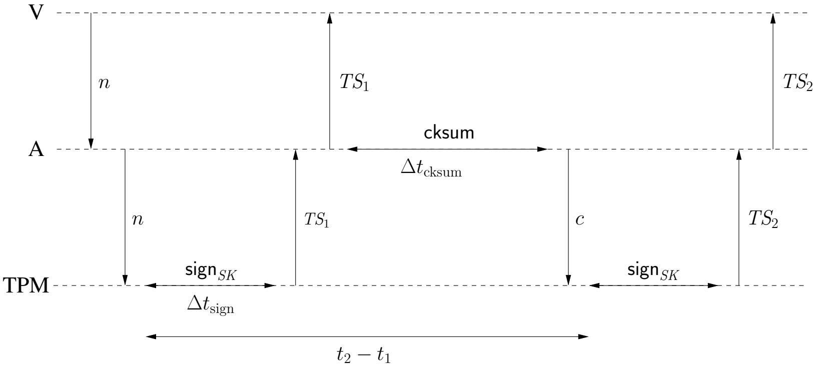


Figure 2.4: Time overview of the improved Pioneer protocol. The three entities involved are the remote verifier V, the local verification agent A and the TPM. $n$ denotes the challenge sent by V, $TS_1$ the timestamp produced by the TPM on $n$, $c$ the checksum computed by A based on $TS_1$, and finally $TS_2$ the timestamp produced by the TPM on $c$.

(d) Check whether the checksum $c$ corresponds with the value that the verifier has calculated on its own local copy of the verification agent.

(e) The subsequent steps are the same as in the original Pioneer protocol (step 4 to 6 in Figure 2.3).

The advantage of this improved Pioneer protocol is that the timing is moved from the verifier to the verification agent on the untrusted platform. Consequently, the verifier no longer needs to take into account the (non-deterministic) network latency. Instead the verifier has to know the duration of a TPM signature generation $\Delta t_{\text{sign}}$, which will depend on the actual TPM used. We expect that this time is constant because the length of the data that is signed, is fixed. Otherwise the TPM would be trivially vulnerable to a timing analysis. Hence, the total expected computation time $\Delta t_{\text{expected}}$ can be estimated accurately.

Because each TPM signs with a unique key $SK$, an authenticated channel is established between the verifier and the verification agent. If a verifier holds a database that links the TPM signing key to the CPU specification of the platform, he can take this into account to estimate the expected execution time $\Delta t_{\text{cksum}}$ of the checksum function. It should be noted that the length of the pseudo-random walk calculated by cksum() has to be sufficiently large as the resolution of the TPM tick counter is limited.

In order to deploy this system, only a TPM device driver, which is available for most legacy operating systems, needs to be installed on the untrusted platform.

There is no need for extensive software support (like a full-blown TSS), because the scheme does not rely on TCG attestation. Note that the Pioneer verification agent is part of the operating system, so this functionality has to be added.

Nevertheless, an adversary is still able to replace the CPU or install faster memory to attack the system. In Section 2.4 we will address this shortcoming with an adapted bootloader.

### 2.3.3 Proxy Attacks

Although the improved protocol addresses a great deal of the issues raised in Pioneer, it still remains vulnerable to a proxy attack. A slow computer with a TPM can send its time stamp $TS_1$ to a fast computer that computes the checksum results. This result $c$ is sent back to the slow machine that provides a signed attestation $TS_2$ to the verifier. The network overhead, of forwarding the communication between the two computers, is absorbed by the reduced computation. We provide two possible strategies to address this attack.

Firstly, in the original Pioneer protocol, a checksum is computed over the memory of the verification function, which includes the send function. The verification agent can be modified to only accept messages from the verifier, based on the network address. This would mean that the fast computer that runs the genuine verification agent, will not accept the request from the slow computer. Similarly, the address of the verifier can be included in the checksum calculation. However, in practice network addresses can be spoofed.

Secondly, the verification agent also contains a function to communicate with the TPM. If the checksum function is computed over this function as well, then there is a guarantee that there is only one way to invoke the verification agent. In this case the genuine verification agent will always use its internal TPM, and not a remote TPM.

### 2.3.4 Experimental Results

In our original research, which is published in [229, 230], we did some experiments with the TPM tickstamping functionality, but we did not make a full implementation of the improved Pioneer protocol.

In 2012 Kovah et al. [159] proposed a software-based attestation scheme that is based on the Pioneer protocol and they presented a working Windows implementation of their scheme. They implemented a variant that measures the execution time of a checksum function using the network round-trip time and a variant that uses TPM tick stamping. They tested both variants on 32 identical hosts that were equipped with a STMicroelectronics TPM. The authors confirmed experimentally that the TPM tick stamping functionality can be used to measure the overhead of a checksum forgery attack and consequently to detect the presence of an attacker. However, they concluded that the TPM tick counter is not accurate enough to measure the network delay of a proxy attack: the network delay was around 1.5 ms for their setup, while the resolution of the TPM tick counter is only 1 ms, and hence the overhead of the proxy attack was only 1 tick.

With their experiments Kovah et al. identified two shortcomings of the TPM-based variant. First, the TPM tick stamping adds a considerable overhead. The TPM-based variant takes about 1.3 seconds for the complete protocol, whereas the running time of their checksum function is only 0.1 second. So the calculation of the two TPM tick stamps takes at least 1 second. Second, the authors observed a lot of drift between the tick counters of different TPMs. The measurements of the checksum calculation differ slightly for each TPM, while the underlying hardware is identical. This signifies that the verifier cannot set a single baseline for the expected number of TPM ticks for the checksum function across different hosts. The authors proposed to measure this baseline for each host with a provisioning process in a dedicated environment. We believe that this issue can also solved with an adapted bootloader (see Section 2.4).

## 2.4 Configuration Identification with Trusted Bootloader

The TPM-assisted Pioneer scheme can be further improved by using the TPM to report the processor specification. In this way some hardware attacks, where the processor or/and the memory get replaced by faster variants, can be detected during attestation. More specifically, we propose to modify the bootloader. Bootloaders tend to be a lot smaller, and hence more trustworthy, than legacy operating systems. The OSLO bootloader [147] for instance is around 1000 lines of code, while a Linux 2.6 kernel contains more than 7 million lines of code. The integrity of the enhanced bootloader can be reported using standard TCG functionality. We still rely on timed execution to detect the compromise of legacy operating systems, given that the correct processor specification is known.

### 2.4.1 Processor Identification

A first approach is to enhance the bootloader to report the processor identifier to the TPM. Pentium class processors for instance have a CPUID instruction that returns the vendor identifier (e.g., Intel or AMD), stepping, model, and family information, cache size, clock frequency, presence of features (like MMX/SSE), etc. All this information needs to be stored in the SML and its hash should be extended to one of the PCRs. Before the improved Pioneer protocol is performed, the TPM will attest that the trusted bootloader was loaded correctly (i.e., its hash is stored in a certain PCR) and identifies the processor by digitally signing the PCR that contains the hashed processor identifier.

This mechanism allows to detect processor replacement and simulation, because the expected execution time will depend on the processor identification. On the other hand, this scheme cannot cope with the replacement of the computer memory (i.e., upgrading Random Access Memory (RAM) with lower latency).

### 2.4.2 Runtime Checksum Performance Measurement

Another strategy is to run performance measurement code during the startup of the platform. The bootloader could be adapted to run the Pioneer checksum function with a locally generated challenge (e.g., produced by the TPM random number generator) and measure the required execution time. This timing can be measured accurately with the CPU cycle counter (i.e., the RDTSC instruction for Pentium class CPUs) or with lower precision using the TPM time stamping mechanism described earlier. The trusted bootloader will report this performance benchmark to the TPM, which later can sign the recorded value. This benchmark is again stored in a PCR register and logged in the SML.

This technique can provide the verifier with a very accurate expectation of the checksum function's execution time. During the attestation phase, the verifier can rely on the timing information determined by trusted bootloader. Both processor and memory changes can be successfully and efficiently detected in this way.

This approach can also address the issue that is raised in [159], namely that the duration of the tick stamp generation $\Delta t_{\text{sign}}$ is TPM specific.

## 2.5 Conclusion

At the moment commercially available operating system only offer limited trusted computing support. At most they provide a TPM device driver, a TSS and/or a TPM-aware bootloader. This however is insufficient to achieve remote attestation of individual applications. In the meantime, pure software-based attestation schemes have been proposed for legacy platforms. They rely on the timed execution of a checksum function, that computes an application fingerprint. The execution time is measured remotely by the verifier, imposing heavy assumptions that are difficult to achieve in practice.

In this chapter, we have proposed improvements for these software-based attestation protocols. By using the time stamping functionality of a TPM, the execution time of the fingerprint computation can be measured locally. This also allows to uniquely identify the platform that is being verified. The solution can be further strengthened with a trusted bootloader. This bootloader can identify the processor specification of the untrusted platform and provide accurate timing information about the checksum function.

# Chapter 3

# Hardware Attacks

A TCG platform consists of two important security components that act as roots of trust: an immutable software component called CRTM which initiates the measurement of the boot process, and a hardware component called TPM which reports the measured platform state and offers protected storage of data. The components must be trusted if the platform is to be trusted.

In this chapter we analyze the resilience of these components against hardware attacks. We mainly focus on the analysis and manipulation of trusted platform communication, as we have presented in [165].

## 3.1 Attacks on Trusted Computing Platforms

There are a number of ways to circumvent the security features provided by a trusted computing platform. A first option is to target the TPM directly and try to extract the secrets that the TPM protects. Once the SRK is revealed, all keys in the TPM's key hierarchy are compromised. Knowledge of the private part of the EK enables the creation of software clones of a genuine hardware TPM. A second approach is to leave the TPM untouched, but falsify the platform state that is recorded in the TPM. This can be done by modifying the CRTM which is supposed to be immutable, by manipulating the integrity measurements as they are sent to TPM, or by compromising software after its integrity has been measured. This second type of attack compromises the remote attestation and sealed storage functionality because a different configuration can be recorded in the TPM than the configuration that is actually started on the platform.

### 3.1.1 Attacks on the TPM

The TPM specifications have received a lot of scrutinizing by independent security researchers. In [54, 55] Chen and Ryan identified attacks on the TCG authorization protocols. In particular, they discourage the practice to set the authorization data of the SRK to a well-known value and share it between multiple users; knowledge of this secret allows a TPM impersonation attack. They also show that in certain circumstances dictionary attacks are possible on authorization data. Other flaws and inconsistencies of less significant nature have been found in [39, 120, 178].

Sadeghi et al. developed a prototype test suite for TPM compliance testing [223]. Their tests revealed non-compliant behavior with respect to the TCG specification and bugs in several early TPM implementations. They also illustrated that these bugs may have a security impact. For instance, in one commercial TPM implementation the dictionary attack mitigation mechanism could be bypassed. Initially the TCG saw itself primarily as a standard body and did not perform any certification of products of its members. However, in April 2009 the TCG announced a certification program. For a TPM product to achieve TCG certification, it must undergo functional testing using an automated compliance test suite and a third-party security evaluation using Common Criteria (CC) according to a Protection Profile developed by the TCG.

The TCG trust model assumes software attacks only, but most TPM chips do offer some form of protection against hardware attacks as they are typically derived from security processors that are also used in smart cards. The TPM is designed to be a low-cost component that should not increase the cost of the platform significantly. This is one of the reasons why TPMs are primarily present in high-end, enterprise-class laptop and PC series, and not in low-end computers (e.g., netbooks). Given their low price tag, it is reasonable to assume that the TPM-chips on the market can be reverse engineered with state-of-the-art techniques [277] and that they are susceptible to physical attacks.

The wide range of techniques that has been developed to physically attack hardware security modules, such as smart cards, can be applied on TPMs as well. Invasive attacks can be used to read out the content of memories or observe data buses while the chip is operating [5, 6, 126, 157]. This type of attacks typically requires a high budget, qualified specialists and expensive equipment such as a Focused Ion Beam (FIB), an electron microscope, a laser cutter, and/or microprobing station. Fault attacks are another way to attack cryptographic implementations [15, 25, 33]. Faults can be induced in a non-invasive manner, for instance with glitches in the external power or clock supply [6], or semi-

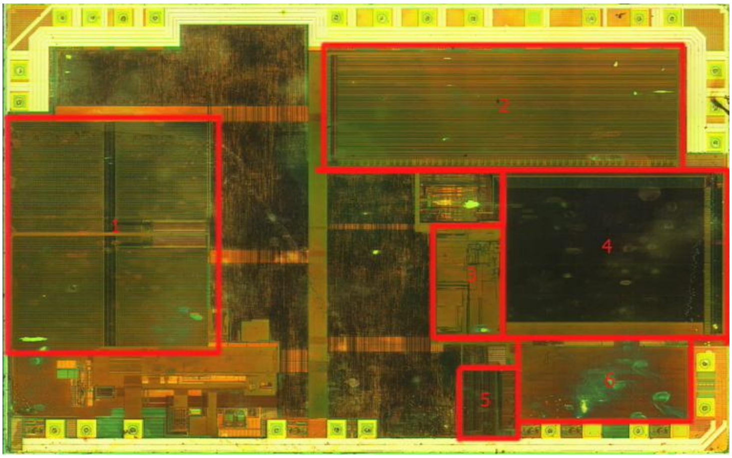


Figure 3.1: Chip layout of Infineon SLB 9635 TT 1.2 TPM [271]. The main components are (1) SRAM memory, (2) EEPROM memory, (3) microprocessor, (4) ROM memory, (5) DES engine and (6) RSA engine.

invasively, for example with a laser or white light [244]. Side-channel attacks exploit the fact that a physical implementation of a cryptographic algorithm leaks unwanted information while processing secret data. The physical leakage, including timing [154], power consumption [155] and Electromagnetic (EM) radiation [98, 219], can be measured externally.

In 2010 Tarnovsky demonstrated a successful invasive attack on the Infineon TPM [110, 271]. This chip is generally considered as one of the most secure on the market because it is the first that has undergone a Common Criteria evaluation at Evaluation Assurance Level (EAL) 4+. Figure 3.1 shows an image that he took of the chip's layout and highlights the main components. The development of the technique to bypass the tamper protective mesh on top of the chip and to microprobe the data bus of the microprocessor took 6 months. Tarnovsky estimates that another 6 months are needed to analyze the microprocessor code and to determine how keys are stored in the EEPROM. The attack is fairly sophisticated and requires access to a FIB. It involves the following steps: de-packaging the chip, removing the passivation layer, bridging the protective mesh, digging holes through the mesh, and probing the wires of the data bus one by one with a needle. In order to ease the eavesdropping of the bus, certain countermeasures, such as the usage of the internal clock and the generation of dummy bus cycles, must be disabled by injecting faults with a second needle.

To date, no side-channel attacks nor fault attacks on a commercial TPM-chip have been reported, not even failed attempts. In Section 3.4 we will outline which TPM commands are good candidates to attack with a side-channel analysis and describe the impact of such an analysis.

In [159] Kovah et al. did some experiments with the TPM tick stamping functionality. Quite surprisingly, they observed that the tick stamp times of the Broadcom TPM differ depending on the signing key that is used, whereas the STMicroelectronics TPM does not exhibit this behavior. This could signify that they unknowingly discovered that the Broadcom TPM is vulnerable to simple timing attacks.

### 3.1.2 Attacks on the Platform

Attackers can target other platform components than the TPM, with as goal the subversion of the integrity measurement. The attack with the highest impact is a compromise of the CRTM. According to the TCG specifications, an appropriate security mechanism should be in place to guarantee that the CRTM is immutable, in the sense that it can only be replaced or modified by an agent approved by the platform manufacturer. In a PC the S-CRTM will either be the BIOS Boot Block or the entire BIOS. In [147] Kauer demonstrated that the immutability condition is not met on an early TPM-equipped laptop, as he was able to reflash the BIOS with a modified image that does not extend PCR registers. New BIOSes have protection against unauthorized reflashing; they will only accept updates that are signed by the manufacturer. However, it is dubious that these solutions provide appropriate protection against hardware attacks, such as in-circuit reprogramming or replacement of the Flash chip. In [304] Wojtczuk and Rutkowska demonstrated an attack on Intel BIOSes with Flash protection. They were able to exploit a heap overflow in the code that parses the (unsigned) boot splash logo. As the parser executes early in the boot process before the reflashing locks are applied, they were able to overwrite the BIOS image. It is unclear whether this vulnerability can be used to compromise the CRTM.

The TCG specifications also state that it should not be possible to reset the TPM without resetting the whole platform. In [165] we suggested that this condition might not hold if an attacker has physical access to the reset line of the LPC bus. Our suspicion was confirmed by Kauer [147] and Sparks [250] independently. Kauer also observed that the Infineon 1.1b TPM can be reset by setting the reset bit in a control register. The TPM reset attack and its impact will be described in more detail in Section 3.2.2.

As already observed in the previous chapter, the TCG's integrity measurement and reporting functionality only provides assurance about the load-time configuration of a platform. The software running on the platform can be attacked at runtime, after it has been measured [37]. Typically, this can be done by exploiting a software vulnerability in the operating system, such as a buffer overflow, a format string bug, a race condition, etc. So trusted computing technology is only a building block to increase the trustworthiness of platforms, but it is useless without appropriate software support.

It is very important to note that the TCB consists of a number of components that must be trusted. In a traditional system, it consists of at least the CPU, the chipset, the TPM, the BIOS, the kernel of the operating system, all the applications running with administrator privilege, and probably all the devices connected to the machine. This implies that a number of additional attack vectors exists besides the exploitation of software bugs in highly privileged code. The memory of the operating system can be read and modified at runtime with Direct Memory Access (DMA). DMA attacks are typically performed using a malicious PCI/PCMCIA card [44, 71, 214] or with a FireWire device [18, 31, 77]. However they can also be performed by a legitimate device that has been compromised. In [84] Duflot et al. describe an attack on a Broadcom network card. Through an implementation bug in a remote administration protocol, they were able to compromise the firmware of the network card's embedded microprocessor and sequentially write to the main host memory using a DMA transfer. In [273] Tereshkin and Wojtczuk demonstrated an attack on Intel's remote management functionality called Active Management Technology (AMT), which is included in the latest generation chipsets. They were able to run malicious code on the microprocessor that is embedded in the Memory Controller Hub (MCH) (i.e., Intel terminology for the northbridge chip). This microprocessor, which is called Management Engine (ME), can access the main memory with DMA and hence can at runtime compromise the operating system and potentially even the S-CRTM. Their attack no longer works on newer AMT versions, which only execute firmware that is digitally signed by Intel [175].

Additional hardware changes to the PC platform complement the TCG technology and address some of the issues described above. The x86 hardware virtualization extensions, in particular the addition of an Input/Output Memory Management Unit (IOMMU) (known as Intel VT-d and AMD-Vi), allow a hypervisor to restrict the memory that devices may access. When this functionality is used, the TCB will typically consist of the CPU, the chipset, the TPM, the BIOS, a minimal hypervisor and its management domain. The guest operating systems and their device drivers do not have to be trusted and devices cannot perform DMA attacks on the hypervisor.

The late launch feature provided by Intel TXT and AMD SVM offers a solution for the TPM reset attack. More details will be provided in Section 3.2.2. It is a misconception that by using late launch the BIOS can be excluded from the TCB. The BIOS is no longer responsible for the integrity measurement of the TCB (i.e., its role as CRTM), but it still provides low-level structures such System Management Interrupt (SMI) handlers and Advanced Configuration Power Interface (ACPI) tables that can be used to subvert the security of the platform [82].

In 2009 two attacks were announced against Intel TXT by Wojtczuk et al. In [303] they presented an implementation bug in a software component of TXT and in [301] an attack was demonstrated using malicious System Management Mode (SMM) code. In the second attack they inserted an SMM backdoor which is undetected by the late launch, and used it later to compromise the hypervisor. This attack method is not new as it was for instance demonstrated in 2006 by Duflot et al. in [80]. Recent chipsets limit access to the memory in which the SMM code resides, but the currently implemented protection is ineffective because it can be circumvented by a chipset bug [301] or with CPU cache poisoning [81, 82, 83, 302]. It is important to note that the design team of Intel TXT was well aware of potential SMM attacks. In [113, 114] the concept of an SMM Transfer Mode (STM) that shields SMM code, is introduced. However currently no known implementation of an STM is available.

## 3.2 Attacking the TPM Communication Bus

The previous section illustrates that the TPM is reasonably secure against software attacks and non-invasive hardware attacks, and that the CRTMs, both static and dynamic, of recent PC platforms seems to be immutable. This suggests that the necessary hardware support is available to build a secure and trustworthy system with the TCG technology. However in [165] we identified that most PC implementations of the TCG architecture literally and figuratively have a weak link: the communication channel between the TPM and the CPU is unsecured. In PCs the TPM is usually connected to the LPC bus, which also hosts the BIOS Flash memory and low-bandwidth "legacy" Input/Output (I/O) devices (such as serial and parallel port, PS/2 keyboard and mouse, floppy disk controller). We demonstrated that it is feasible to sniff the LPC bus and eavesdrop the TPM communication. Section 3.3 will provide more details about our experiments.

In our work we used a fairly expensive logic analyzer and the lines of the LPC bus were easily accessible because the TPM was mounted on a daughterboard. However, this does not imply that the monitoring of trusted platform communication is impossible when access to the bus is less straightforward (e.g., the TPM is mounted directly on the motherboard), nor that it is costly. In [135, 136] Huang demonstrated that the HyperTransport bus between the northbridge and southbridge chip of the Microsoft Xbox can be sniffed cheaply with a custom tapping board connected to an FPGA board. The HyperTransport bus of the Xbox has a much higher bandwidth than the LPC bus (8-bit/200 MHz DDR versus 4-bit/33 MHz) and it is more difficult to access. Huang gained physical access to the bus by removing the solder mask of the Xbox's motherboard (i.e., the protective coating over the copper traces) with fine-grit sand paper and by soldering the tapping circuit onto the exposed copper traces.

During the European Trusted Infrastructure Summer School 2009 Winter [298] demonstrated a low-cost LPC bus analyzer, based on a Xilinx Spartan 3E FPGA and a Cypress FX2 microcontroller with USB 2.0 interface, and used it to eavesdrop the TPM communication. Winter extended his attack setup [299, 300] and showed how the TPM communication can be actively manipulated on the LPC bus.

Three types of attacks can be distinguished that utilize physical access to the communication bus of the TPM: passive eavesdropping of the TPM communication, hardware reset of the TPM after platform startup, and active modification of the TPM communication. For each attack scenario we will discuss the security implications and we will also describe possible countermeasures.

### 3.2.1 Passive Monitoring

The commands issued to the TPM and the corresponding responses produced by the TPM can be recorded. This technique proved useful in the development of open source device drivers for the first TPMs. The TPM 1.1b specification does not specify the low-level interface of the TPM (e.g., type of bus cycles or I/O addresses). The undocumented, proprietary protocols could be reverse engineered by sniffing the communication of the closed Windows drivers. For the 1.2 version of the specification, the interface has been standardized in [279] and consequently only one device driver is needed to support TPMs of different manufacturers.

Eavesdropping on the TPM communication bus can be useful to reconstruct the extension operations leading to a certain PCR value. As explained in Section 2.1.1 the content of a PCR is the result of a cryptographic hash function and it is computationally infeasible to determine the input of the extension operation because of the one-wayness property of a cryptographic hash function. In practice, eavesdropping of the TPM_Extend commands is not necessary because the measurements that are extended in a PCR register, are normally also logged in the so-called SML. In the case of a PC, this event log is stored as an ACPI table.

The security implications of this attack scenario are rather limited. Authorization data is never sent in the clear, so it will not be exposed by eavesdropping the trusted platform communication. As explained in Section 2.1.2, the proof of knowledge of authorization data is done with a cryptographic challenge-response protocol, which protects against replay and man-in-the-middle attacks but not against offline dictionary attacks on low-entropy passwords. Furthermore, authorization data is always encrypted when inserted or modified in the TPM. During the TPM_TakeOwnership command the authorization data of the TPM owner and of the SRK are encrypted with the public EK. When these secrets are changed (with TPM_ChangeAuthOwner), the new authorization data is encrypted with (a secret derived from) the current owner password. Similarly, when the authorization data of a key in the TPM's key hierarchy is inserted or modified (with TPM_ChangeAuth), the data is encrypted with a shared secret derived from the authorization secret of its parent key.

Passive monitoring of TPM communication mainly poses a concern when an external entity stores secrets in a remote TPM that is under attack of the local user. This could for instance be the case in a DRM scheme where a secret key is sealed to a certain virtual machine that is not under control of a local user. The secret key can be intercepted on the LPC bus when it is inserted in the TPM (with TPM_Seal) or released from the TPM (using TPM_Unseal). For this reason, the TPM 1.2 specifications define a new command that addresses this issue: TPM_SealX. The command behaves like TPM_Seal, but the input data of the command is encrypted with the same encryption scheme that is used to insert and change authorization data. It places information in the sealed blob such that the result of the TPM_Unseal operation will also be encrypted.

As described earlier, it is a common practice to set the SRK password to a well-known value, e.g., all-zeros in the case of Windows Vista and some open source tools. However, if this is the case, secret data still leaks when the data is inserted under the SRK (using TPM_SealX). The encryption key that is used to encrypt the TPM_SealX input, is derived from the known SRK authorization data, and hence it is also known to an attacker. In fact, Chen and Ryan made the same observation about the insertion of authorization data under a shared/well-known authorization secret in [55]. If authorization data is shared between users, it is advisable to seal/unseal secret data and insert/change authorization data in an encrypted transport session (see Section 3.2.4).

### 3.2.2 Reset Attacks

At startup of the platform or after a platform reset the TPM is initialized in two steps. A hardware-based signal triggers the TPM_Init command. This command sets an internal flag to indicate that the TPM is undergoing initialization and puts the TPM in a state where it waits for the command TPM_Startup. The TPM Interface Specification (TIS) recommends to use the reset line of the LPC bus as hardware trigger for TPM_Init [279]. Next, the S-CRTM issues the startup command, which can select three different modes: clear, save and deactivated. In a "clear" start all variables will go back to their default state. The "save" mode on the other hand informs the TPM to restore various values based on a prior TPM_SaveState and continue operation from the saved state. This mode will for instance be used to recover from standby, also known as Suspend to RAM. During a recovery from hibernation, also known as Suspend to Disk, or a complete reboot the S-CRTM will perform a "clear" start. In the "deactivated" mode the TPM is turned off and all subsequent command requests fail. The deactivated state can only be reset by performing another TPM_Init.

If an adversary has physical access to the platform, he can disconnect the LPC reset line from the TPM, either directly at the pin of the TPM-chip or at the connector of the TPM daughterboard. This allows for two types of attack: (1) the TPM can be reset without resetting the rest of the platform and (2) the platform can be reset without resetting the TPM. We call the first attack a *TPM reset attack* and the second a *platform reset attack*.

With the TPM reset attack an adversary can start the platform, physically reset the TPM, reinitialize the TPM by issuing TPM_Startup with the "clear" option, and record an arbitrary configuration in the PCRs that differs from the one that is really booted. The TPM can be reset with the LPC reset signal or by temporally grounding the supply voltage. The feasibility of this attack has been demonstrated by Kauer in [147] and Sparks in [250]. Both attacked a TPM that was mounted on a daughterboard, which made physical access to the reset line relatively easy. It is important to note that Sparks did not disconnect the TPM pin from the LPC reset line. Therefore other devices on the LPC bus, such as the keyboard and mouse controller and the fan controller, were also reset. However, the reset signal does not propagate to the CPU or devices on other busses, e.g., Peripheral Component Interconnect (PCI) bus. This signifies that the attack does not need to target the TPM reset pin directly, but it can also be performed by physical access to the reset pin of another chip on the LPC bus, e.g., the BIOS Flash chip or the Super I/O controller.

The platform reset attack enables an attacker to start the platform in a certain configuration (that gets recorded in the PCRs), physically disconnect the reset signal of the TPM and reboot the platform into a different configuration. If the platform is restarted with the hardware reset signal, this is often called a warm reboot or a soft reboot. Another approach is to connect the TPM to an external supply and to do a complete power cycle (turn off and then on); this is known as a cold reboot or hard reboot. During the reboot the S-CRTM will try to startup the TPM, but the TPM will return an error code because the TPM_Startup command was not preceded by TPM_Init. As a consequence the PCRs are not reset to their default value. It is not properly specified in [284] whether or not the S-CRTM will extend new measurements in the PCRs if TPM_Startup fails. If the measurements get extended for a second time, the static PCRs will contain values that are of no interest to the attacker. Nevertheless the PCRs used by the D-CRTM are left untouched, which can be exploited. If the static PCRs also retain their value, the static chain of trust can be subverted as well.

The latest TCG specifications have two features that in combination with the D-CRTM can be used as countermeasure against reset attacks: resetable PCRs and locality. Whereas in the 1.1b TPM specification [285] the PCRs can only be manipulated by the extension operation (see Section 2.1.1), the 1.2 version [281, 282, 283] defines a new command TPM_PCR_Reset which under certain conditions can reset the PCR value. The PC specific specifications [279, 284] define 16 static PCRs, which can only be extended, and 8 dynamic PCRs, which can also be reset. The 1.2 specification also introduces the concept of locality which allows the TPM to distinguish between different privilege levels. Certain dynamic PCRs can only be reset and extended by software/hardware with a high locality level. For instance, the D-CRTM, which has the highest locality level (i.e., locality 4), has exclusive access to PCR 17. Special LPC bus cycles that can only be generated by the chipset/CPU (see Section 3.3.2) are used to reset and extend the D-CRTM PCR. It is also possible to specify what the locality level must be in order to unseal certain data and to remotely attest the locality level with the TPM_Quote2 command.

In [147] Kauer argues that, in order to protect against a TPM reset attack, the S-CRTM must be abolished and the D-CRTM must be used instead. His key observation is that the dynamic PCR of the D-CRTM is reset to the value $-1$ during TPM_Startup and that only the D-CRTM can reset the register to the value 0 with the special LPC bus cycles. Therefore it is impossible for an attacker to reset the PCR to 0 with a hardware attack and extend "fake" measurements in the register. Kauer developed a proof-of-concept bootloader called OSLO that is measured by the D-CRTM provided by AMD SVM. The open source bootloader tboot[1] provides similar functionality for Intel TXT.

However, the dynamic chain of trust does not protect against the platform

[1] http://sourceforge.net/projects/tboot/

reset attack. When the platform reboots, only the static chain of trust will be measured. If an adversary loads an operating system that does not start the D-CRTM, the dynamic PCRs will still contain the measurements that were recorded before the reboot. This situation can particularly be exploited to forge a remote attestation. The platform configuration is measured during the boot process and the remote attestation protocol is executed afterwards, when network connectivity has been established. The timing of the platform reset is not so critical and, if needed, the attacker can even delay the TPM_Quote request by temporarily disabling the network connection. It is less straightforward to extract a sealed secret with a platform reset attack. Typically, a software component will extend measurements in a PCR until the register contains the desired platform configuration. Subsequently the software asks the TPM to reveal the secret data that is sealed under this configuration. It is common practice for the software component to extend another value in the PCR immediately after the TPM has unsealed the data. The extra PCR extension guarantees that other software components, which are loaded later in the boot process, cannot unseal the same secret data. Clearly this makes the timing of a platform reset more difficult: the platform must be rebooted after the first TPM_Extend operation(s) that record the correct platform state and before the TPM_Extend command that invalidates the PCR content.

### 3.2.3 Active Monitoring

The previous attack scenario disconnects the reset pin of the TPM from the reset wire of the LPC bus and actively manipulates the reset signal of the platform and the TPM. This type of hardware attack is reasonably easy to perform, because the timing of the reset signal can be done manually. A more powerful attack is the active monitoring of the four data wires of the LPC bus. This enables an attacker to intercept the commands that are sent to the TPM and selectively drop or modify them. Again the main objective of the attacker will be to fool the TPM about the platform status, and thus circumvent the sealed storage or remote attestation functionality.

Two attacks can be distinguished: (1) blocking TPM communication and (2) modifying TPM communication. The second type requires an electronic device that intercepts and decodes the TPM commands that are sent by the host platform, overwrites certain parameters, and transmits the modified commands to the TPM. Similarly the interposing device can also modify the response of the TPM. The monitor device can be realized with a microcontroller or with a FPGA. The delay introduced by the interposing device will typically not be detected by the host platform because a TPM is a slow device; some operations, such as the generation of signature, may take several hundred milliseconds. The blocking attack can be performed manually for instance by introducing switches on the LPC data lines.

The static chain of trust can be attacked by blocking the TPM communication at platform startup, for instance by (temporarily) disconnecting the LPC data wires from the TPM pins. The S-CRTM is unable to perform the TPM_Startup and integrity measurements cannot be extended in the PCRs. Afterwards, the LPC communication can be unblocked and malicious software can start up the TPM and record a fake platform configuration. This attack scenario is equivalent to the TPM reset attack.

The dynamic chain of trust can be attacked by modifying the TPM communication. The locality level is determined by the LPC address, and the chipset guarantees that only the D-CRTM instruction use the address for the highest locality level. However an interposing device can overwrite the LPC address and hence malicious software can assert the highest privilege level. This signifies that an attacker can reset the D-CRTM PCR and extend arbitrary values into the register. In 2011 Winter and Dietrich demonstrated the practical feasibility of this attack scenario in [299, 300].

The TCG authorization sessions (described in Section 2.1.2) are used to determine whether an entity is authorized to perform a function or use a certain TPM object, say a key, by proving knowledge of shared authorization data. The authorization protocol protects the integrity (and freshness) of TPM commands: a Message Authentication Code (MAC) value, keyed with the authorization secret, is computed over some input parameters of the command and appended to the command. However, the locality level is added to the TPM command at the level of the LPC interface. So the locality level is not included in the MAC computation and modification of the level remains undetected. Moreover, the TPM_Extend command does not support authorization sessions, so an interposing device can overwrite the parameters of this command, irrespectively whether locality is used.

### 3.2.4 Transport Session

As illustrated above, physical access to the TPM interface allows for some reasonably simple, yet powerful attacks. Sealed storage and remote attestation can be deceived by falsifying the content of the PCRs with a reset attack or with active manipulation of the TPM commands, and sealed data or authorization passwords may leak when transmitted over the LPC communication bus. The 1.2 version of the TPM specification introduces the notion of encrypted *transport sessions*, which can be utilized as a generic countermeasure against attacks on the TPM interface.

Transport sessions are initiated with the TPM_EstablishTransport command, which establishes a session key using a public storage key. The storage key must be loaded before the establishment of the transport session. An external entity can be assured that the storage key was created inside TPM with the TPM_CertifyKey command, which produces a signature with an identity key. A privacy CA can vouch that the identity key that is used to certify the storage key, belongs to a genuine TPM. Once the transport session is established, any TPM command can be wrapped in the session using the TPM_ExecuteTransport command. Transport sessions use a protection scheme similar to that of the TCG authorization sessions: rolling nonces, which are provided by the caller and the TPM, are used for freshness and a MAC value protects the integrity of the wrapped commands.

Three options can be selected when a transport session is established: (1) the session provides encryption, (2) it provides a log of all operations that occurred in the session, or (3) it is exclusive and any command executed outside the transport session causes the invalidation of the session. Typically the encryption is done by using the mask generation function MGF1 from PKCS #1 as a stream cipher; the output of the function is XORed to the plaintext messages. The MGF1 function uses the SHA-1 hash function and is seeded with the established shared secret and the rolling nonces. Optionally the TPM can support a proper symmetric encryption algorithm such as AES in Counter (CTR) or Output Feedback (OFB) mode. If session logging is enabled, the TPM_ReleaseTransportSigned command ends the transport session and returns a signed log of all operations performed during the session. For every wrapped command the log contains a digest of the input parameters, a digest of the output parameters, the current tick counter value and the locality level that called TPM_ExecuteTransport.

An encrypted transport session typically serves as a confidential and authenticated end-to-end channel between a remote entity and the TPM. Clearly it can also be used to protect against passive eavesdropping of the LPC bus. Data can be sealed and unsealed securely in an encrypted transport session and authorization data can be inserted and modified safely, without the risk of the secrets leaking on the communication interface. However, the establishment of an encrypted transport session introduces a considerable overhead. A random session key must be generated and it must be encrypted with a public storage key of the TPM. It will be rather cumbersome to use an encrypted transport session early in the boot process. Rivest Shamir Adleman (RSA) public key encryption must be added to a least the S-CRTM and the D-CRTM, which adds complexity. It is not immediately clear whether every software component in the boot process needs to establish a transport session on its own or whether the session key can be passed along.

Transport sessions protect the confidentiality of TPM commands against a passive eavesdropper. However, this functionality cannot protect against an active monitoring attack that blocks certain commands, for instance a critical PCR extend operation. Likewise they cannot prohibit an adversary from sending additional commands outside the transport session. If the transport session is exclusive, the execution of commands outside the session will terminate the session. Consequently the execution of a malicious command can be detected by making the transport session exclusive, but it cannot be prevented.

### 3.2.5 LPC Bus Encryption

Transport sessions work at the level of TPM commands and require public key cryptography to establish a session key. This stems from the fact that the feature is designed for remote entities to establish a secure channel to a distant TPM. As motivated above, this mechanism is less suitable as countermeasure against a local hardware attack on the TPM communication interface. From a theoretical viewpoint a much simpler solution would be the encrypt and authenticate the low-level LPC bus cycles with a low-latency symmetric cipher [34, 153]. However, there are some practical hurdles with this approach.

In order to have a transparent and manageable solution, the southbridge chip which drives the LPC bus, will only encrypt the LPC bus cycles that are addressed to the TPM. This signifies that the TPM communication is only protected on the slow LPC bus, but not on the high speed busses between the southbridge and northbridge and between the CPU and the northbridge, which is known as the Front Side Bus (FSB). The underlying motivation is that these other busses are more difficult to eavesdrop, let only manipulate, because of their very high bandwidth and that it is hard to gain physical access to them.

The southbridge chip and the TPM must share a symmetric key that has to be inserted in both devices during assembly of the motherboard. This signifies that the southbridge chip has to contain a bit of Non-Volatile Memory (NVM) to store the bus encryption key (see Section 4.2.6). Moreover, in order to protect against a replay attack the TPM communication must be protected with randomly generated nonces. The addition of NVM, a Random Number Generator (RNG) and perhaps a cryptographic engine signifies a higher hardware cost for the southbridge chip. The programming of the shared bus encryption key introduces extra overhead during the assembly process of the motherboard.

Data transfers on the LPC bus are serialized over four data wires: information such as cycle type and direction, address and data, are transferred serially. The LPC bus is shared by the TPM and other peripheral devices. All devices monitor the bus and determine based on the cycle type, direction and address to which device the communication is destined. Therefore, only the data field of the bus cycle may be manipulated by the bus encryption engine. It seems advisable to not only protect the confidentiality of the low-level TPM communication, but also its integrity and freshness. Hence, nonces must be provided by the southbrigde and the TPM and a MAC must be calculated over the TPM command and the fresh nonces. The nonces and the MACs must also be transmitted over the LPC bus.

As described above, the locality level is determined by the LPC address. Therefore it is paramount that the address field of the LPC bus cycles is also included in the calculation of the MAC. Otherwise, an interposing device can still assert the locality level of the D-CRTM and assist malicious software to manipulate the content of the PCR that normally can only be accessed by the D-CRTM.

### 3.2.6 Integrated TPM

Another approach to mitigate hardware attacks on the TPM communication interface is the integration of the TPM into another device. This approach is used in practice, but primarily as a cost saving technique, because there is no longer a need to add a separate TPM chip on the motherboard.

An integrated TPM has both advantages and disadvantages from a security perspective. If the pins of the chip in which the TPM functionality is integrated, are more difficult to access physically, the hardware attacks on the communication bus become more difficult. On the other hand, an integrated TPM might be less secure against invasive attacks, because some hardware countermeasures such as protective shielding might be too expensive to be applied on a bigger integrated circuit. Similarly an independent security evaluation is probably also more costly, because the influence of the other functionality of the device on the TPM functionality has to be assessed.

Table 3.1 gives an overview of all TPM products that are currently available or have been in the past; at least these TPM implementations have been listed on the website of the respective manufacturers. Some chips, in particular those implementing the 1.1b version of the TPM specification, are out of production. There are three manufacturers that have integrated the TPM functionality into another device.

National Semiconductor integrated a TPM into a Super I/O controller, which is connected to the LPC bus anyway. Most discrete TPM chips use a 28-pin TSSOP package, which has pins that are spaced 0.65 mm apart, where as the National Semiconductor integrated TPMs use a 128-pin PQFP or 100-pin

Table 3.1: Overview of commercial TPM products.

| manufacturer | product number | version | interface | integrated | EK cert |
|---|---|---|---|---|---|
| Atmel | AT97SC3201 | 1.1b | LPC | no | no |
| Atmel | AT97SC3201S | 1.1b | SMBus | no | no |
| Atmel | AT97SC3202 | 1.2 | LPC | no | no |
| Atmel | AT97SC3203 | 1.2 | LPC | no | no |
| Atmel | AT97SC3203S | 1.2 | SMBus | no | no |
| Atmel | AT97SC3204 | 1.2 | LPC | no | no |
| Atmel | AT97SC3204T | 1.2 | I2C | no | no |
| Broadcom | BCM5751M | 1.1b | LPC | GE[a] | no |
| Broadcom | BCM5752 | 1.2 | LPC | GE | no |
| Broadcom | BCM5752M | 1.2 | LPC | GE | no |
| Infineon | SLD 9630 TT 1.1 | 1.1b | LPC | no | yes |
| Infineon | SLB 9635 TT 1.2 | 1.2 | LPC | no | yes |
| Intel | iTPM | 1.2 | FSB | chipset | no |
| National[b] | PC21100 | 1.1 | LPC | no | no |
| National | PC8374T | 1.1b | LPC | Super I/O | no |
| National | PC8392T | 1.1b | LPC | Super I/O | no |
| National | PC8374S | 1.2 | LPC | Super I/O | no |
| Winbond | WPCT200 | 1.2 | LPC | no | no |
| Winbond | WPCT210 | 1.2 | LPC | no | no |
| Winbond | WPCT300 | 1.2 | SPI | no | no |
| Winbond | WPCT301 | 1.2 | I2C | no | no |
| Nuvoton | NPCT42x | 1.2 | LPC | no | optional |
| Nuvoton | NPCT50x | 1.2 | I2C | no | optional |
| Sinosun | SSX35 | 1.2 | LPC | no | no |
| STM[c] | ST19WP18-TPM | 1.2 | LPC | no | no |
| STM | ST19NP18-TPM | 1.2 | LPC | no | no |
| STM | ST19NP18-TPM-I2C | 1.2 | I2C | no | no |
| STM | ST33TPM12I2C | 1.2 | I2C | no | yes |
| STM | ST33TPM12LPC | 1.2 | LPC | no | yes |
| STM | ST33TPM12SPI | 1.2 | SPI | no | yes |

[a] Gigabit Ethernet controller

[b] National Semiconductor

[c] STMicroelectronics

LQFP package with a pin spacing of 0.50 mm. Physical access to the pins is a bit more difficult, but not that much. Active and passive monitoring of the LPC bus is still possible. A TPM reset attack will reset the entire Super I/O controller, but this is not a big issue. The attacker just has to reinitialize the Super I/O controller as well. In fact, Sparks [250] did not disconnect the TPM reset pin from the LPC bus. So in his experiments the Super I/O controller got reset as well.

Broadcom on the other hand includes the TPM functionality into Gigabit Ethernet controllers and these chips use a BGA package. The Ethernet component of the chip is accessed by the platform through the PCI Express bus and the TPM part is connected to the LPC bus. The pins of a Ball Grid Array (BGA) package are located underneath the package and hence they are more difficult to access physically. It is therefore nearly impossible to insert an interposing device between the southbridge and the Broadcom TPM. However, if another device is connected to the LPC bus as well (e.g., BIOS Flash or Super I/O controller), it might still be possible to eavesdrop the LPC bus at the pins of the other device and perform a TPM reset attack.

Intel has adopted a more radical approach by integrating the TPM into the MCH, which is the northbridge of the chipset. This means that the TPM is accessed directly through the very high speed FSB between the CPU and the northbridge, instead of the slow LPC bus. Consequently it is more difficult to monitor the TPM communication. For instance, the Intel Q45 chipset, which has the integrated TPM (iTPM), supports a 1333/1066/833 MHz FSB and has a Flip Chip BGA package with 1254 solder balls. Reset attacks are also infeasible because it is impossible to decouple the TPM reset from the platform reset. If the northbridge chip is reset, not only the TPM functionality is reset but also the memory controller of the platform. There are some design challenges to the Intel solution. The MCH chip lacks internal non-volatile memory to store the TPM firmware and persistent state, and hence it has to rely on the external SPI Flash chip that stored the BIOS image. The topic of non-volatile state protection is the subject of Chapter 4. As mentioned earlier, certification and security evaluation of an integrated TPM is more difficult than with a discrete TPM. In the case of the Intel iTPM it is more challenging, because the embedded microprocessor inside the MCH that runs the TPM firmware, also runs the AMT remote management functionality [175].

## 3.3 Experimental Results

We did our experiments in the middle of 2005 on one of the first desktop PCs that shipped with a TPM. In particular we used an IBM ThinkCentre M50, which had an Atmel 1.1b TPM. The TPM is installed on a daughterboard and therefore its communication interface is easily accessible. Furthermore, we chose an IBM machine, because in those days most of the open source support of TCG technology was developed by IBM. The Linux device drivers for the most TPMs and the open source TSS called TrouSerS have been developed by IBM's Linux Technology Center. In 2008 IBM employees wrote a book on practical implementation aspects of trusted computing technology [51].

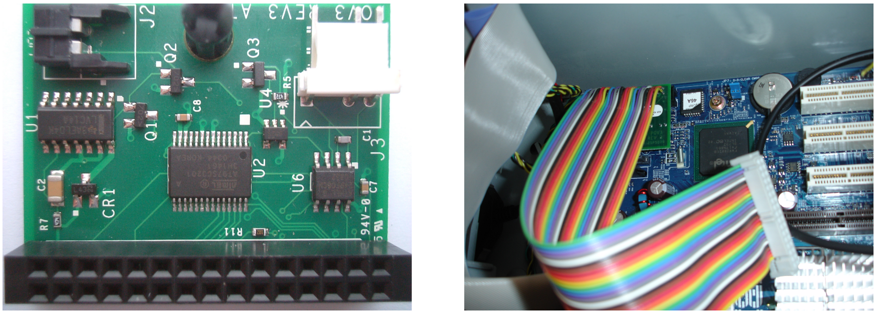
When we did our analysis, the pin layout of the Atmel TPM was not properly documented. The meaning of all pins was only defined in a later version of the datasheet. We had to do some reverse engineering in order to determine how the TPM pins were connected to the connector of the daughterboard and which pins corresponded with signals of the LPC bus. The reverse engineering of the daughterboard will be presented in Section 3.3.1.

The 1.1b specification does not specify the low-level TPM interface. As a result the method to transfer the TPM commands over the LPC bus differs slightly for every TPM manufacturer, requiring a different device driver. The interface of the Atmel 1.1b TPM is specified in documentation that is not publicly available. However, we were able to determine the interface from the open source Linux device driver, which was written by IBM people who had access to this documentation, and by analyzing the communication on the LPC bus. The interface of the Atmel TPM will be described in Section 3.3.2.

The situation has changed with the 1.2 TIS, which rigorously standardized the low-level interface [279]. Consequently a single device driver can support any 1.2 TPM that implements the TIS interface. This is exactly the reason why Microsoft Vista only supports 1.2 TPMs, even though Vista only uses TPM 1.1b features (e.g., for the full disk encryption scheme called Bitlocker).

### 3.3.1 Reverse Engineering of TPM Daughterboard

Some effort was needed to determine the electrical specification and function of all signals on the TPM daughterboard. Figure 3.2(a) shows the front side of the daughterboard, i.e., the side facing the motherboard. The chip that is denoted with U2 on the circuit board, is the Atmel AT97SC3201 TPM. All communication to the motherboard happens over the 30 pin connector at the



(a) Front side view of daughterboard.

(b) Wire connected to daughterboard.

Figure 3.2: TPM daughterboard of IBM ThinkCentre M50.

bottom. In order to simplify physical access to the LPC bus we soldered a wire onto the daughterboard connector, as depicted in Figure 3.2(b).

We measured the voltage level on the pins of the daughterboard connector when the platform was powered off and powered on. This enabled us to determine which pins correspond with the ground level and the supply voltage. We noticed that one of the pins is high even if the PC is turned off and concluded that this pin is connected to the CMOS battery, which powers the computer's Real-Time Clock (RTC). The Atmel datasheet mentions that the TPM senses the battery to detect if it is removed from the PC. However, we believe that it is possible to (temporally) remove the daughterboard by connecting the pin to an external battery.

With an oscilloscope and a logical analyzer we were able to find the pins that are connected to the LPC bus and determine the meaning of the different signals. The main challenge was to identify the assignment of the four data wires of the bus. We solved this by modifying the Linux TPM device driver to display which bytes are transferred over the LPC bus to/from the TPM. By comparing this information with what we measured on the four wires, we were able to determine the order of the signals.

With the modified device driver and the logical analyzer we were also able to understand how the TPM commands and responses are transferred over the LPC bus in the case of the Atmel TPM (see Section 3.3.2). At that stage we were able to eavesdrop the TPM communication (see Section 3.3.3).

We accidentally damaged the motherboard of the PC, presumably by short-circuiting some pins on the daughterboard connector. Because the computer was unusable at that stage, we removed the daughtboard from the PC and

Table 3.2: Interconnection of Atmel AT97SC3201 and daughterboard connector.

| Pin Name | Description | TPM Pin | Connector Pin |
|---|---|---|---|
| GND | Ground | 12 | 1 |
| LAD[3:0] | LPC Command, Address & Data | 24,23,26,25 | 9,22,8,23 |
| LFRAME# | LPC Frame Indicator | 2 | 28 |
| LCLK | LPC Clock (33 MHz) | 28 | 29 |
| LRESET# | System Reset (active low) | 1 | 5 |
| SIRQ | Serialized IRQ | 27 | 11 |
| VBB | Battery Input | 14 | 16 |
| VCC | 3.3V Supply Voltage | 13 | 18 |
| CLKRUN# | Clock Control | 7 | GND |
| IOA | Input/Output A (SMBus clock) | 3 | GND |
| IOB | Input/Output B (SMBus data) | 6 | unknown |
| IOC | Input/Output C (GPIO pin) | 22 | GND |
| Xtamper | External Tamper Detect | 21 | GND |
| Xtal1 | 32.768 kHz Crystal | 9 | GND |
| Xtal2 | 32.768 kHz Crystal | 8 | unknown |

measured with a multimeter how all pins of the Atmel TPM are wired to the connector of the daughterboard.

Table 3.2 describes the interconnection of the TPM pins and the connector pins. We start numbering the connector pins as viewed on Figure 3.2(a) from the left lower corner (this is pin 1), and continue counter clockwise (thus the upper left pin is pin 30). For the pins of the TPM we follow the notation used in the AT97SC3201 datasheet; on Figure 3.2(a) the pin in the right upper corner is pin 1 and the numbering is also counter clockwise. The name and functional description of the TPM pins is also extracted from the Atmel datasheet. As mentioned before, the earlier version of the datasheet, which was available when we started our experiments, did not contain this information, but we were still able to determine the location of the pins of interest of the LPC bus.

Some signals are connected to the ground level, which originates from connector pin 1, because their functionality is unused. The CLKRUN# signal can normally be used to stop the LPC bus clock; this is useful in a mobile computer as power saving mechanism (e.g., during a Suspend to RAM). The two-wire SMBus interface, with IOA the clock and IOB the data signal, provides an alternative to the LPC bus interface. The external tamper detection signal Xtamper can presumably notify the TPM to take certain countermeasures. An external

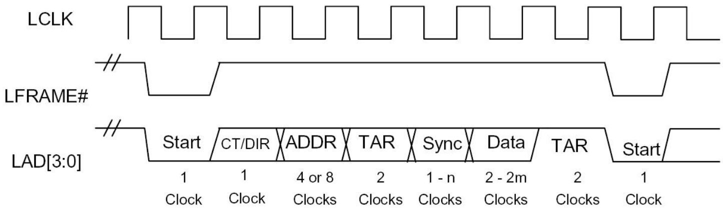


Figure 3.3: Typical LPC bus timing. LFRAME# is the frame indicator, LCLK the clock and LAD[3:0] the address and data wires. CT/DIR indicates the cycle type and direction and TAR denotes turn-around.

crystal can be used by the TPM in combination with the motherboard battery to generate the RTC. If this crystal is connected to the TPM, the signal is also used (much like VBB) to detect if the TPM has been removed from the platform.

### 3.3.2 Low Pin Count Bus

As explained earlier, the initial TCG PC client specification did not standardize the low-level TPM interface. According to the datasheet, the Atmel 1.1b TPM supports both the LPC bus and the SMBus as transfer mechanism. From the analysis at the daughterboard it was clear that the LPC bus is used in the IBM machine. Nowadays Atmel produces a 1.2 TPM with an LPC-based TIS-interface for PCs, but also one with a 100 kHz SMBus two-wire interface that is targeted at embedded devices (see Table 3.1). In this section we explain in more detail the proprietary LPC interface of the Atmel 1.1b TPM.

The LPC specification [137] was developed by Intel for ISA-less PCs. It offers a cost-effective and easy bus, with only seven mandatory signals namely LAD[3:0], LFRAME#, LRESET# and LCLK, and some optional signals, such as CLKRUN# and SIRQ. The LPC bus uses the 33 MHz clock of the PCI bus as clock LCLK, and allows various transfer protocols (memory, I/O, DMA, firmware memory and bus master).

Data transfers on the LPC bus are serialized over a 4-bit bus. The frame indicator signal LFRAME# is used by the host to start or stop transfers, and by peripherals to determine when to monitor the bus for a cycle. The LAD[3:0] bus communicates address, control, and data information serially. The general flow of cycles is visualized in Figure 3.3 and goes as follows:

1. A cycle is started by the host when it drives LFRAME# active, and puts the appropriate start value on the LAD[3:0] signal lines.

2. The host sends information regarding the cycle, such as the cycle type and direction (denoted with CT/DIR in Figure 3.3), and the address of the peripheral.

3. The host optionally sends data, and turns the bus around to monitor the peripheral for the completion of the cycle; the turn-around is denoted with TAR in Figure 3.3.

4. The peripheral can assert a number of wait states to synchronize and indicates the completion of the cycle by sending the appropriate ready value on the `LAD[3:0]` signal lines. Optionally the peripheral sends response data.

5. Finally, the peripheral turns the bus around to the host, ending the cycle.

By studying the Linux device driver and the LPC communication captured with the logic analyzer, we determined that LPC I/O cycles are used by the Atmel TPM: I/O Write cycles are used to send requests and I/O Read cycles to get back the responses. The 1.2 TIS interface also uses I/O cycles for legacy software, but defines two new special cycles (TPM-Write and TPM-Read) to assert a locality level. The LPC locality cycles are identical to the standard I/O cycles with the exception of the START field, which is set to a previously reserved value.

Table 3.3 gives the definition of all I/O cycle fields. The exact sequence of the fields for an I/O Read cycle matches the generic timing of Figure 3.3. For I/O cycles the address field is 16 bits (so 4 clock cycles), and the data returned is 8 bits (thus 2 clocks). A write cycle is very similar: after the address is transferred, one data byte is sent, and the bus is handed over to the TPM, that will add wait states until it is ready to turn around the bus.

The value of `LAD[3:0]` is given in hexadecimal format, and the 16-bit I/O address (represented with a 4-digit hex code) is transferred with the most significant nibble first. During the turn-around time `LAD[3:0]` will be driven to `0xF` during the first clock cycle, and tri-state during the second clock cycle. The TPM needs to assert wait states, and does so by driving `0x6` (i.e., Long Wait) on the bus until it is ready (so $n-1$ clock cycles); when ready, it drives `0x0` during 1 clock cycle.

The two I/O addresses in Table 3.3 are specific to the Atmel TPM, and the procedure to send and receive TPM datagrams over the LPC bus is also proprietary. According to comments in the Linux driver, the Atmel chip does not support interrupts. This is strange because the public datasheet list the `SIRQ` signal, which suggests that interrupts are supported. Anyway, the Atmel 1.1b TPM has a port mapped interface, with a data port and a status

Table 3.3: LPC I/O cycle field definitions.

| Field | # Clocks | Description | `LAD[3:0]` |
|---|---|---|---|
| START | 1 | *Start of Cycle* | `0x0` |
| CT/DIR | 1 | *Cycle Type/Direction*<br>I/O Read<br>I/O Write | <br>`0x0`<br>`0x2` |
| ADDR | 4 | *Address*<br>TPM data port (Atmel specific)<br>TPM status port (Atmel specific) | <br>`0xE000`<br>`0xE001` |
| TAR | 2 | *Turn-Around* | `0xFF` |
| SYNC | $n$ | *Synchronize*<br>Ready<br>Long Wait | <br>`0x0`<br>`0x6` |
| DATA | 2 | *Data byte*<br>least significant nibble first<br>most significant nibble next | <br>`Data[3:0]`<br>`Data[7:4]` |

port. The status of the TPM must be determined by polling its status port and the transfer protocol goes roughly as follows:

1. A TPM command is transferred using multiple I/O Writes to the TPM's data port.

2. The device driver will repeatedly check the status, using I/O Reads of the status port, to see if the TPM is still busy and to determine when data is available.

3. Once data is available, the TPM response can be read byte by byte using I/O Read cycles from the TPM's data port. As the driver does not know the size of the response, it will always check the status of the TPM to determine if data is still available, before reading the next byte.

4. The transfer is completed whenever no data is left.

Note that Chapter 4 of the IBM book on trusted computing [51] describes how to write a TPM device driver and gives as example the driver of the Atmel 1.1b TPM.

### 3.3.3 Analysis of Trusted Platform Communication

For our analysis of the LPC interface, we used the TPM_GetCapability command, which queries the TPM for certain information including the manufacturer, the version, the supported key size and algorithms. This command is available when TPM ownership has not been taken and thus it was a natural choice for a first analysis. We logged in on the computer remotely in order to be assured that there is no activity from the keyboard controller on the LPC bus. An Agilent 16900 Logic Analysis System was used to analyze the communication on the LPC bus.

Once we had figured out the meaning of the different signals of the LPC bus, in particular the order of `LAD[3:0]`, and the transfer protocol that is used, we were able to log arbitrary TPM commands and their respective responses on the LPC bus.

Next we wanted to recorded the TPM commands during the startup of the PC. We were interested to see whether the S-CRTM calculates the SHA-1 hash over the BIOS image in software or whether it relies on the comparatively slow TPM to do so. In our previous experiments we were sure that the TPM was the only device communicating on the LPC bus, but during the boot process a lot of additional traffic is generated on the LPC bus. The S-CRTM and the BIOS image are read from the BIOS Flash over the same bus.

The Agilent logic analyzer can trigger on the first packet to the TPM, based on a trigger sequence that includes the LPC address, and fill its memory from that moment in time. However, after the S-CRTM has issued the TPM_Startup command, it loads the BIOS firmware over the LPC bus such that its integrity can be measured. This fills up the acquisition memory of the logic analyzer, and the subsequent TPM packets cannot be captured. The logical analyzer has no feature to only log TPM packets, so selectively store datagrams. It is possible though, but not really practical, to startup the PC many times, and use a different trigger condition to ultimately capture all LPC traffic during startup; post-processing of this data is required to filter out the relevant TPM communication.

We considered two options to efficiently address this practical limitation of the Agilent logic analyzer:

- Implement a dedicated logging device that only logs communication addressed to the TPM. Such device has to analyze the packets on the bus, and only store the ones from and to the TPM. The logged data has to be sent to a computer for further analysis.

- Insert a device between the LPC bus and the logic analyzer, which acts as a packet filter. This device will only forward packets to the logic analyzer if they satisfy certain criteria, namely that they have the TPM as destination.

Both solutions can be implemented in software, for instance with a microcontroller, or in hardware on an Field Programmable Gate Array (FPGA). We choose the second approach and implemented the filtering functionality on a Xilinx Virtex-II Pro. The idea was also that this would be a good starting point for active bus attacks, e.g., blocking or manipulation of TPM commands or reset attacks. However, because of a misconfiguration of the FPGA pins, we accidently damaged the PC motherboard and we had to stopped the experiment.

In [165] we described our experimental results and outlined some active attack scenarios. In this paper we suggested the LPC reset signal (`LRESET#`) as possible attack candidate, and the feasibility of this attack was confirmed later in [147, 250]. We also emphasized the potential of active manipulation on the LPC bus. In 2011 Winter and Dietrich demonstrated the first practical active attack on the TPM communication that requires a single wire of the LPC bus (i.e., `LFRAME#`) to be hijacked. They built a dedicated logging device, consisting of a microcontroller board and an FPGA board.

## 3.4 Side-Channel Attacks on TPMs

To date no independent security evaluation of commercial TPM products has been published, except for the advanced invasive attack on the Infineon TPM by Tarnovsky [110, 271]. Some compliance testing was performed on early TPM implementations and minor issues were identified by Sadeghi et al. [223], but the security implications of this analysis were fairly limited. It would be interesting to see how resistant the existing TPM products are against side-channel attacks.

The TPM implementations of Atmel, Infineon and STMicroelectronics are derived from secure microprocessors that are used in smart cards and other security applications. Hence these products will probably implement some countermeasures against side-channel attacks. Other products, such as the integrated TPM of Broadcom or Intel, might offer a lower security level because they originate from PC components which are traditionally optimized for performance and/or cost instead of security.

Different side-channel attacks can be applied to TPMs. The power consumption can be measured on the external supply pins. This can be done by connecting `GND` and `VCC` to an external power supply instead of the motherboard supply, and measuring the current flowing into the TPM. If physical access to the pins is hard, which is the case for integrated TPMs, the attacker might have to resort to the analysis of the EM radiation produced by the TPM. Another option is the non-invasive injection of faults (i.e., glitches) on the supply voltage or the clock input. However, many TPM manufacturers claim to have implemented defense mechanisms against this type of attack. For instance, the reverse engineering of the Infineon TPM by Tarnovsky has revealed that it generates an internal clock signal, and consequently glitches on the externally provided LPC clock do not influence the cryptographic operations.

Clearly, more work is needed to determine the effectiveness of side-channel attacks on commercial TPM implementations. In the remainder of this section we will highlight which TPM commands manipulate important secret data and hence are interesting candidates, albeit in theory, for a side-channel attack. In our discussion we make abstraction of the type of side channel attack that would be used in practice.

### 3.4.1 Attacking the Endorsement Key

If the TPM ships with an endorsement certificate, the extraction of the private part of the EK is an important security threat. Once the private EK is revealed, the key can be programmed into a software TPM and used to circumvent the remote attestation protocol. A privacy CA will believe that the software TPM is a genuine hardware TPM because it has a valid endorsement certificate. As a result the CA will issue certificates on the AIKs of the software clone. At that stage the software TPM can report arbitrary “fake” platform configurations.

Before ownership is taken, the public EK can be read with the TPM_ReadPubek command. Afterwards, the public portion of the EK and the SRK can be read with TPM_OwnerReadInternalPub, but this command requires authorization from the TPM owner. This does not really pose a problem, because an attacker can always clear the ownership and then read the public EK without any restriction. Note that the SRK is deleted if the ownership is cleared, so this technique has to be avoided when the SRK is the attack target (see Section 3.4.2).

There are only two commands that process the private part of the EK: TPM_TakeOwnership and TPM_ActivateIdentity. The first command takes as input the owner password and the SRK authorization data, encrypted with the public EK. It will decrypt the two passwords with the private EK and generate some secret persistent values including a new SRK. The public part of SRK is exported by this command. The TPM_ActivateIdentity command decrypts a blob which contains the symmetric session key that the privacy CA has used to encrypt an AIK certificate. This command requires the authorization of the

TPM owner and it must be preceded by the TPM_MakeIdentity which generates the AIK that the CA needs to certify.

If the side-channel attack requires the measurements of a large number of public key decryptions with the private EK, the second TPM command seems the best choice. Repetitively taking and clearing ownership of the TPM will be slow because the TPM_TakeOwnership command will every time generate a number of RSA key pairs. On the other hand, one TPM_MakeIdentity command may be followed by a number of TPM_ActivateIdentity operations, all with a different session key. If a side channel attack also needs leakage measurements of cryptographic operations with a known key, this can be done by decrypting data with a legacy key (see Section 2.1.2) using the TPM_Unbind operation. This is for instance a requirement for a template attack [53].

### 3.4.2 Attacking the Storage Root Key

As explained in Section 2.1.2, the TPM stores the SRK in internal non-volatile memory. All other keys protected by the TPM are stored externally albeit in an encrypted form. This makes the SRK, which forms the root of the TPM's key hierarchy, a critical asset. If a side-channel attack is successful in the extraction of the SRK, the protected storage of the TPM is completely compromised: all keys stored outside the TPM can be decrypted.

Because the SRK is a storage key, it can only be used as an encryption key and not as a signature key. This means that an attack should target an operation that uses the private SRK as a decryption key. A naive attack scenario would be to repetitively generate a key under the SRK with TPM_CreateWrapKey and load the newly created key with TPM_LoadKey2. However, this approach is inefficient because the generation of an RSA key pair is time consuming.

It is much easier to attack the TPM_Unseal operation. The TPM expects an encrypted blob that was created with a prior TPM_Seal operation. If a random ciphertext is provided, the TPM will decrypt it and subsequently validate the structure of the plaintext. The validation will fail and the TPM will return an error indicating that the provided input is not a sealed blob. However, at that stage the TPM has already used the private SRK for the decryption and the side-channel leakage during this operation can still be measured.

One potential hurdle is the fact that authorization data must be provided in order to make use of the SRK. In most practical applications, this authorization data is set to a well-known value, namely all-zeros. However, if the value is unknown, the decryption with the SRK will not be performed because the authorization protocol fails. Consequently, the attacker must first perform a side-channel attack on the SRK password, before he can attack the SRK itself. As described in Section 2.1.2, in order to determine whether the caller knows the SRK password, the TPM will calculate a MAC value on the parameters of the TPM command, keyed with the secret password, and verify whether it matches with the MAC value provided by the caller. Every time the attacker sends a command with an incorrect authorization digest, the TPM will calculate a MAC value and subsequently return an error indicating an authorization failure. The attacker can probably perform an attack on the MAC function because he knows all the inputs to the function except for the secret password. However, in practice the described attack is unlikely to succeed. TPMs implement schemes that will gradually slow down authorization after a number of failed attempts. Hence it will take an extremely long time before a sufficient number of leakage measurements can be taken.

## 3.5 Conclusion

In this chapter we have studied practical and theoretical hardware attacks on TCG compliant platforms. The two roots of trust that define a TCG compliant platform, are the CRTM, which starts the measured boot process, and the TPM, which is responsible for protected storage of sensitive data and for reporting of the platform state. It is clear that these two components are attractive attack targets. A compromise of the CRTM enables an attacker to subvert the chain of trust by recording false integrity measurements in the TPM. If the TPM is attacked directly, the endorsement key and the storage root key are the most interesting targets. An extraction of the former enables an attacker to report arbitrary platform configurations and a compromise of the later exposes all data that is protected by the TPM. In Section 3.4 we have discussed how these two keys can be theoretically attacked with a side-channel attack. The practical applicability of these attacks on commercial TPMs depends heavily on whether countermeasures have been taken in the RSA implementation.

We were the first to observe that the communication interface between the TPM and its host is literally and figuratively the weakest link in a TCG compliant platform. We demonstrated experimentally that the commands and responses can passively be monitored when they are sent over the LPC interface of an Atmel 1.1b TPM. In practice this attack scenario is useful to eavesdrop keys when they are unsealed by the TPM. We have also described two attack scenarios that exploit the fact that the host platform and the TPM can be reset independently. Finally we discussed the impact of other active attacks that manipulate the trusted platform communication.

The attacks on the TPM communication interface are fairly easy to perform using off-the-shelf lap equipment, yet they have severe security implications. In this chapter we have discussed various countermeasures, such as locality and transport sessions, that are present in the TPM specification. The most effective way however to resolve these attacks is the integration of the TPM in another hardware component with a communication interface that is more difficult to analyze. Ideally the TPM should be integrated in the same chip that acts as CRTM, namely the CPU. This topic will be covered in the Chapter 4, where we will study how to embed the TPM in a hardware component that lacks reprogrammable non-volatile memory.

# Chapter 4

# Non-Volatile State Protection

The integration of a TPM into another hardware component can be advantageous over an implementation as a discrete chip. On the one hand, hardware attacks on the communication interface of an integrated TPM could be more difficult to perform. On the other hand, the integration could mean a cost saving. One of the key problems when integrating a trusted module into another hardware component or an embedded system-on-chip design, is the lack of on-chip Multiple-Time Programmable (MTP) Non-Volatile Memory (NVM).

In this chapter, we investigate how the non-volatile state of a trusted module can be protected in external non-volatile memory. The approach based on authenticated external NVM has been presented in [228] and the approach based on a reconfigurable PUF in [163].

## 4.1 Introduction

As illustrated in Chapter 3, in the past TPMs were often implemented as a discrete cryptographic chip that is physically bound to the rest of platform by soldering it to the platform's mainboard. This meant that during the initial deployment of TCG technology the TPM was typically added as an optional component, only available in some PC/laptop configurations. Later, the TPM functionality got integrated in other peripheral components, such as a network controller, and it became more widespread. In 2008 Intel introduced a TPM that is integrated into the MCH, a core hardware component of the PC platform, also referred to as the northbridge of the chipset. The integration of the TPM into another hardware component signifies a lower bill of materials. However, this integration has an impact on the security of the trusted computing platform as well. On the one hand, it makes the TPM less vulnerable to attacks on the communication interface, because physical access to the interface becomes cumbersome. On the other hand, the security evaluation of an integrated TPM may be more difficult because the TPM functionality is part of bigger component. Additionally, it may be harder to implement countermeasures against physical attacks, such as shielding or special logic styles.

### 4.1.1 Mobile Trusted Module

The MTM specification abstracts a trusted mobile platform as a set of trusted engines, each acting on behalf of a different stakeholder. For a mobile phone a number of stakeholders can be identified: the device manufacturer, the cellular network operator, the application providers and the user. Every trusted engine will have its own MTM. As most stakeholders do not have physical access to the device, a distinction is made between Mobile Remote Owner Trusted Modules (MRTMs) and Mobile Local Owner Trusted Modules (MLTMs). MRTMs have a pre-installed remote owner, cannot be disabled by the local user and need to support secure boot [73]. MLTMs are more like a regular TPM, except optimized for embedded device. For an in-depth summary of the MTM specification we refer to Ekberg and Kylänpää [91].

Some proof-of-concept implementations of the MTM specification have been presented by academic research groups as well as by the research centers of Nokia and Samsung. However, mobile phone manufacturers have not yet expressed the intention to incorporate MTMs in commercial products.

In the scientific community a number of approaches has been proposed to implement the MTM specification. We briefly describe them here. For a more detailed description of the different approaches we refer the reader to [116].

A first approach is to implement the MTM in hardware like a traditional TPM, either as a separate dedicated chip or integrated into another hardware component. Researchers of the Korean Electronics and Telecommunications Research Institute demonstrated the feasibility of this approach by making a discrete MTM chip in 0.18 $\mu$m CMOS technology [150].

A second implementation option is to run the MTM as software in a programmable execution environment that is isolated from the – potentially untrustworthy – main operating system. The software MTM can run on a secondary microprocessor that is physically isolated from the main application processor. Examples of secondary processor in a mobile phone are the embedded

Secure Element (SE) of a Near Field Communication (NFC)-enabled phone, a secure microSD (e.g., provided by G&D Secure Flash Solutions) or even the SIM card. Dietrich demonstrated the implementation of the MTM specification on a JavaCard-based SE [72].

Alternatively, the software MTM can run in the so-called Trusted Execution Environment (TrEE) [158] of the main application processor. This environment is logically isolated from the rest of the platform. Typical examples of TrEE technologies are ARM TrustZone [177, 296] and TI M-Shield [13]. Proof-of-concept MTM implementations executing in a TrEE have been presented in [74, 90, 297].

A final implementation option, which has been proposed in [24, 231, 234, 319], is to run multiple software/virtual MTMs in separate environments that are isolated from each other and from the main operating system by a microkernel or hypervisor.

### 4.1.2 Embedded Trusted Computing

In this chapter we investigate the integration of the TPM/MTM functionality into existing platform components. As stated earlier, this integration is desirable, because a discrete trusted module increases the cost of the platform. Sometimes an embedded trusted module is also less prone to attacks on its communication interface to the CRTM component, either because the bandwidth of the interface is very high or because the interface is not accessible except with an invasive attack. The former applies for instance to Intel's iTPM. The latter is the case when the trusted module is integrated in the same chip as the platform's main processor.

We focus primarily on the protection of the trusted module's *non-volatile state* or *persistent state*, when the module is embedded into a component that lacks *on-chip* MTP NVM. The persistent state includes cryptographic keys, authorization data and *monotonic counters*. There are two main reasons why embedded NVM is often not available. First, traditional floating gate-based MTP NVM technologies, such as Flash memory and EEPROM, are too expensive because they require additional masks and processing steps relative to a standard Complementary Metal Oxide Semiconductor (CMOS) logic process. Second, logic NVM memory solutions that are compatible with CMOS [69], as offered by IP providers such as eMemory, Kilopass, Novocell Semiconductor, NSCore, Sidense and Synopsys (formerly Virage Logic), are still relatively new and hence not (yet) widely deployed.

We see three important scenarios where the trusted module can be integrated into a hardware component which lacks on-chip NVM: the integration of the TPM functionality into the chipset of a PC, the implementation of the MTM specification on the application processor of a mobile device, and the implementation of a trusted module on an FPGA. These are now discussed in more detail.

### Integration of TPM into Chipset

Intel has built a secondary microprocessor, known as the Management Engine (ME), into business PCs with vPro technology. The ME processor is embedded in the MCH of the chipset and it is traditionally used to remotely manage the PC; this feature is known as Intel AMT. However, in the latest generation of the vPro platform, the ME also runs the integrated TPM (iTPM) functionality.

The MCH is made in the latest CMOS technology and it does not contain Flash memory for cost reasons. This means that the firmware and the non-volatile state of the ME – and consequently of the iTPM – have to be stored externally in the Serial Peripheral Interface (SPI) Flash memory which also stores the BIOS image.

### MTM on Application Processor

For quite some time *application processors* for smart phones and tablets have had security features such as a cryptographic accelerator and internal Static Random Access Memory (SRAM) and ROM for secure boot [158]. Modern high-end application processors also support ARM TrustZone and TI M-Shield, which can be used to run security critical tasks in a general-purpose Trusted Execution Environment (TrEE). This TrEE is isolated from the other software of the platform by low-cost hardware extensions and minimal software support. As demonstrated in [74, 90, 297] this isolated environment is ideally suited to run a software MTM.

However, in general the application System-on-Chip (SoC) of a mobile phone lacks embedded reprogrammable NVM. It will have some One-Time Programmable (OTP) non-volatile memory, which can be programmed during manufacturing (e.g., laser fuses) and/or in the field (i.e., electrical programmable fuses or eFuses). This memory is typically used for secure key storage.

The lack of on-chip MTP NVM implies that the MTM's firmware and non-volatile state must be stored in external NVM, albeit in a secured manner. A protection scheme is necessary to guarantee the confidentiality, integrity and freshness of the externalized firmware and state.

### Trusted Module on FPGA

A third scenario is the implementation of a trusted module on reconfigurable hardware, such as a Field Programmable Gate Array (FPGA). Reconfigurable hardware facilitates field upgrades of the trusted module implementation. The firmware of the module can be update to fix non-compliance bugs [223] or potentially support future versions of the TPM specification. Additionally the hardware can be upgraded for instance to replace broken/depreciated cryptographic algorithms (e.g., the SHA-1 hash function).

The majority of the commercial FPGAs are volatile, which means that the *bitstream* that configures the reconfigurable logic, is stored in external NVM when the FPGA is powered down. High-end volatile FPGAs support bitstream encryption, which works in the following manner: the configuration bitstream is stored in the external NVM in encrypted form and is passed through a hard-wired decryption engine before it is loaded on the reconfigurable logic. For on-chip key storage FPGA manufacturers rely on battery-backed SRAM or on OTP fuses. The bitstream encryption functionally only protects against cloning and reverse engineering of the bitstream. An additional protection scheme is necessary to protect against downgrading of the bitstream to an older version and to secure user data, such as the persistent state of a security module, in the external NVM.

We will look in more detail at reconfigurable trusted computing in Chapter 6.

## 4.1.3 Non-Volatile State

In Section 2.1.1 we gave a detailed description of the trusted modules that have been specified by the TCG, namely the TPM and the MTM. We briefly recapitulate the state information that these TCG modules contain. A distinction can be made between a volatile part, which is cleared when the platform and thus the module are reset, and a persistent part, which has to survive power cycles and consequently must be stored in non-volatile memory.

On the one hand, the trusted module needs volatile memory for temporary data. This includes key slots to load keys stored outside the trusted module, a number of Platform Configuration Registers (PCRs) that store measurements (i.e., hash values) made during startup of the platform, and information (e.g., nonces) about the concurrent authorization sessions.

On the other hand, some information maintained by the trusted module has to be stored persistently. For the TPM, this includes the EK that uniquely identifies each TPM, the SRK that encrypts other keys maintained by the TPM,

Table 4.1: Monotonic counters in MTM and TPM.

| type | size (bits) | increment command |
|---|---|---|
| counterBootstrap | 5 | MTM_IncrementBootstrapCounter |
| counterRIMProtect | 12 | TPM_IncrementCounter |
| counterStorageProtect | 12 | TPM_IncrementCounter |
| TPM 1.2 | 32 | TPM_IncrementCounter |

the owner's authorization data (i.e., password), and the monotonic counters. The persistent state of an MTM contains similar data.

As a dictionary attack mitigation technique, the trusted module keeps track of the number of failed authorization attempts. This information should also be stored persistently. Finally, the TPM_SaveState command can be used to temporally save volatile state information (e.g., content of PCRs) in persistent state.

### 4.1.4 Monotonic Counters

The TPM 1.2 specification supports at least 4 concurrent 32 bit monotonic counters that can be created, incremented, read and destroyed. In [227] Sarmenta et al. examine how one physical TPM monotonic counter can be used to support virtual monotonic counters. They also propose new TPM commands to efficiently support an arbitrary number of virtual monotonic counters and so-called count-limited objects.

The generic TPM 1.2 counters have been defined as optional in the MTM specification. Instead three dedicated monotonic counters are specified (listed in Table 4.1). The bootstrap counter has a length of 5 bits, which means that it is initialized to 0 and runs up to 31. This counter is used to verify the validity of the firmware image that is initially executed during the bootstrap process, and it is crucial for the integrity of firmware upgrades. The small range of values makes it possible to implement the counter as 31 OTP bits in hardware (e.g., with fuses). The RIM protection counter enables a higher resolution of software upgrades (i.e., $2^{12} = 4096$ steps), whereas the storage protection counter is an additional monotonic counter for other purposes.

It should be noted that the idea to protect the boot process against downgrades with OTP fuses is used in other commercial products, e.g., the Microsoft Xbox

360 game console[1] and the Motorola Droid X phone [152].

## 4.2 Protection of Non-Volatile State in External Memory

In this section we present a framework to protect the persistent state when it is stored in external NVM. Any state protection scheme will rely on *authenticated encryption*, secure key storage and a *replay-detection* mechanism. These building blocks follow quite naturally from the security requirements that we will define in Section 4.2.1.

In the remainder of this chapter, we will describe in detail two state protection schemes. One scheme, which we presented in [228], relies on external authenticated non-volatile memory. The other makes use of a new security primitive called Reconfigurable Physical Unclonable Function (RPUF), which we introduced in [163].

We assume that an adversary has access to the communication interface between the chip in which the trusted module is embedded, and the external NVM chip that stores the persistent state. For now we do not address invasive hardware attacks on either the trusted module or the external NVM chip. The SPI bus is used as communication interface in many of the use cases described in Section 4.1.2; for instance SPI Flash memory is used in a modern PC to store the BIOS image and it is a popular choice to store FPGA bitstreams. The bandwidth of such an NVM interface is typically rather low. As we have demonstrated in Section 3.3 of the previous chapter, it is fairly straightforward to monitor a low-speed communication bus. Therefore we assume that the attacker can passively eavesdrop the communication and/or actively access the interface to read/write the content of the external memory.

### 4.2.1 Security Requirements

In order to sufficiently protect the persistent state of a trusted module, which we denote with $\mathcal{T}$, the following requirements have to be considered:

1. **State confidentiality:** It should be infeasible to read the content of $\mathcal{T}$. Disclosure of $\mathcal{T}$ will for instance reveal the private part of the SRK or of the EK.

[1] http://www.free60.org/Fusesets explains the function of the Xbox 360's fuse sets.

2. **State integrity:** Unauthorized modification of $\mathcal{T}$ should be infeasible. Otherwise an adversary can for instance change the owner's password.
3. **State uniqueness:** Cloning of the trusted module must be infeasible. Hence, copying of the EK to another module has to be prevented.
4. **State freshness:** Replay of an old state must be infeasible. This is mainly necessary to protect the monotonicity of counters.

The TCG specifications differ regarding the last requirement. The TPM has to store its general purpose monotonic counters in *physically shielded* locations, which implies tamper-resistant or tamper-evident hardware. The mobile specific monotonic counters should only be shielded from software executing outside the context of the MTM. This implies that for MTMs state freshness must not be guaranteed in the case of hardware attacks. The TCG Mobile Phone Work Group intends to tighten the security requirements of the MTM counters to the level of the TPM specifications "*immediately when it becomes feasible to implement such counters in mobile phones.*" This comment clearly acknowledges our observation that the application processor of mobile phones lacks the necessary embedded NVM.

Note that we do not list state availability as a security requirement. An attacker can always delete $\mathcal{T}$ in the external NVM or physically disable the communication interface between the trusted module and the NVM chip. In some sense this is equivalent to performing a Denial-of-Service (DoS) attack on the interface to a discrete TPM. Moreover, the external NVM typically also stores other platform components, such as the mobile phone's operating system or the FPGA bitstream. Hence the consequences of this type of attack reach further than the disabling of the trusted module.

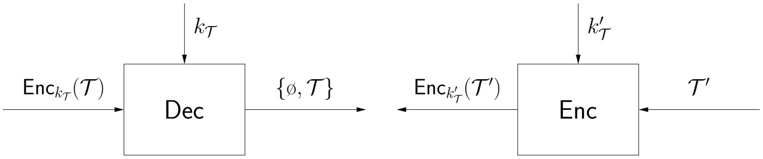
### 4.2.2 Generic Approaches

We identify two generic approaches to achieve the defined security requirements. Both approaches store the persistent state externally in an encrypted and authenticated form, but they differ slightly with respect to replay detection.

#### Non-Volatile State Protection with Updatable Key

The first approach, which is illustrated in Figure 4.1, relies on an updatable key.

With the algorithm $\mathsf{Enc}$ we mean an Authenticated Encryption (AE) scheme. The AE scheme simultaneously protects the confidentiality and authenticity


$k_{\mathcal{T}}$
$\mathsf{Enc}_{k_{\mathcal{T}}}(\mathcal{T})$
Dec
$\{\emptyset, \mathcal{T}\}$
$k'_{\mathcal{T}}$
$\mathsf{Enc}_{k'_{\mathcal{T}}}(\mathcal{T}')$
Enc
$\mathcal{T}'$

(a) Reading the non-volatile state $\mathcal{T}$ (b) Writing the non-volatile state $\mathcal{T}'$

Figure 4.1: Non-volatile state protection with updatable key.

of the persistent state (see Section 4.2.3). The corresponding Dec algorithm decrypts the non-volatile state and verifies its integrity. If the verification fails, Dec will return ø (indicating an integrity failure). Otherwise it will return the plaintext state $\mathcal{T}$ (see Figure 4.1(a)).

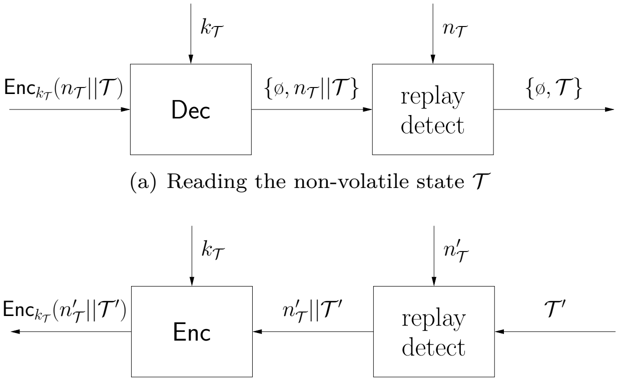
Whenever the non-volatile state is updated, it is authenticated and encrypted with a newly generated key $k'_{\mathcal{T}}$ (see Figure 4.1(b)). Older versions of the state will be rejected because they were protected by a different key. The generation of the new key can be done with a random number generation or it can be derived from the previous key, for instance, with a one-way function.

### Non-Volatile State Protection with Nonces

The second method, which is depicted in Figure 4.2, uses a fixed key $k_{\mathcal{T}}$ and a nonce $n_{\mathcal{T}}$. The nonce can either be a monotonic counter or a random number. In this approach the replay detection is done explicitly, whereas in the first approach it is implicit.

When reading its persistent state, the trusted module verifies whether the nonce $n_{\mathcal{T}}$ that is stored externally, corresponds with its own local copy (see Figure 4.2(a)). If the nonce does not match, the trusted module knows that it received an old version of $\mathcal{T}$; in this case the replay-detection algorithm returns ø. Note that the Dec algorithm can also return ø.

When writing an updated version of the state to the external NVM, a new nonce $n'_{\mathcal{T}}$ will be generated, respectively by incrementing the monotonic counter or by generating a new random number. Next, the updated state $\mathcal{T}'$ and the fresh nonce $n'_{\mathcal{T}}$ are authenticated and encrypted with the fixed key $k_{\mathcal{T}}$.


$k_{\mathcal{T}}$
$n_{\mathcal{T}}$
$\mathsf{Enc}_{k_{\mathcal{T}}}(n_{\mathcal{T}}||\mathcal{T})$
Dec
$\{\varnothing, n_{\mathcal{T}}||\mathcal{T}\}$
replay detect
$\{\varnothing, \mathcal{T}\}$
$k_{\mathcal{T}}$
$n'_{\mathcal{T}}$
$\mathsf{Enc}_{k_{\mathcal{T}}}(n'_{\mathcal{T}}||\mathcal{T}')$
Enc
$n'_{\mathcal{T}}||\mathcal{T}'$
replay detect
$\mathcal{T}'$

(a) Reading the non-volatile state $\mathcal{T}$

(b) Writing the non-volatile state $\mathcal{T}'$

Figure 4.2: Non-volatile state protection with nonce and fixed key.

### 4.2.3 Authenticated Encryption

Both schemes utilize an AE algorithm to protect the confidentiality and the integrity of the persistent state. Typically this type of algorithm can be constructed with a *generic composition* of an encryption algorithm and a keyed hash function (i.e., MAC algorithm) or by using a block cipher in a dedicated AE *mode of operation* [27]. In the former case, two independent keys must be used instead of a single key $k_{\mathcal{T}}$ like depicted in Figure 4.1 and 4.2.

In the current TPM specification the only mandatory encryption algorithms are RSA encryption with Optimal Asymmetric Encryption Padding (OAEP) and RSA encryption with PKCS #1 version 1.5 encoding. In theory the trusted module's persistent state could be encrypted with one of these RSA encryption schemes. In fact, this is how the TPM's key hierarchy is protected, as explained earlier in Section 2.1.2.

However, for efficiency reasons it is preferable to use a symmetric key algorithm instead. The TPM specification defines Advanced Encryption Standard (AES) in CTR mode and in OFB mode as optional encryption algorithms, e.g., for encrypted transport sessions (see Section 3.2.4). For instance, the Common Criteria evaluation report of the Infineon TPM indicate that it support AES in CTR mode (with key size of 128 bits) for transport sessions and that it also uses Triple DES (3DES) in Cipher Block Chaining (CBC) mode (with key size of 168 bits) internally to protect firmware upgrades. It is unclear whether other TPM manufacturers also support a symmetric encryption algorithm.

Note that if a symmetric key algorithm is available in the trusted module, it could also be used to protect the module's key hierarchy. This would mean that the SRK is made a symmetric key. In fact, Ekberg and Bugiel chose to make the SRK symmetric in order to minimize the memory footprint of their MRTM software implementation. They highlight the impact of this decision in [90].

By default TPMs already support a MAC algorithm, namely the HMAC construction [19] using the SHA-1 hash function. Alternatively, a MAC algorithm can be constructed from a block cipher (e.g., CBC-MAC [29], CMAC/OMAC [138], EMAC [213]) or based on a universal hash function (e.g., UMAC [28], GMAC [195]).

Various implementation options exist for an AE scheme $\mathsf{Enc}$ that is suited to protect the trusted module's persistent state. A first option is the combination of a MAC algorithm and a symmetric key algorithm. A logical choice for the two components are HMAC-SHA1 and AES respectively, because these algorithms are already used in the TCG specifications. Traditionally three composition methods are considered:

- **Encrypt-and-MAC:** Encrypt the plaintext and append a MAC of the plaintext.
- **MAC-then-encrypt:** Append a MAC to the plaintext and then encrypt them together.
- **Encrypt-then-MAC:** Encrypt the plaintext to get a ciphertext and append a MAC of this ciphertext.

Bellare and Namprempre showed that the third construction provides the strongest security guarantees [21]. The Encrypt-then-MAC method also has the advantage that the integrity of the state can be verified without the need to decrypt it first.

A second option is to use a block cipher, preferably AES, in a dedicated AE mode of operation. Examples of AE modes are CCM [141], CWC [156], EAX [22], GCM [195], IAPM [142], and OCB [221]. A distinction can be made between one-pass and two-pass schemes. The one-pass modes IAPM and OCB are highly efficient because they process the input message in a single pass, but sadly they are encumbered by patents. The two-pass modes on the hand are patent-free, but slower. However, the TPM was never intended to be a high-speed device, so it should not be an issue to use a slower AE mode.

In Figure 4.2 we indicated that the nonce $n_{\mathcal{T}}$ is prepended to the plaintext state $\mathcal{T}$ before it is processed by the AE algorithm. Block cipher modes of operations typically use an Initialization Vector (IV). It is logical to use the nonce $n_{\mathcal{T}}$ as

IV for the Enc algorithm. We denote the former method with $\mathsf{Enc}_{k_{\mathcal{T}}}(n_{\mathcal{T}}||\mathcal{T})$ and the latter with $\mathsf{Enc}_{k_{\mathcal{T}}}(n_{\mathcal{T}}, \mathcal{T})$.

### 4.2.4 Frequency of State Updates

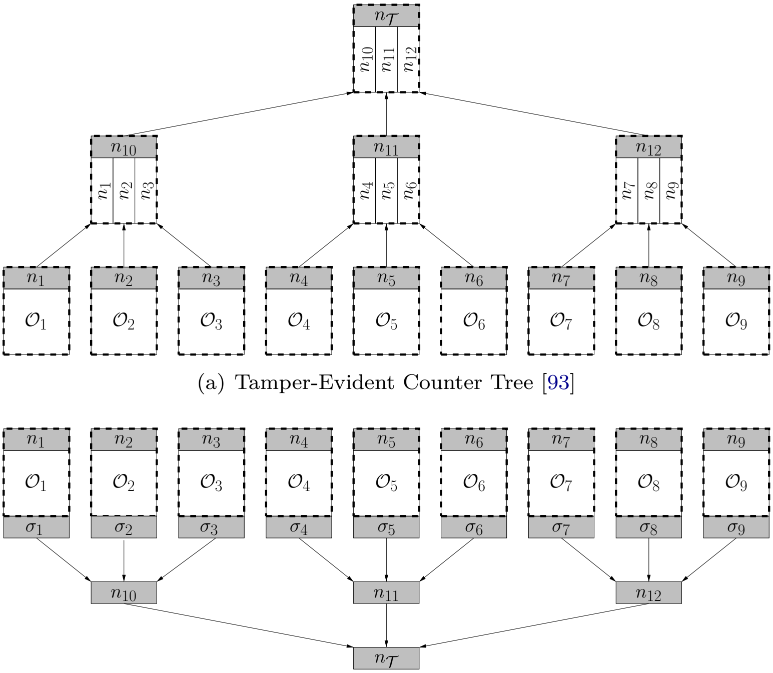
Irrespectively whether the state protection scheme relies on a fixed key and a nonce or an updatable key, or whether the Encrypt-then-MAC construction or a combined AE mode is used, the complete persistent state must be processed whenever it is accessed by the trusted module. When reading the state from the external NVM, it must be read entirely in order to verify its integrity. Every time the content of the non-volatile state is modified, the whole state must be re-encrypted, re-authenticated and overwritten in external NVM. This signifies that the trusted module needs enough internal RAM to cache the persistent state.

As explained in Section 4.1.3 the non-volatile state consists of some objects that are rarely changed and others (in particular the counters) that can be updated more frequently. Note that the TPM specification limits the rate at which the monotonic counters can be incremented (with TPM_IncrementCounter) to once every 5 seconds. Examples of infrequently updated objects are the EK and the SRK. The EK is installed during manufacturing and it will normally remain the same during the lifetime of a TPM; the TPM 1.2 specification does include the ability to reset the EK, but this feature is optional and often not supported in practice. The SRK on the hand is created when ownership is taken of the TPM and it will remain the same until the TPM is cleared and ownership is retaken. In the case of an MRTM the SRK and EK will never change because they are pre-installed by the remote owner (e.g., manufacturer or mobile operator).

The overhead of the state protection scheme can be reduced by splitting the state into two parts: one part containing the dynamic objects (i.e., counters) and the other comprising the infrequently updated objects (e.g., long term keys). As a result, whenever a monotonic counter is incremented, only the dynamic part of the persistent state has to be updated in the external NVM. Moreover, only one of the two parts has to be cached internally in the trusted module.

### 4.2.5 Authentication Tree

It is possible to go one step further by authenticating and encrypting logical objects (keys, counters, authorization data, ...) individually. The two generic approaches of Section 4.2.2 can be applied directly to these objects. However, this means that for every object the trusted module needs to remember either


$n_{\mathcal{T}}$
$n_{10}$ $n_{11}$ $n_{12}$
$n_1$ $n_2$ $n_3$
$n_4$ $n_5$ $n_6$
$n_7$ $n_8$ $n_9$
$\mathcal{O}_1$ $\mathcal{O}_2$ $\mathcal{O}_3$ $\mathcal{O}_4$ $\mathcal{O}_5$ $\mathcal{O}_6$ $\mathcal{O}_7$ $\mathcal{O}_8$ $\mathcal{O}_9$
$\sigma_1$ $\sigma_2$ $\sigma_3$ $\sigma_4$ $\sigma_5$ $\sigma_6$ $\sigma_7$ $\sigma_8$ $\sigma_9$

(a) Tamper-Evident Counter Tree [93]

(b) Hash tree [196]

Figure 4.3: Non-volatile state protection with an authentication tree. The non-volatile state $\mathcal{T}$ is split into different logical objects $\mathcal{O}_i$. Elements in a dashed box are authenticated and encrypted with the key $k_{\mathcal{T}}$. $\sigma_i$ denotes an authentication tag and $n_i$ a nonce. The trusted module only stores the root of the authentication tree $n_{\mathcal{T}}$ in on-chip memory and the rest of the tree is stored in untrusted external NVM.

an updatable key or a nonce. This issue can be overcome by protecting the object keys/nonces with a single updatable key/nonce and storing them in the external NVM.

In Figure 4.3 we illustrate two possible schemes for the protection of individual non-volatile objects. Every object $\mathcal{O}_i$ is authenticated and encrypted with the same key $k_{\mathcal{T}}$ that is stored inside the trusted module. A fresh nonce $n_i$ is associated with every object $\mathcal{O}_i$. We call these the *object nonces.*

In order to guarantee the freshness of a certain object, the corresponding object nonce must be regenerated every time the object is modified. As discussed earlier, this can be done by prepending the nonce to the object before encryption (i.e.,

$\mathsf{Enc}_{k_{\mathcal{T}}}(n_i||\mathcal{O}_i)$) or by using it as IV for the AE algorithm (i.e., $\mathsf{Enc}_{k_{\mathcal{T}}}(n_i, \mathcal{O}_i)$). The following two examples illustrate how the object nonces can be generated:

- If the AE algorithm works in CBC mode, the object nonce can be used as the IV. The IV should be random and unpredictable (i.e., $n_i = rnd_i$).
- If the AE algorithm works in CTR mode, the object nonce can be used as the initial counter value. It can be chosen partially random (i.e., $n_i = rnd_i||0$) or as $n_i = i||ctr_i||0$ with $ctr_i$ a monotonic counter that is incremented on every update of $\mathcal{O}_i$. The number of zeros depends on the length of the object.

In order to detect replay of the non-volatile objects we propose to protect the integrity of the object nonces with a so-called *authentication tree* [93, 123, 196]. The authentication tree reduces the problem of replay detection to the integrity protection of the root of the tree. The root of the authentication tree must be stored in the on-chip memory of the trusted module, while the rest of the tree can be stored in an untrusted external NVM.

Authentication trees have been used in a variety of schemes for the encryption and integrity protection of external RAM memories [92]. We will briefly discuss how two authentication tree schemes can be used for the protection of the trusted module's non-volatile state.

### Tamper-Evident Counter Tree

The solution that we describe in Figure 4.3(a), is an adaptation of the Tamper-Evident Counter Tree (TEC-Tree) scheme that was proposed by Elbaz et al. [93]. The original scheme use the principle of AREA (Added Redundancy Explicit Authentication) for integrity protection. They insert redundancy (a nonce) in the plaintext message before encryption and check it after decryption. This principle requires a block cipher mode of operation that provides infinite error propagation on encryption and on decryption. Elbaz et al. apply the AREA principle on a block level [94], which means that they add redundancy to every block and use Electronic Code Book (ECB) mode. We believe that it makes more sense to use a proper AE scheme for Enc instead of the AREA principle.

Our adapted scheme works as follows. The objects $\mathcal{O}_i$ and their corresponding object nonces $n_i$ are encrypted and authenticated with key $k_{\mathcal{T}}$ (i.e., $\mathsf{Enc}_{k_{\mathcal{T}}}(n_i||\mathcal{O}_i)$) and subsequently stored externally as the leaves of the tree. The intermediate nodes of the tree group a number of the object nonces. These nodes are also encrypted and authenticated by the key $k_{\mathcal{T}}$ and their freshness is protected by another nonce, which we call an *intermediate nonce*. Finally, the root node of the tree groups the intermediate nonces. The nonce $n_{\mathcal{T}}$ protects the freshness of the root of the tree and hence it has to be stored internally in the trusted module in order to detect replay of any element in the authentication tree. We call $n_{\mathcal{T}}$ the *root nonce*.

Elbaz et al. proposed to use counters for the nonces and hence they named their scheme Tamper-Evident Counter Tree or TEC-Tree. The nonces are constructed in the following manner: $n_i = a_i || ctr_i$ with $a_i$ the address of the element and $ctr_i$ a counter.

In Figure 4.3(a) we illustrate an example where the persistent state $\mathcal{T}$ consists of nine objects and where a *ternary* TEC-Tree is used. In a ternary tree there are (at most) three child nodes under every node. The intermediate nodes are calculated by grouping three objects nonces: $\mathsf{Enc}_{k_{\mathcal{T}}}(n_i||\mathcal{O}_i)$ with $\mathcal{O}_{10} = n_1||n_2||n_3$ for the left node, $\mathcal{O}_{11} = n_4||n_5||n_6$ for the middle node and $\mathcal{O}_{12} = n_7||n_8||n_9$ for the right node. The root node is computed as $\mathsf{Enc}_{k_{\mathcal{T}}}(n_{\mathcal{T}}||\mathcal{O}_{\mathcal{T}})$ with $\mathcal{O}_{\mathcal{T}} = n_{10}||n_{11}||n_{12}$.

When the trusted module wants to read a certain object $\mathcal{O}_i$, the following steps must be performed:

1. The trusted module reads the encrypted object (e.g., $\mathsf{Enc}_{k_{\mathcal{T}}}(n_i||\mathcal{O}_i)$) from the external NVM, decrypts it and verifies its integrity (using the $\mathsf{Dec}$ algorithm of the AE scheme).

2. The trusted module reads the intermediate node (e.g., $\mathsf{Enc}_{k_{\mathcal{T}}}(n_j||\mathcal{O}_j)$) from the external NVM, decrypts it and verifies its integrity. Next it checks whether the object nonce $n_i$ that was decrypted in the previous step, matches the nonce that is stored in the plaintext intermediate node $\mathcal{O}_j$.

3. The trusted module reads the root node (i.e., $\mathsf{Enc}_{k_{\mathcal{T}}}(n_{\mathcal{T}}||\mathcal{O}_{\mathcal{T}})$) from the external NVM, decrypts it and verifies its integrity. Next it verifies whether the intermediate node $n_j$ that was decrypted in the previous step, corresponds to the nonce that is stored in the plaintext root node $\mathcal{O}_{\mathcal{T}}$.

4. The trusted module checks whether the root nonce $n_{\mathcal{T}}$ that was decrypted in the previous step, is the same as the value that the module stores internally.

Every time an object $\mathcal{O}_i$ is changed, the trusted module must update the path in the authentication tree from the corresponding leaf node to the root node. For every element that is modified in the tree, a new nonce must be generated by incrementing the counter value. For instance, if $\mathcal{O}_4$ changes in Figure 4.3(a), the nonces $n_4$, $n_{11}$ and $n_{\mathcal{T}}$ will be altered.

An important feature of the TEC-Tree scheme is that the authentication tree is *parallelizable* [123] (for read and write operations). This means that the cryptographic operations involved when reading and updating the tree can be performed in parallel.

### Hash Tree

In Figure 4.3(b) we illustrate an alternative solution, which builds upon the memory protection schemes of the MIT research group of Devadas [62, 107, 266].

The non-volatile objects $\mathcal{O}_i$ are authenticated and encrypted with a fixed key $k_{\mathcal{T}}$, like in the previous scheme. The AE algorithm typically generates message authentication tags on the objects; e.g., in the case of the Encrypt-then-MAC construction the tag is the MAC on the ciphertext. This can be denoted as follows: $(c_i, \sigma_i) \leftarrow \mathsf{Enc}_{k_{\mathcal{T}}}(n_i, \mathcal{O}_i)$ with $n_i$ the object nonce (used as IV), $c_i$ the object ciphertext and $\sigma_i$ the *object tag*.

Note that in Figure 4.3(b) we depict the object tags explicitly, while in Figure 4.3(a) they are also present but not shown.

A hash tree, also known as Merkle tree [196], is used to protect the freshness of the persistent state. The object tags $\sigma_i$ form the leaf nodes of the tree and they are hashed together yielding the *intermediate hashes*. This process is repeated until the root of the tree is reached. Figure 4.3(b) gives the example of nine objects and a ternary hash tree (with height 2). The intermediate nodes hash together three object tags: $n_{10} = \mathcal{H}(\sigma_1||\sigma_2||\sigma_3)$ for the left node, $n_{11} = \mathcal{H}(\sigma_4||\sigma_5||\sigma_6)$ for the middle node and $n_{12} = \mathcal{H}(\sigma_7||\sigma_8||\sigma_9)$ for the right node. The root of the tree $n_{\mathcal{T}}$, which is called the *root hash*, is calculated by hashing the intermediate hashes; i.e., $n_{\mathcal{T}} = \mathcal{H}(n_{10}||n_{11}||n_{12})$.

A disadvantage of this scheme compared to the previous is that a hash tree is non-parallelizable. When the persistent state is modified, the path from the modified object to the root of the tree must be updated *sequentially*. For instance, if in Figure 4.3(b) $\mathcal{O}_7$ is modified, first it must be authenticated and encrypted (with a fresh nonce $n_7$) yielding a new value for $\sigma_7$, subsequently the intermediate hash $n_{12}$ can be recalculated and finally the root of the tree $n_{\mathcal{T}}$ can be recomputed.

### Incremental Hashing

Usually a collision-resistant hash function is used to construct a hash tree. However, in [134] Hu et al. propose a fast memory authentication protocol that is based on a Merkle tree built using the universal hash family NH [28] combined with the AES algorithm. The main advantage of their scheme is that the NH family of universal hash functions has the ability to incrementally update hashes.

The concept of *incremental* hashing was introduced in the work of Bellare et al. [20]. The idea behind this concept is that once having once the hash of a document $M$ has been calculated, the time to update the hash upon modification of $M$ should be proportional to the amount of modification done on $M$.

## 4.2.6 On-Chip Non-Volatile Memory

The goal of this chapter has been to minimize the on-chip non-volatile memory that is needed inside the trusted module. The state protection scheme protects the confidentiality, the integrity and the freshness of the module's persistent state when it is stored in untrusted external NVM. All the approaches that we have described, still require a small amount of embedded NVM.

In Section 4.2.2 we have identified two generic approaches for non-volatile state protection. The first relies on an authentication/encryption key $k_{\mathcal{T}}$ that is modified on every state update. The on-chip non-volatile memory that is needed to store the key $k_{\mathcal{T}}$ has to be reprogrammable because the key is changed repeatedly. Whenever the state is altered, the content of the external NVM will be re-encrypted and re-authenticated with a newly generated $k_{\mathcal{T}}$. In order to recover from failures during this update process, it is advisable to temporarily store the old and new version of $k_{\mathcal{T}}$ in the on-chip NVM. So in practice the size of the on-chip storage is at least twice the key size.

The second generic approach makes use of a fixed key $k_{\mathcal{T}}$ and an additional value that protects against state replay. This additional value is changed every time the non-volatile state is altered. Examples of the replay-detection value are the root of an authentication tree or a counter or random number that is used as the IV to authenticate and encrypt the persistent state.

On the one hand, the state protection key $k_{\mathcal{T}}$ can be programmable in OTP NVM during manufacturing of the trusted module. It is important that the key $k_{\mathcal{T}}$ is unique for every trusted module. Otherwise the content of the external NVM can be copied from one module to another. This attack will be successful under the condition that the replay-detection value is the same for both modules;

if a counter is used for $n_{\mathcal{T}}$ there is a reasonable chance that this condition is satisfied.

ON the other hand, the replay-detection value has to be stored in on-chip MTP memory because it changes on every state update. If the value is a counter, it is not strictly necessary to maintain a backup copy for reliability reasons; if a failure occurs during the state update process the previous value can be deducted from the counter value itself. However, if the value is random (either a randomly generated nonce or a calculated hash value), it is prudent to store the value twice. The amount of embedded MTP memory that is needed, depends on the number of state updates that have to be supported. The TCG requires that a TPM can store at least one 32-bit monotonic counter and an MTM at least one 5-bit counter (see Table 4.1).

### One-Time-Programmable Non-Volatile Memory

The key difference between strictly ROM and OTP NVM, also referred to as Programmable Read-Only Memory (PROM), is that the content of ROM is determined before production of the device and hence fixed for all manufactured devices, whereas OTP NVM is programmed after the device is produced. As described earlier, the externalized state must be uniquely bound to a specific trusted module and consequently the state protection key $k_{\mathcal{T}}$ must be different for every module. This implies that the key must to be stored with OTP memory instead of ROM.

A common technique to realize OTP NVM are fuses and antifuses. Whereas a fuse starts with a low resistance and is designed to permanently break an electrically conductive path (e.g., when the current through the path exceeds a specified limit), an antifuse starts with a high resistance and is designed to permanently create an electrically conductive path. The fact whether a fuse or antifuse is blown or not, can be used to store logical bits.

Some fuse technologies can only be programmed during the manufacturing of an Integrated Circuit (IC); e.g., laser fuses must be blown up with a laser beam. More recent electrically programmable fuse (eFuse) technologies can be programmed during manufacturing as well as in the field. The high voltage that is needed to blow the (anti)fuse can be provided through an external pin or generated internally by a built-in charge pump.

In practice the module-specific, non-updatable key $k_{\mathcal{T}}$ will be generated externally and programmed into the trusted module by the manufacturer (e.g., through an external pin or with a laser) or alternatively it will be generated internally and burned in the non-volatile memory by the trusted module itself.

In both scenarios field programmability is not a strict requirement for the NVM technology that is used. In the latter scenario the manufacturer can provide the voltage externally that is needed to burn the (anti)fuses, even though the key is generated internally.

As explained in Section 4.1.4 a limited-size monotonic counter can be implemented with multiple OTP NVM cells: the number of OTP fuses that have been blown determines the value of the counter. Traditional eFuse technologies, which use silicided polysilicon lines, are not suited for high density NVM because the fuses take considerable chip area. Novel logic OTP NVM solutions promise smaller NVM cells, sometimes even the equivalent of a single CMOS transistor. However, OTP memory will never be large enough to efficiently support the general-purpose 32-bit TPM monotonic counters.

### Multiple-Time-Programmable Non-Volatile Memory

The motivation of this chapter is that there are a number of computing platforms where embedded reprogrammable NVM is unavailable or scarce, predominantly for cost reasons. The cases that we identified are the integration of a TPM in the chipset of a PC, the realization of an MTM on the application processor of a mobile phone, and the implementation of a trusted module on a volatile FPGA.

The only MTP NVM technology that is used in practice in the identified cases, is battery-backed RAM. The Intel iTPM solution uses OTP fuses for a 128-bit *chipset key*, that takes the role of $k_{\mathcal{T}}$ in our terminology, and it relies on the motherboard battery, which traditionally powers the RTC, to retain a monotonic anti-replay counter in volatile RAM across power cycles. Altera and Xilinx FPGAs can store a cryptographic key to protect their configuration bitstream, either in OTP fuses or in battery-backed SRAM. However, battery-backed RAM is not a viable solution for all platforms because of its drawbacks: the extra battery signifies an additional cost and the maintenance issues arise when the battery is drained. Modern mobile phones already struggle with their battery lifetime and they often have a user replaceable battery, which implies that the backup power is not necessarily guaranteed.

A number of small Intellectual Property (IP) companies, including eMemory, Kilopass, Novocell Semiconductor, NSCore and Sidense, offer unconventional embedded NVM technologies [69]. These NVM solutions are not based on floating gate transistors like traditional Electrically Erasable Programmable Read-Only Memory (EEPROM) and Flash memory and they can be manufactured in standard CMOS processes without additional mask layers or process steps. However, these solutions are often immature: they are not available in the latest CMOS process technology or they support only a limited number of reprogrammability cycles.[2]

In [228] we determined that an alternative approach, until low-cost embedded MTP NVM technology becomes a commodity, is to *extend the security perimeter* of the trusted module to the external NVM chip. By adding a MAC primitive and some additional logic to the external NVM chip and by programming a symmetric key in both the trusted module and the external NVM, the integrity and freshness of NVM's read and write operations can be protected cryptographically. In Section 4.4.1 this proposal will be described in detail.

Ekberg and Asokan build upon our work in [89]. They mainly focused on lifecycle management issues, such as field replacement of NVM units and testability of newly fabricated as well as field-returned units. Micron (formerly Numonyx, and originally Intel and STMicroelectronics) sells security-enhanced NOR Flash memory that supports authenticated and replay-protected operations. The original Intel concept consisted of a standard Flash memory with an integrated RSA and SHA-1 engine and a hardware RNG [4].

## 4.3 Physical Unclonable Function-Based Key Storage

In this section we discuss how Physical Unclonable Functions (PUFs) can be used to solve the problem of non-volatile state protection. Previous work has shown that unique identifiers and device specific cryptographic keys can be derived from the responses of a PUF [264, 268, 287]. It is clear that the key $k_{\mathcal{T}}$ that is needed to protect the trusted module's persistent state, can be derived from a PUF response. In Chapter 6 we will show that this approach is ideally suited when implementing a trusted module on reconfigurable hardware.

Regular PUFs have a static challenge-response behavior, i.e., for a given challenge the PUF will always return (a noisy version of) the same response. This implies that the PUF-derived key $k_{\mathcal{T}}$ is fixed and thus that the state protection scheme still needs to rely on additional source of freshness for the prevention of state replay (see Section 4.2.2).

Ideally, however, a dynamic PUF would be desirable in order to allow the key $k_{\mathcal{T}}$ to be updated. This would remove the need to include additional MTP NVM in the trusted module. In [163] we investigated how to modify the

[2] Some logic NVM technologies are in fact antifuse-based and mimic the MTP behavior with multiple OTP NVM cells. Sidense calls this emulated MTP mode and eMemory uses the term pseudo MTP.

challenge-response behavior of a PUF, which would enable the realization of an updatable PUF-derived key. This new security primitive, which we call a reconfigurable PUF, can be used to protect the trusted module's persistent state without on-chip MTP NVM.

### 4.3.1 Physical Unclonable Functions

PUFs are a relatively new[3] security primitive introduced in [206, 207] and extensively studied since the beginning of this century. They provide a low-cost manner to make devices tamper evident and may protect them against counterfeiting. Their unclonability is not based on fundamental physical laws (as in the case of quantum cryptography and qubits), but it is based on physical phenomena that are known to be very hard, time consuming and expensive to replicate. Typically, a challenge-response mechanism provides an interface to the PUF and defines the functional behavior of the PUF.

The way to work with PUFs is very similar to the way of working with human biometrics. Just as in the biometrics situation, an enrollment phase is carried out first. During this phase the PUF is challenged for the first time and its response(s) is (are) processed and characterized. Later, during the verification or reconstruction phase, a response is measured according to the same challenge(s) as applied during enrollment. The fresh response(s) will be close (according to some distance measure) to the corresponding enrollment response(s) on the same PUF but very different from the response(s) on a different PUF. Hence, the challenge-response behavior of PUFs can be used to identify or authenticate the PUF itself and thus, the object in which the PUF is embedded or inherently present.

Today many PUF implementations are known [105, 117, 162, 176, 206, 287] and thanks to their low-cost tamper evident or tamper resistant implementations, PUFs are proposed as security primitives in a multitude of applications, which include: Radio Frequency Identification (RFID) authentication when used in combination with the HB and HB+ protocols [124, 125], secret key storage and tamper evident containers [265, 268, 287], building blocks of block and stream ciphers [10], and in IP applications [118, 119, 243].

In our research we focus on the usage of a PUF for key storage and we are primarily interested in PUFs that are *intrinsically* present in ICs, i.e., without requiring any change to the hardware or the production process. The main examples of intrinsic PUFs are based on (volatile) memory structures such as

[3] In the early 90s, well before the introduction of the PUF concept, Simmons [242] and Tolk [276] proposed an unclonable identification system based on random optical reflection patterns.

SRAM [117, 132, 133], flip-flops [184, 290] and latches [162, 263] or on delay variations in combinatorial circuits [105, 106, 176, 267, 269].

In [264, 287] a new paradigm for key storage was introduced and discussed: *do not store a key in digital form in a device, but resurrect it from the available hardware (read PUF) every time you need it.* This new paradigm offers an attractive alternative for key storage with traditional non-volatile memory, for two reasons:

- When the device is not powered, the long-term key is not present and therefore it is less vulnerable to (semi-)invasive attacks. For instance, it has been shown that invasive FIB attacks on a coating PUF will significantly modify the key that is derived from the protective coating [287]; in some sense, this behavior can be considered a form of automatic *key zeroization.* However, it remains unclear to what extent intrinsic PUFs protect against active invasive attacks (e.g., probing) [198]. It is sometimes argued that the challenge-response behavior of a intrinsic PUF will change when the chip is decapsulated and its passivation layer is removed, but, as far as we can tell, this behavior has not (yet) been experimentally verified.
- When this approach is implemented with an intrinsic PUF, the cost penalty is very limited since intrinsic PUF are only based on base-line semiconductor components (e.g., SRAM cells) present in all current and in all near future foreseeable technology nodes.

For an in-depth overview of various applications of PUFs and of the most important PUF constructions that have been proposed in the literature we refer to Maes [182].

### 4.3.2 Reliable Key Extraction with Fuzzy Extractors

The generation of a secret key from PUF responses needs some additional processing steps. The process that turns a PUF response into a cryptographic keys, consists of two steps: *error correction* and *randomness extraction.* We briefly repeat the concept of a *fuzzy extractor* or *helper data algorithm* that allows to extract a cryptographically secure key from noisy and non-uniformly random PUF responses. For details we refer to the literature [76, 180, 288, 294].

### Enrollment

During *enrollment* of the PUF, the procedure $\mathsf{Gen}$ is carried out:

$$(k, w) \leftarrow \mathsf{Gen}(r)\,.$$

It takes as input a noisy PUF response $r$ and creates a key $k$ and helper data $w$. Clearly the key $k$ has to be kept secret. The helper data $w$ on the other hand is public and can be stored in non-secure NVM. In order to protect against active attackers (changing the helper data) robust fuzzy extractors should be used [36].

### Reconstruction

During the *key-reconstruction phase*, a noisy PUF response $r'$ is measured. Then the procedure $\mathsf{Rep}$ is run:

$$k \leftarrow \mathsf{Rep}(r', w)\,.$$

It produces the same key $k$ on input $r'$ and $w$ given that $r'$ is sufficiently close to $r$.

### Practical Aspects

Proof-of-concept implementations of fuzzy extractors have been presented in the academic literature [35, 185, 314]. Intrinsic-ID, a spin-out of Philips Research, offers commercial IP for intrinsic PUFs and accompanying fuzzy extractors.

In 2011 Merli et al. demonstrated that side-channel attacks can be performed on fuzzy extractors [198]. In particular they targeted the Toeplitz universal hash function that is often used for randomness extraction in the key reconstruction phase. Countermeasures against this type of attack have yet to be proposed. Note however that side-channel attacks can also be mounted on the encryption algorithm $\mathsf{Enc}$ while it processes the PUF-derived key $k_{\mathcal{T}}$, so after the fuzzy extractor has executed.

## 4.3.3 Reconfigurable PUFs

The traditional PUF constructions have always considered a static challenge-response behavior: for a given challenge, the PUF should always yield the same or a similar response.[4] In order to achieve updatable PUF-based key storage, a mechanism is needed to alter the challenge-response behavior. Lim was the first to observe that this is a desirable feature and denoted this as a Reconfigurable Physical Unclonable Function (RPUF) [176]. He also sketched an arbiter PUF with floating gate transistors. However, it is unclear how the proposed PUF construction would be reconfigured and whether a reconfiguration sufficiently changes the PUF's responses. In [163] we defined the RPUF primitive rigourously, presented the first physical implementations of a reconfigurable PUF and introduced a scheme that uses an RPUF to protect the non-volatile state of a trusted module.

Majzoobi et al. also use the term reconfigurable PUF [187] for the implementation of a static PUF in reconfigurable hardware. However, this approach does not yield a secure RPUF because the reconfigurations of contemporary FPGAs can be undone.

Loosely speaking an RPUF is a PUF with a mechanism to change the PUF into a new one, ideally with a new unpredictable and uncontrollable challenge-response behavior even if one would know the challenge-response behavior of the original PUF. Additionally the new PUF inherits all the security properties of the original one. Furthermore, the reconfiguration mechanism has to be uncontrollable and should not be based on updating a hidden parameter; e.g., part of the challenge [176] or the location of the PUF structure in reconfigurable FPGA logic [187]. The PUF reconfiguration must be difficult to revert, even with invasive means.

In the following, we present two possible implementations of reconfigurable PUFs. The disadvantage with the proposed implementations is that the PUF is not inherently present in ICs and the manufacturing process to make such a device (IC with embedded reconfigurable PUF) is significant. It is an active research area to design more practical reconfigurable PUFs that can be implemented intrinsically (i.e., in CMOS technology). So for now the practicality of this solution (as a replacement for embedded MTP NVM) is limited.

Before describing the physical implementation of the RPUF, we introduce certain parameters which will make the treatment of the RPUF more concise and specific.

[4] In practice, due to measurement noise, the responses should be as close to each other as possible.

### Formal Definition of Reconfigurable PUF

Let $\mathcal{S} = \{0,1\}^n$ denote the configuration space of the physical system that constitutes the PUF, $\mathcal{C}$ be the space of challenges that can be applied to the system and $\mathcal{R}$ be the response space of the system. The responses in $\mathcal{R}$ are observed through a noisy channel. Therefore we model the mapping that maps challenges to responses as random variables as follows,

$$\mathcal{P} = \{R : \mathcal{S} \times \mathcal{C} \to \mathcal{R} | R(s,c) \text{ is a random variable distributed according} \\ \text{to } \mathbb{P}_{s,c}\} ,$$

where $\mathbb{P}_{s,c}$ denotes a probability distribution on $\mathcal{R}$. The responses $R(.,.)$ of the PUF depend on the configuration $s \in \mathcal{S}$ of the physical system, which is secret, and on the challenge $c$ (which is public) applied to it.

Note that for a fixed challenge $c$ there are $2^n$ possible noisy responses $R(.,c)$ since the number of physical systems is $2^n$.

The state space $\mathcal{S}$ defines the configurations of the random components in the PUF that determine its functional behavior. As an example we mention that it defines the positions of the scattering particles in the case of optical PUFs [206, 289].

More precisely we define PUFs as follows:

**Definition 1 (Type of PUF)** *We define a PUF type as a set of physical systems represented by the tuple $(\mathcal{S}, \mathcal{P}, \mathcal{C}, \mathcal{R})$ with state space $\mathcal{S}$, function space $\mathcal{P}$, challenge space $\mathcal{C}$, and response space $\mathcal{R}$.*

**Definition 2 (PUF Instantiation)** *An instantiation of the PUF type $(\mathcal{S}, \mathcal{P}, \mathcal{C}, \mathcal{R})$ is defined by $(s, R(s,.))$ with secret state $s \in_R \mathcal{S}$, where $R(s,.)$ is the restriction of the random variable $R(.,.)$ to the space $\mathcal{C}$.*

Next we consider the security of PUF instantiations.

**Definition 3 (PUF Security)** *We call a PUF instantiation* information theoretically secure[5] *iff the following conditions hold:*

$$\mathbf{I}(R(C'); R(C), C, C') \approx 0 ,$$

*where $C, C', R(C), R(C')$ are random variables over $\mathcal{C}$ and $\mathcal{R}$, and $\mathbf{I}(.;.)$ denotes the mutual information. We call it* physically secure *if*

[5] Analog definitions can be given to define cryptographically secure PUFs.

1. *It is very hard to clone the PUF, i.e., the time complexity of making a physical or mathematical (simulation) copy is exponential in $n$.*

2. *The PUF is tamper evident.*

We remind the reader that unclonability of a PUF means that the PUF is also unclonable for the manufacturer of the PUF (*manufacturer non-reproducibility*). The production of the PUF is **not** based on a secret that the manufacturer has to hide. Tamper evidence stands for the fact that when the PUF is damaged (e.g., by an invasive attack) the PUF is damaged up to such an extent that its challenge-response behavior is completely changed in an unpredictable way.

For a given challenge $c \in \mathcal{C}$ we denote by $r(s, c)$ the actual response of the PUF (often we will write $r(c)$ when it is clear which PUF is being considered).

**Definition 4 (PUF Interfaces)** *An instantiation of a PUF $(s, R(s, .))$ has the following interface function*

$$r(c) \leftarrow \mathsf{Read}(c) \quad \textit{with} \quad r(c) \in \mathcal{R}\,.$$

*Moreover, an instantiation of a reconfigurable PUF has additionally the following interface function*

$$s' \leftarrow \mathsf{Reconf} \quad \textit{with} \quad s' \in_R \mathcal{S}\,.$$

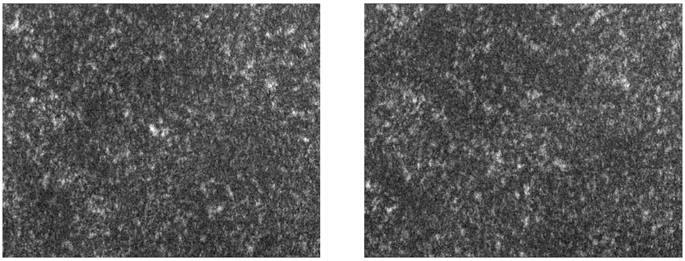
Note that $s' \in_R \mathcal{S}$ implies that a completely new uncontrollable physical system (and hence challenge-response behavior) is generated.

For $|\mathcal{C}| = 1$ (i.e., a PUF with only one challenge) the PUF is called a Physically Obfuscated Key (POK) [103]. In the remainder of this section we mainly talk about information theoretically and physically secure instantiations of a PUF/POK, but just call them PUFs.

### Reconfigurable Optical PUF

As a first example of a reconfigurable PUF we present the *reconfigurable optical PUF*. The physical structure is an object containing light scattering particles. The position and physical state (polarization) of these particles define the configuration of the structure [289]. The structure satisfies the following two conditions:

1. When the structure is irradiated with a laser beam within *normal* operating conditions, the structure does not change its internal configuration and produces a "steady" speckle pattern (see Figure 4.4(a)).



(a) Before Reconf. (b) After Reconf.

Figure 4.4: Example of speckle pattern.

2. When the structure is irradiated with a laser beam outside the *normal* operating conditions, the structure will change its internal configuration (see Figure 4.4(b)).

Concretely we propose a structure that consists of a polymer containing randomly distributed light scattering particles[6] as opposed to the normal glass (optical) PUF by Pappu [206].

In terms of the parameters that we previously described, the reconfigurable optical PUF is described as follows:

- The configuration space is given by $\mathcal{S} = \{0,1\}^n$ where $n = V/\lambda^3$ and $V$ the volume of the structure and $\lambda$ the wavelength of the laser. A cube of volume $\lambda^3$ is usually called a *voxel*. A configuration string $s \in \mathcal{S}$ defines the voxels in which a scatterer is present and defines completely the speckle patterns that are generated when the structure is irradiated with a laser beam.
- The challenge space $\mathcal{C}$ is defined by the set of angles and locations under which the structure should be irradiated by the laser beam.
- The response space $\mathcal{R}$ is the set of all possible speckle patterns (see Figure 4.4 for two examples).
- It was shown in [289] that the angles and locations have to be sufficiently far apart to satisfy the information theoretical security condition, which states that the mutual information between two responses corresponding to two different challenges is (approximately) equal to zero.

[6] Alternatively a phase change substance, widely used in rewritable optical discs, can be used instead of a polymer.

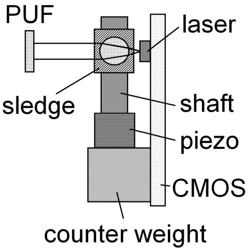


Figure 4.5: Schematic side view of integrated optical PUF [288].

- The fact that an optical PUF is hard to clone in a physical way follows from the fact that speckle phenomena are very sensitive to very small variations in the locations of the scattering particles [206]. Accurate mathematical modeling of speckle phenomena is very difficult [111].
- The interface Read($c$) applied to a challenge $c$ is implemented by irradiating the PUF with the laser according to the angle and position defined by $c$ and measuring the speckle pattern with the CMOS sensor (see Figure 4.5).
- The Reconf command is implemented by driving the laser at a higher current such that a laser beam of higher intensity is created which heats up the polymer locally and starts to melt. After a short time the laser beam is removed and the structure cools down such that the particles freeze.

Finally we note that Škorić et al. [294] have shown that optical PUFs (containing a laser and a structure) can be completely integrated and therefore be produced very compactly. The results obtained there clearly transfer to the reconfigurable optical PUF too. Hence we believe that a practical implementation is feasible.

Further investigation is needed to determine the quality of the optical RPUF construction. Will other external factors such as environmental temperature also trigger the reconfiguration behavior? How independent are challenge-response pairs before and after reconfiguration, especially after repeated reconfigurations? How many reconfigurations can be supported before the polymer material deteriorates?

### Phase-Change Memory-based RPUF

Phase Change Memory (PCM) is a new type of fast non-volatile memory that has the potential to replace Flash memory and even DRAM [220, 306]. Each

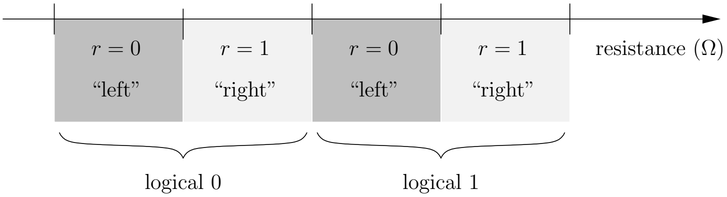
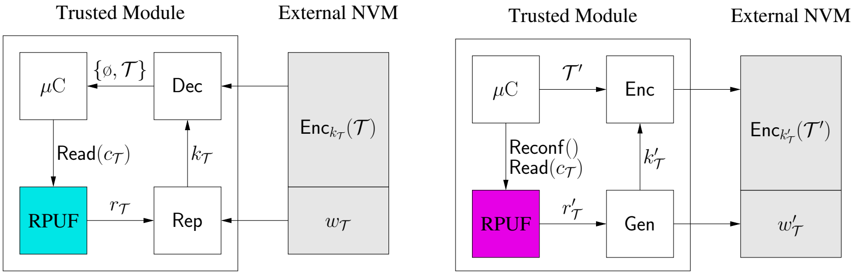


Figure 4.6: PCM-based RPUF. Limited control over the heating allows for 2 logical states and an accurate measurement gives one RPUF bit $r$.

memory cell contains a piece of chalcogenide glass, usually a doped alloy of germanium, antimony and tellurium (GeSbTe), the same material used in rewritable optical discs. By subjecting it to a specific heating pattern, a phase change is induced: in the *amorphous* state the resistivity is high (logical "1" state) and in the *crystalline* state it is low ("0"). Intermediate states (e.g., semi-amorphous and semi-crystalline) can be realized as well, allowing for more than one bit of storage per cell. This is known as Multi-Level Cell (MLC) PCM [205]. The heating is regulated by passing a current through the cell. The state is read out by measuring its resistance. PCM has very favorable properties. Phase transition times of 5 ns have been achieved. Furthermore, a PCM cell may endure around $10^8$ write cycles.

We observe that *the control over the phase can be made less precise than the accuracy of the resistance measurement.* Consider a cell whose state can be controlled just well enough to reliably realize $n$ logical states (resistance axis divided into $n$ intervals), while measurements are precise enough not only to tell in which interval the resistance lies, but also *where* in that interval. Each logical interval is subdivided into a number of more fine-grained intervals, e.g., "left" and "right" (Figure 4.6). This additional information is easy to read, but cannot be controlled during the writing process. Hence we have an uncontrolled process resulting in a long-lived random state that can be reconfigured at will; precisely what is needed for an RPUF. PCM's suitability for embedded memory allows for a compact RPUF located inside an integrated circuit.

At the moment the PCM-based RPUF remains a concept. In 2010 Micron (formerly Numonyx) and Samsung released Single-Level Cell (SLC) PCM products as an alternative for NOR Flash memory. However, these commercial products can not be used to practically verify the proposed RPUF construction, because internal access is needed to measure the resistance of the memory cells. Furthermore, the literature suggests that MLC PCM suffers from *resistance drift*, i.e., the resistance of a PCM cell increases over time [59]. This phenomenon will have a negative effect on the RPUF construction that we propose.



(a) Reading the non-volatile state $\mathcal{T}$.

(b) Writing the non-volatile state $\mathcal{T}'$.

Figure 4.7: Non-volatile state protection with RPUF-derived key.

Also note that it is uncertain whether a PCM-based RPUF will protect against physical attacks. It might be possible to distinguish the state of a PCM cell visually with an electron microscope. Alternatively an adversary might try to measure the resistance of memory cells with an invasive probing attack.

### 4.3.4 Non-Volatile State Protection with RPUFs

In Figure 4.7 we illustrate how the non-volatile state of a trusted module can be externalized and protected with an RPUF-derived key. The scheme is a straightforward implementation of the generic state protection scheme that relies on an updatable key (see Section 4.2.2). Each time the state $\mathcal{T}$ is modified, the RPUF is reconfigured, new helper data $w'_{\mathcal{T}}$ is generated and the persistent state is re-encrypted and re-authenticated with the resulting new key $k'_{\mathcal{T}}$.

We consider that PUF reconfigurations are uncontrollable and consequently difficult to revert. This enables the trusted module to detect replay attacks. An adversary might be able to read the encrypted state from the external NVM and can try to overwrite it at a later moment in time with this old copy. However, he will not be successful as the RPUF-derived state protection key will have been altered.

#### Efficiency Improvements

When we introduced authentication trees as a technique to efficiently protect the integrity and freshness of the non-volatile state (see Section 4.2.5), we assumed that the state would be authenticated and encrypted with a fixed key $k_{\mathcal{T}}$. We also assumed that the root of the authentication tree $n_{\mathcal{T}}$, which is a counter in the case of a TEC-Tree and a hash value for a Merkle tree, would be persistently stored inside the trusted module. In [163] we argued that the root can also be stored in external NVM, if it is authenticated with an updatable RPUF-derived key.

### Reliable State Updates

In order to recover from accidental or malicious failures during an update of the state $\mathcal{T}$ we propose to authenticate the root of the authentication tree[7] with two keys $k_{\text{auth}_1}$ and $k_{\text{auth}_2}$ derived from different RPUFs: $\sigma_{\mathcal{T}_1} = \mathcal{H}_{k_{\text{auth}_1}}(n_{\mathcal{T}})$ and $\sigma_{\mathcal{T}_2} = \mathcal{H}_{k_{\text{auth}_2}}(n_{\mathcal{T}})$. This requirement stems from the fact that the reconfiguration mechanism of an RPUF is uncontrollable and unpredictable: the new PUF responses will only be known after the reconfiguration has occurred.

Whenever an object $\mathcal{O}_i$ changes, the following steps must be performed:

1. A copy of the path from $\mathcal{O}_i$ to the root hash $n_{\mathcal{T}}$ (see Figure 4.3) is temporary stored in non-volatile memory.

2. The path in the authentication tree is updated: a new object nonce $n'_i$ is generated, the modified object $\mathcal{O}'_i$ is authenticated and encrypted with the fixed key $k_{\mathcal{T}}$ and the new nonce $n'_i$ (yielding a new object ciphertext $c'_i$ and object tag $\sigma'_i$). The intermediate hashes and root hash are recomputed.

3. The first RPUF is reconfigured, its new response is read, and the Gen algorithm is run to derive a fresh key $k'_{\text{auth}_1}$.

4. The updated root hash $n'_{\mathcal{T}}$ is authenticated with the new first key:

$$\sigma'_{\mathcal{T}_1} = \mathcal{H}_{k'_{\text{auth}_1}}(n'_{\mathcal{T}})\,.$$

5. The second RPUF is reconfigured, its new response is read, and the Gen algorithm is run to derive a fresh $k'_{\text{auth}_2}$.

6. The updated root hash $n'_{\mathcal{T}}$ is authenticated with the new second key:

$$\sigma'_{\mathcal{T}_2} = \mathcal{H}_{k'_{\text{auth}_2}}(n'_{\mathcal{T}})\,.$$

7. The backup copy (made in step 1) is deleted.

Three failure scenarios can be distinguished:

[7] We use a Merkle tree as an example, but the same approach can be taken for other authentication trees.

- If a failure occurs before step 3, the trusted module will use the backup copy that was made at the beginning, to rollback to the previous state $\mathcal{T}$. Any changes that have been done to the authentication tree have to be undone.
- A failure during step 3 or 4 results in $\sigma_{\mathcal{T}_1}$ and $k_{\mathrm{auth}_1}$ being out of sync. The integrity of the previous state $\mathcal{T}$ can still be validated with $\sigma_{\mathcal{T}_2}$ which has not been modified. All changes that have been made to the authentication tree must be rolled back. Furthermore the first RPUF must be reconfigured, yielding a new key $k''_{\mathrm{auth}_1}$, and the old root hash $n_{\mathcal{T}}$ must be authenticated with the new first key:

$$\sigma''_{\mathcal{T}_1} = \mathcal{H}_{k''_{\mathrm{auth}_1}}(n_{\mathcal{T}})\,.$$

- If a failure occurs after step 4, the trusted module will use $\mathcal{T}'$ as its state. The update process must be restarted from step 5, because otherwise $\sigma_{\mathcal{T}_2}$ remains invalid.

Note that three PUFs are needed in total: one static PUF to store the fixed key $k_{\mathcal{T}}$ and two reconfigurable PUFs to derive the dynamic authentication keys $k_{\mathrm{auth}_1}$ and $k_{\mathrm{auth}_2}$.

### 4.3.5 Discussion

#### Implementation Cost

The RPUF-based protection scheme comes at a cost in chip area. Additional hardware resources are needed for the fuzzy extractor, a symmetric cipher and a hash function. In most systems, adding these additional functions is relatively cheap. The main cost will be the implementation cost of the actual PUFs.

For static PUFs there is a good understanding of the practical cost since various prototype implementations, both in Application Specific Integrated Circuits (ASICs) and on FPGAs, have been demonstrated. Moreover, IP cores for intrinsic PUFs are commercially available today: Intrinsic-ID commercializes SRAM-based PUFs and Verayo offers solutions based on delay variations (i.e., arbiter PUF and ring oscillator PUF).

However, right now it is impossible to predict the implementation cost of an RPUF because a practical realization of RPUFs in standard CMOS technology has yet to be found. The working principle of the optical RPUF has been experimentally verified, but this is not a good candidate for integration into an

IC. On the other hand, the PCM-based RPUF construction seems a promising concept, but it still has to be practically verified. Furthermore, it is unclear at what cost PCM can be integrated in a trusted module. At the moment embedded Flash memory is probably cheaper than on-chip phase-change memory.

In addition, the protection scheme introduces an overhead in memory wear, as the persistent state is encrypted. Whereas minor state updates normally only require a small portion of NVM to be changed, now at least one encrypted block, as well as a number of nodes in the authentication tree need to be updated.[8] We believe though that the total overhead of write operations into the memory is relatively small, and standard techniques used in NVM memory controllers (such as relocation of 'hotspots') can be used to mitigate the additional effects.

### Logically Reconfigurable PUFs

In [163] we observed that the security properties of an RPUF can be also be achieved with the combination of a static PUF and non-volatile memory[9] and we gave the example of coating PUF on top of NVM. Katzenbeisser et al. introduced the name Logically Reconfigurable Physical Unclonable Function (LRPUF) for this type of construction [146].

We proposed to split the challenge $c_{\text{int}}$ that is applied to the static PUF into two parts: the challenge $c_{\text{ext}}$ that is exposed using the external interface, and a reconfiguration state $s_{\text{reconf}}$ that is stored in the NVM. When the response of logical RPUF is read, the parts of the challenge are combined:

$$r \leftarrow \mathsf{Read}(c_{\text{int}}) \quad \text{with} \quad c_{\text{int}} = c_{\text{ext}} || s_{\text{reconf}} \,.$$

A reconfiguration of the LRPUF corresponds with updating the state $s_{\text{reconf}}$. It should not be possible to revert to a previous reconfiguration state, so the state must be generated inside the LRPUF with a RNG or as a monotonic counter.

In order to protect against invasive replay attacks on the LRPUF, it is crucial that the static PUF and the NVM are tightly integrated. Attempts to access the embedded NVM from the outside should significantly change the responses of the PUF. That is why we proposed to use a protective coating as static PUF.

[8] It should be noted though that the description of our scheme omits several optimizations. It is possible, for example, to have an unbalanced authentication tree with dynamic items (such as counters) close to the root and in small blocks, while more static objects (such as fixed cryptographic keys) can be in bigger blocks and further from the root of the authentication tree).

[9] The construction of Lim [176] that is based on an arbiter PUF and floating gate transistor, can be considered as the first proposal for a logical RPUF. The controlled PUF construction with "multiple personalities" of Gassend et al. [104] also strongly resembles a logically reconfigurable PUF.

In 2011 Lao and Parhi [172, 173] and Katzenbeisser et al. [146] proposed alternative LRPUF constructions. Moreover, schemes implementing a PUF on an FPGA can be considered LRPUFs. By changing the FPGA's configuration bitstream, which resides in NVM, to implement a different PUF (e.g., another construction or another location in the reconfigurable logic) the challenge-response behavior will be modified [187]. These constructions do not satisfy our (informal) definition of an RPUF, which states that "*the reconfiguration mechanism has to be uncontrollable and should not be based on updating a hidden parameter*" and that "*the PUF reconfiguration must be difficult to revert.*"

### Non-Volatile Memory versus Reconfigurable PUFs

Our research on reconfigurable PUFs has been motivated by the fact that this new security primitive offers a promising alternative for on-chip non-volatile memory and hence that it can be used to securely store the persistent state of a security module in external memory. However, above we have introduced the concept of LRPUFs which requires embedded NVM, and we presented an RPUF construction based on a novel NVM technology namely phase-change memory. This leads to an intriguing situation, where the distinction between reconfigurable PUFs and NVM is vague and where one might wonder whether RPUFs make any sense.

We try to address this confusion by making three remarks.

- Logically reconfigurable PUFs might be a useful security primitive for some of the applications described in [146]. However, it is our belief that they are a contrived solution for the state protection problem: it makes more sense to derive a fixed key directly from the PUF response and use the embedded MTP NVM to store a replay-detection value instead (see Section 4.2.2).

- As shown with the PCM-based construction, novel NVM technologies are good candidates for the physical realization of an RPUF. Two requirements must be met: the read operation on the memory cell should be (fairly) stable,[10] but the write operation must be uncontrollable. This suggests that experimental technologies that are not reliable enough as NVM, could potentially be used as an RPUF.

- We suspect that an uncontrollable physical reconfiguration (e.g., melting a phase-change material) is more secure against physical attacks than a logical reconfiguration (i.e., replacing a key in NVM with new random

[10] The fuzzy extractor can correct a certain noise level and remove a bias.

number). Note however that invasive microprobing on the read interface of an RPUF or of NVM will likely be similarly successful.

## 4.4 Extending the Security Perimeter of the Trusted Module

In Section 4.2 we studied how the non-volatile state of a trusted module can be securely stored in external NVM and we introduced two generic approaches to do so. The first approach relies on a fixed key to protects the state's confidentiality and integrity and an additional dynamic value for state replay detection. The second scheme uses a single dynamic key that is altered on every state update. We identified that the appropriate technology is available to store fixed cryptographic keys in a cost-effective way, namely OTP fuses and intrinsic PUFs. However, we argued that today's MTP NVM solutions, except perhaps battery-backed RAM, are not fully suited for low-cost integration into mass-produced ICs. In Section 4.3.3 we presented reconfigurable PUFs as an enabling technology for secure updatable key storage and suggested them as an alternative for conventional reprogrammable NVM. However, we failed to provide a convincing practical realization of an intrinsic RPUF.

We determined in [228] that the most promising approach, until low-cost embedded MTP NVM becomes commonly available, is to *extend the security perimeter* of the trusted module to another hardware component that already has NVM. Understandably the communication interface between the two components must be protected with a cryptographic protocol. Otherwise the attacks that we demonstrated on the TPM's bus interface (see Chapter 3) could be applied.

### 4.4.1 Non-Volatile State Protection with External Authenticated NVM

In Figure 4.8 we give an example that applies the principle of security perimeter extension. This example is a simplification of the scheme that we presented in [228].

The main idea of our proposal is to use a state protection scheme that relies on a fixed key $k_{\mathcal{T}}$ and an updatable nonce $n_{\mathcal{T}}$ (as generically described in Section 4.2), and to securely externalize $n_{\mathcal{T}}$ with a special memory interface that provides mutual authentication. The external NVM component exposes an authenticated memory interface, that will be described in Section 4.4.2, as well

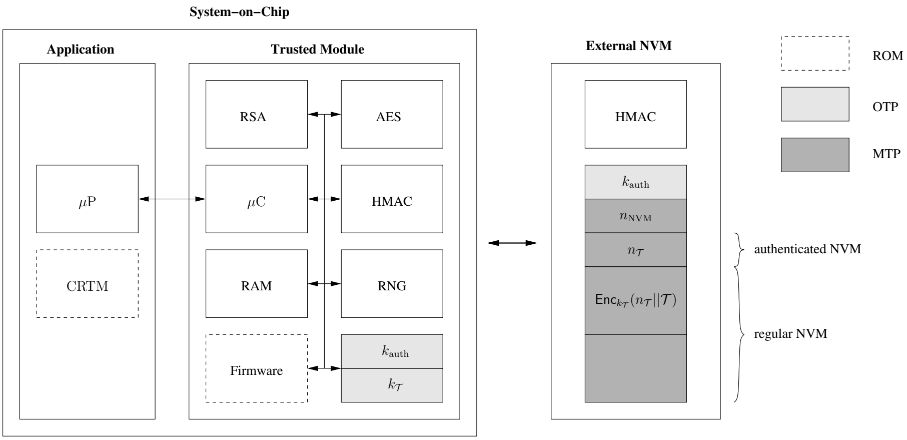


Figure 4.8: Non-volatile state protection with external authenticated NVM.

as a regular interface. The trusted module uses the legacy interface to store the protected non-volatile state[11] and the authenticated interface to store the nonce $n_{\mathcal{T}}$.

We tried to minimize the hardware primitives that are needed to implement the authenticated memory interface. A MAC algorithm, an OTP key store containing $k_{\text{auth}}$, a monotonic counter $n_{\text{NVM}}$, and some additional control logic must be added to the external NVM chip. The building blocks that must be included in the trusted module for the memory authentication protocol, are a MAC algorithm, a hardware RNG and an OTP key store holding $k_{\text{auth}}$. Additionally, to securely disembed $\mathcal{T}$, the trusted module requires a symmetric AE algorithm $\mathsf{Enc}$ and some embedded OTP memory holding $k_{\mathcal{T}}$.

Remember that TPMs and MTMs must already include the HMAC-SHA-1 algorithm and an RNG. So the overhead for the trusted module is limited to the symmetric cipher and a limited amount of embedded OTP NVM.

We believe that our solution divides the security responsibilities in a sensible way. The external NVM module is only entrusted with the task of state replay detection, while the trusted module is responsible for state confidentiality and state integrity protection. This segregation of responsibilities has two advantages:

[11] In Figure 4.8 we denote the protected state as $\mathsf{Enc}_{k_{\mathcal{T}}}(n_{\mathcal{T}}||\mathcal{T})$. However, as explained in Section 4.2.3 and 4.2.5 the updatable nonce $n_{\mathcal{T}}$ could also be the root of an authentication tree or it could be used as the IV of the AE scheme (i.e., $\mathsf{Enc}_{k_{\mathcal{T}}}(n_{\mathcal{T}}||\mathcal{T})$).

Table 4.2: Access control table of external NVM.

| | key<br>$k_{\text{auth}}$ | counter<br>$n_{\text{NVM}}$ | regular<br>memory | authenticated<br>memory |
|---|---|---|---|---|
| read | never | optional | always | optional |
| write | once | never | always | never |
| authenticated read | never | always | optional | always |
| authenticated write | optional | never | optional | always |

- The hardware requirements for the external NVM chip and hence the additional cost are low. For instance, a scheme that stores the persistent state in plaintext inside the external NVM and encrypts it on-the-fly when transmitted over the NVM interface, will be more expensive because a symmetric cipher has to be included in the external NVM chip. Also note that the external NVM does not need a hardware RNG in our proposal.
- The security impact of a compromised external NVM component is limited. If an adversary extracts the authentication key $k_{\text{auth}}$ from the external NVM, he can only replay old versions of the state. He cannot read the non-volatile state nor make arbitrary modifications to it, as this requires knowledge of key $k_{\mathcal{T}}$.

Remark that the regular external NVM and the authenticated external NVM do not necessarily have to be offered by the same hardware component, as shown in Figure 4.8. The scheme can also be implemented with three discrete components: the (embedded) trusted module, a standard Flash memory chip storing the encrypted state, and a third security-enhanced NVM component hosting the replay-detection nonce. This type of setup is for instance suggested in a patent of Nokia [11].

### 4.4.2 Memory Authentication Protocols

As shown in Figure 4.8, the external NVM is divided into four parts: a key store that contains $k_{\text{auth}}$, a monotonic counter $n_{\text{NVM}}$, a memory range that is protected with a mutual authentication protocol, and a memory range that is accessible with the regular memory interface. Our proposal only stores the replay-detection nonce $n_{\mathcal{T}}$ in the authenticated external NVM. The regular external NVM is used to store the externalized state $\mathsf{Enc}_{k_{\mathcal{T}}}(n_{\mathcal{T}}||\mathcal{T})$ and other data, such as program code, configuration data, user data, etc.

Furthermore, the external NVM chip provides a regular interface as well as an authenticated interface. Table 4.2 lists which operations are allowed on which parts of the external NVM. The main restrictions that must be enforced are the following:

- The key $k_{\mathrm{auth}}$ can never be read and it can be programmed once with a regular write operation.
- The monotonic counter $n_{\mathrm{NVM}}$ is read only.
- A regular write operation fails on the authenticated memory range.

The support of some combinations is optional because they are not used in our scheme, e.g., reading the content of the authenticated memory with a regular read operation.

### Authenticated Read Operation

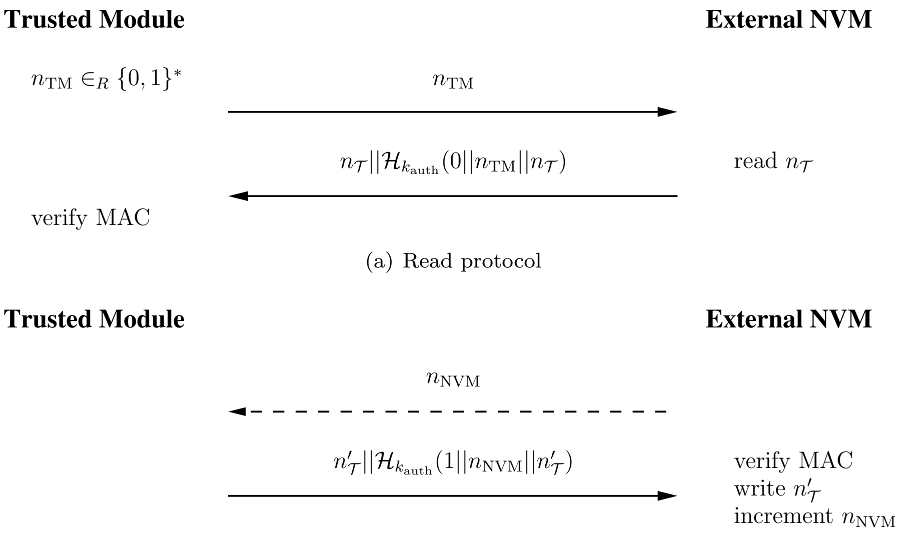
In order to read the nonce $n_{\mathcal{T}}$ from the external authenticated NVM, the following steps must be performed (see Figure 4.9(a)):

1. The trusted module generates a random challenge $n_{\mathrm{TM}}$ and sends it to the external NVM.
2. The external NVM reads the nonce $n_{\mathcal{T}}$ from its internal memory.
3. The external NVM returns $n_{\mathcal{T}}$ accompanied with a MAC on the value 0, the challenge $n_{\mathrm{TM}}$ and the data $n_{\mathcal{T}}$.
4. The trusted module verifies the MAC. If verification fails, the trusted module aborts.

Afterwards the trusted module can check whether the nonce $n_{\mathcal{T}}$ that it just read, corresponds with the value that is embedded in the encrypted state $\mathsf{Enc}_{k_{\mathcal{T}}}(n_{\mathcal{T}}||\mathcal{T})$.

### Authenticated Write Operation

In order to write the new nonce $n'_{\mathcal{T}}$ to the external authenticated NVM, the following steps must be performed (see Figure 4.9(b)):


**Trusted Module**
**External NVM**
$n_{\text{TM}} \in_R \{0,1\}^*$
$n_{\text{TM}}$
$n_{\mathcal{T}} || \mathcal{H}_{k_{\text{auth}}}(0 || n_{\text{TM}} || n_{\mathcal{T}})$
read $n_{\mathcal{T}}$
verify MAC
**Trusted Module**
**External NVM**
$n_{\text{NVM}}$
$n'_{\mathcal{T}} || \mathcal{H}_{k_{\text{auth}}}(1 || n_{\text{NVM}} || n'_{\mathcal{T}})$
verify MAC
write $n'_{\mathcal{T}}$
increment $n_{\text{NVM}}$

(a) Read protocol

(b) Write protocol

Figure 4.9: Authenticated memory interface to access external NVM.

1. The trusted module sends $n'_{\mathcal{T}}$ accompanied with a MAC on the value 1, the current counter value $n_{\text{NVM}}$ and the data $n'_{\mathcal{T}}$.
2. The external NVM determines whether the trusted module knows the current counter value $n_{\text{NVM}}$ by checking the MAC. If the verification fails, the external NVM aborts.
3. The external NVM writes the new value $n'_{\mathcal{T}}$ to its internal memory and increments its monotonic counter.
4. The trusted module verifies whether the nonce $n'_{\mathcal{T}}$ was written correctly by performing the authenticated read protocol from Figure 4.9(a).

In the first step we assume that the trusted module knows the current counter value $n_{\text{NVM}}$, which acts as the challenge $n_{\text{NVM}}$ of the external NVM. This implies that the trusted module maintains a copy of the monotonic counter in volatile RAM memory. This local copy is also incremented after every authenticated write operation. However, if the counter value gets lost (e.g., because of a power cycle of the trusted module), the trusted module can retrieve it with an authenticated read operation on the memory address of the external monotonic counter.

The trusted module needs to perform step 4 in order to determine whether the write operation was successful. In [228] we described a slightly different variant of the authenticated write operation, in which the external NVM returns $\mathcal{H}_{k_{\mathrm{auth}}}(n_{\mathrm{NVM}}+1)$ as an explicit confirmation that the write operation succeeded.

### Security Analysis

The authenticated read operation is a two-pass challenge-response protocol that provides unilateral authentication of the external NVM component, whereas the authenticated write operation is a one-pass stateful protocol that provides unilateral authentication of the trusted module. Mutual authentication is achieved when both operations are combined; e.g., by reading the current counter value $n_{\mathrm{NVM}}$, subsequently writing the new value $n'_{\mathcal{T}}$ and finally confirming that the write operation was successful with an additional read operation.

Both operations use the one-way function $\mathcal{H}$ keyed with the shared secret $k_{\mathrm{auth}}$. If this long-term key is compromised, an attacker can impersonate the trusted module in the write operation and the external NVM in the read operation. The protocols rely on nonces for freshness: the random number $n_{\mathrm{TM}}$ for the read operation and the synchronized monotonic counter $n_{\mathrm{NVM}}$ for the write command. It is important that these nonces are never reused.

The fixed values 0 and 1, that are included in the authentication tag to indicate a read and a write operation respectively, are essential for the security of the protocol. If they would be removed, then the protocol can be attacked as follows: the adversary performs an authenticated read operation with a future counter value $n_{\mathrm{NVM}} + i$ as challenge and gets the authentication tag $\mathcal{H}_{k_{\mathrm{auth}}}(n_{\mathrm{NVM}}+i||n_{\mathcal{T}})$ as response; next he waits for $i$ legitimate write operations by the trusted module, and finally he uses the response that he got earlier to overwrite the future version of replay detection with the old version $n_{\mathcal{T}}$.

## 4.4.3 Practical Aspects

### Pairing the Trusted Module and the External NVM

For simplicity reasons we assume that a trusted entity generates the authentication key $k_{\mathrm{auth}}$ and programs it in the trusted module and the external NVM. This *pairing* phase must take place in a secure environment, typically in the manufacturing plant of the end device that embeds the trusted module. At the same time the trusted entity programs the state protection key $k_{\mathcal{T}}$ in the trusted module.

The external NVM provides a simple interface that guarantees one-time-programmability of $k_{\text{auth}}$: one ordinary write operation can be performed on the address of the key store and all subsequent read and write operations to this memory address will fail (see Table 4.2). For various other options to implement OTP Flash memory we refer to the work of Handschuh and Trichina [127, 128].

Note that, once the keys $k_{\mathcal{T}}$ and $k_{\text{auth}}$ are programmed, the internal state of the trusted module will be initialized. This process includes the generation of the EK (for a TPM or MLTM) and the SRK (for an MRTM), the creation of an endorsement certificate on the public EK, programming of the necessary information for secure boot, etc.

### Lifecycle Management

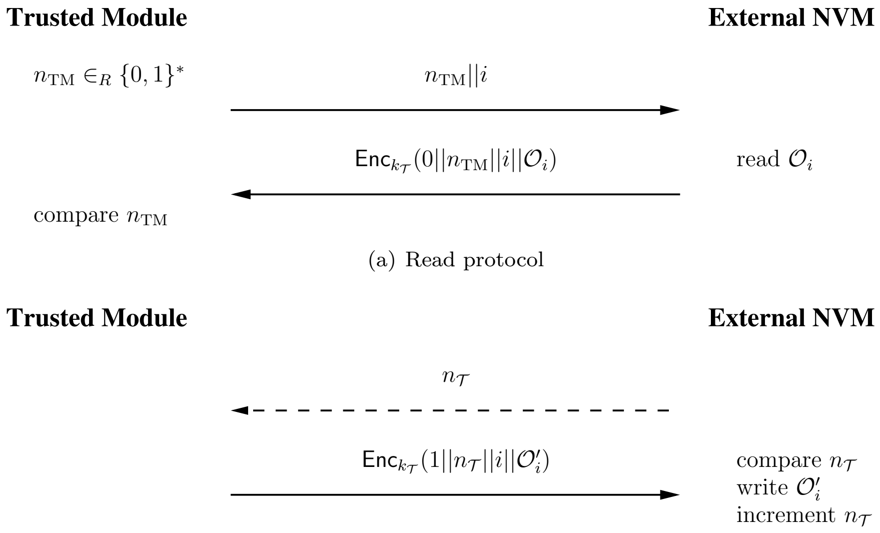
The focus of our research was the definition of a secure and low-cost NVM authentication scheme. We paid less attention to practical aspects such as testability, field replaceability and auditability. Some of these aspects can be addressed by supporting a reset operation that puts the external NVM in an uninitialized state: this operation erases the key $k_{\text{auth}}$, the monotonic counter $n_{\text{NVM}}$ and the content of the authenticated memory range (i.e., $n_{\mathcal{T}}$). For an elaborate discussion on lifecycle management for external authenticated memory we refer to the work of Ekberg and Asokan [89].

## 4.4.4 Alternative Segregation of Responsibilities

We briefly describe alternative ways to extend the security perimeter of the trusted module. The underlying concept remains the same: the trusted module relies on the non-volatile memory of an additional hardware component and the communication between the two components is protected with a cryptographic protocol.

### Entrusting the External NVM with State Integrity Protection

The above solution only utilizes the authenticated memory of the external NVM to store the nonce $n_{\mathcal{T}}$, while the state itself is stored in the regular external NVM. In the scheme that we originally proposed in [228], the state is stored completely with the authenticated memory interface. This implies that the monotonic counter $n_{\text{NVM}}$ of the external NVM assumes the role of the state replay nonce: $n_{\text{NVM}} = n_{\mathcal{T}}$. The trusted module is still held responsible for the


Trusted Module
External NVM
$n_{\mathrm{TM}} \in_R \{0,1\}^*$
$n_{\mathrm{TM}}||i$
$\mathsf{Enc}_{k_{\mathcal{T}}}(0||n_{\mathrm{TM}}||i||\mathcal{O}_i)$
read $\mathcal{O}_i$
compare $n_{\mathrm{TM}}$
Trusted Module
External NVM
$n_{\mathcal{T}}$
$\mathsf{Enc}_{k_{\mathcal{T}}}(1||n_{\mathcal{T}}||i||\mathcal{O}'_i)$
compare $n_{\mathcal{T}}$
write $\mathcal{O}'_i$
increment $n_{\mathcal{T}}$

(a) Read protocol

(b) Write protocol

Figure 4.10: Encrypted memory interface to access external NVM.

state's confidentiality. This also means that the non-volatile state can be stored in the external authenticated NVM as $\mathsf{Enc}_{k_{\mathcal{T}}}(\mathcal{T})$.

### Entrusting the External NVM with State Confidentiality Protection

One step further is to also make the external NVM responsible for the state confidentiality and integrity. In Figure 4.10 we describe an encrypted memory interface to accomplish this. The MAC algorithm is replaced by an AE scheme $\mathsf{Enc}$ that is keyed with the shared secret $k_{\mathcal{T}}$. Note that there is no longer a need for two separate keys $k_{\mathrm{auth}}$ and $k_{\mathcal{T}}$.

An additional advantage of this approach is that the non-volatile state can be read and updated in smaller logical object $\mathcal{O}_i$ instead of processing the state as one whole.

## 4.5 Conclusion

In this chapter we investigated the integration of a trusted module into a system-on-chip design that lacks embedded reprogrammable non-volatile memory. In particular, we studied how to securely store the persistent state of the trusted module in external memory. Practical examples where this is a requirement, are the integration of a TPM into the chipset of a PC, the implementation of the MTM specification in the trusted execution environment of an application processor, and the implementation of a trusted module on a volatile FPGA.

We presented two generic approaches to protect the externalized state. Both approaches authenticate and encrypt the persistent state with a secret key that is embedded in the trusted module. The detection of state replay, which is the most challenging security property to accomplish, is done differently. The first approach generates a new state protection key every time the persistent state is updated. Older versions of the state are rendered invalid because they are authenticated and encrypted with a different key than the newly generated key. In the second approach the state protection key remains fixed, but a fresh nonce is included in the persistent state on every update.

We also discussed techniques to improve the efficiency of the schemes. Normally, a small update in the persistent state (e.g., incrementing a monotonic counter) would require that the trusted module re-encrypts and re-authenticates the complete state. However, by splitting the state into logical objects and by authenticating these objects with an authentication tree, the overhead of the state protection scheme can be reduced. When an object is changed, only the path to the root of the tree has to be modified.

The generic approaches that we introduced, still requires a small amount of non-volatile memory inside the trusted module for the storage of the secret key and (optionally) the replay detection nonce. In this chapter we made the fundamental assumption that floating gate based Flash memory or EEPROM cannot be embedded cheaply in a system-on-chip design; hence the need to externalize the persistent storage. Consequently, alternative NVM technologies, such as battery backed RAM, fuses and PUFs, must be used.

PUFs have been proposed in the literature as a technique for key storage and this technology is starting to be used in commercial applications. In this chapter, we proposed the concept of a reconfigurable PUF, which is a PUF with a reconfiguration mechanism to irreversibly alter its challenge-response behavior. This reconfiguration functionality can be used to modify the key that is derived from the PUF response. Consequently this new primitive can be utilized in a state protection scheme that relies on an updatable secret key. We have also presented a formal definition of an RPUF, as well as two RPUF constructions.

Finally we proposed to extend the security perimeter to the external NVM as an alternative to including MTP NVM in the trusted module. In this case, a cryptographic protocol protects the communication between the trusted module and the external memory. The external memory can be entrusted with different responsibilities, ranging from the external memory being only responsible for state replay detection to a fully authenticated and encrypted memory interface.

Chapter 5

# Flexible TPM Architecture

With the utilization of TPM-based trusted platforms in real applications, and the subsequent adaption of the specification to the experience gained from such utilization, it increasingly appears that the TPM architecture has some fundamental flaws that result in more and more complex and expensive hardware requirements.

In this chapter, we propose a new architecture, which we presented in [164], to reset the trust boundary to a much smaller scale, thus allowing for simpler and more flexible TPM implementations, without sacrificing the security gains from a classical TPM.

## 5.1 Introduction

As explained in Chapter 2 TPMs were introduced by the TCPA to provide a low-cost, universal building block on which platforms could build systems to provide a certain level of trust to the platform. However, over time, the TPM specification has gained substantially in scope and complexity to accommodate different functional requirements. As explained in Section 3.1.1, a number of mostly minor flaws and inconsistences have been identified in the specifications [39, 54, 55, 120, 178] and a study by Sadeghi et al. has revealed that some vendors already struggle with the corresponding implementation complexity [223]. Also, for the more cost sensitive mobile world, a different specification, namely the MTM, was required to keep the complexity to a manageable level. In practice, however, great effort is still needed to implement an MTM in a resource-constrained

TrEE [90].

As a consequence, current TPM implementations have moved away from the original philosophy of the TCPA, namely, a simple, easy to implement and verify module that serves as a foundation for trust in the platform. In addition, thanks to increasing interest and utilization of TPMs, more requirements for additional functionality are generated, e.g., support of multiple executions for virtualized machines [24], support for distributed protocols, etc.

In this chapter, we investigate how TPM functionality can be achieved with simpler hardware, and with greater flexibility towards supporting specialized application demands. To this end, we re-investigate which of the functionality needs to be implemented inside the trust boundary (i.e., in the trusted hardware), and which parts can safely be externalized onto the host platform. By externalizing large parts of the TPM implementation, we arrive at a hardware base that is smaller in size, more flexible to use, and simpler to implement and verify.

In Chapter 4 we already illustrated that the non-volatile state of a TPM can be securely externalized. In [164] we presented a flexible TPM architecture that goes even further by also "disembedding" the TPM's firmware. In the remainder of this chapter we will describe this alternative TPM architecture, which we named $\mu$TPM.

### 5.1.1 Related Work

Chevallier-Mames et al. [57] proposed a theoretical blueprint of a ROM-less smart card called Externalized Microprocessor (X$\mu$P). All the executable code of the X$\mu$P is stored in the terminal and program instructions are fetched when needed. The advantages of a ROM-less secure token are numerous: chip masking time disappears, bug patching becomes a mere terminal update and hence does not imply any roll-out of cards in the field, and most importantly, code size ceases to be a limiting factor. The terminal is considered untrustworthy and hence the X$\mu$P must authenticate the external program code. Chevallier-Mames et al. propose a number of public-key oriented alternatives to authenticate the disembedded code: verification per instruction, batch verification with RSA screening, and verification of code sections. In the extended version of the paper [58] MAC-based variants are also given.

In [87] we investigated how to embed a TPM in a reconfigurable SoC design. We defined a new FPGA architecture that includes a minimal root of trust called Bitstream Trust Engine (BTE). This novel architecture not only enables field updates of the TPM's firmware, but also of the TPM hardware (i.e., a partial configuration bitstream). The BTE component records the partial configuration bitstreams that are loaded on the reconfigurable logic, and limits access to the FPGA's internal NVM. We will describe this proposal in detail in Chapter 6.

Costan et al. [65, 66] proposed the Trusted Execution Module (TEM) as a more flexible alternative for TPMs. The TEM can execute arbitrary general purpose applications, which are split into executable fragments called *closures*. The closures are encrypted with the TEM's public encryption key, guaranteeing that only designated modules can run the application. In [164] we independently proposed the idea of splitting a security application into atomic tasks as a strategy to minimize the memory footprint of the trusted module. However, we took a different approach than Costan et al. Our $\mu$TPM architecture explicitly supports remote attestation; i.e., it proves to an external entity with a digital signature that a result is produced in an identifiable module. The TEM scheme, on the other hand, guarantees that only a designated module can run a certain application by encrypting the application with the module's public key.

In [90] Ekberg and Bugiel demonstrated a practical implementation of a minimal MRTM that runs in the M-Shield TrEE of a Nokia N96 handset. Because the secure RAM of their target application processor is limited in size (around 7 kB), the implementation is divided into parts (*collections*), individually minimized in terms of code and data size. In particular, they have grouped the MRTM commands by size and function into 12 collections of 1-4 commands each. Depending on the command to be executed, one of these collections is loaded into the secure environment and executed. The integrity of the disembedded code collections is maintained by the underlying M-shield security architecture, by means of digital signatures. In a similar fashion, the state for the MRTM(s) is loaded and returned with every command invocation. The confidentiality and integrity of the externalized state is protected with AES in CBC mode and HMAC-SHA-1. They proposed a list of state size optimizations, mainly targeting the key structures. The work of Ekberg and Bugiel proves the practical viability of the code/state disembedding principle.

Dietrich and Winter also built upon our work. In [75] they use the principle of dynamic command loading to implement flexible MTMs on the resource-limited TrEE of a mobile phone. The potential advantages that they listed, are algorithm flexibility (e.g., replacing the SHA-1 function or the RSA-based DAA scheme) and field updates (e.g., installing optional MTM commands on modules that have already been deployed in the field). They described two implementations, the first on an ARM TrustZone TrEE and the second on a JavaCard-based Secure Element (SE). For instance, in the case of the JavaCard implementation, they split the MTM into different JavaCard applets, each implementing a set of commands. A master applet that is always present provides access to MTM's state using a shared interface and the command applets are loaded and unloaded when needed.

### 5.1.2 Towards an Alternative TPM Architecture

With the work on trusted computing progressing over the last ten years, the design of current TPM implementations is starting to show the first limitations. The primary problem is that the hardware is supporting both too much and too little functionality. In attempting to accommodate all requirements from different users of the technology, the "one-size-fits-it-all" approach towards TPMs has lead to huge implementations, making the hardware complex, hard to verify, and expensive. Each of those issues has already started to cause problems: a different specification was required for more cost-sensitive mobile devices [91], the complexity has been difficult to manage for some manufacturers [223], and governments keep worrying about verifiability of the security critical component [40]. With future specifications, this problem is likely to get only bigger rather than smaller. While few commands are likely to disappear, experience with TPMs in real platforms has lead to new requirements that future versions will have to accommodate (e.g., cryptographic algorithm agility and advanced virtualization support).

We see that one way to approach these issues is to redefine the trust boundaries of the TPM architecture. By putting more functionality of a TPM outside of the trusted hardware, we increase flexibility (as outside mechanisms can easily be adapted, and usually have more resources at their disposal than functionality implemented inside the TPM), and reduce the size of the critical hardware components that we need to protect. To this end, we propose to remove the entire TPM code base (i.e., the implementation of the TPM functionality) outside of the secure hardware. As with other TPM related storage (see Chapter 4), there is no security requirement to store this data inside the trusted hardware, as long as the TPM can authenticate it properly.[1] The TPM trust boundary then only needs to incorporate an authentication key as well as enough RAM memory to store the command code during execution.

While this approach does require new mechanisms for code authentication and attestation, the functionality of the hardware can now be reduced to a very small number of commands, and future extensions to the functionality can be added without needing to modify the hardware specification. More importantly though, externalizing the code from the hardware adds a new degree of flexibility that can handle most of the limitations current TPM implementations face:

[1] For protection of intellectual property, a producer may also want to encrypt the data; for the TPM security functionality this is not required though.

**Verifiability.** Given that the TPM executes security relevant code, users (especially government related users) do want assurance that the TPM is implemented properly. By separating code and hardware, the hardware becomes substantially easier to verify, while it can be assured that the code has not been modified for a particular TPM. Also, it is possible to separate code and hardware implementation, allowing for alternative implementations. If the standard proposes a generic programming language (e.g., basing the TPM on a JavaCard runtime environment [72]) it is even possible for pure software vendors or the open source community to provide different TPM implementations. In addition, users with special security needs can use their own, individual code.

**Customizability.** TPM functionality becomes easier to extend or to limit. Current TPMs suffer from the fact that platform producers demand an increasing amount of functionality, while TPM manufacturers want to decrease the implementation complexity. With externalized commands, each platform can supply the commands needed for its particular operation, and omit functionality designed for different platforms. This even allows very special-purpose commands in the TPM, or even freely programmable code. The only limit here is that guarantees given by the TPM according to the TPM specification (e.g., the protection of certain keys) are never violated. This is assured by having one manufacturer authenticate their (external) firmware, and not allowing other processes to access the resources of that implementation.

**Upgrades.** TPM code can easily be updated in the field; the real hardware does not carry any functionality anymore, allowing to fix implementation bugs, retire insecure algorithms (such as SHA-1), or upgrading to newer versions of the TPM specification. To upgrade the TPM code, the old code simply can be replaced by (authenticated) new code. It must be taken care of though that the two versions cannot be mixed, as this may cause unpredictable behavior.

**Multiprocessing.** Several TPMs can be implemented on the same hardware. This is, for example, helpful for virtual machines running on the same server, which is a growing application for TPM usage and causes problems that are difficult to solve with only one hardware TPM. In addition, it allows for TPMs with different functionalities or for several MTMs protecting different stake holders.

**Specialized TPM implementations.** Current TPMs are designed in a way that the same hardware is used in high-end servers and low-end embedded devices. In our proposed settings, low-end TPMs can implement the minimal hardware necessary to run, while high-end server TPMs can offer

> more memory, command cache and faster process management hardware to allow for rapid context switches. It is also possible for very low-end devices to build a TPM that does not have the resources for all commands in the TCG standard, but supports the commands required for the particular platform. While such a device cannot claim TCG compliance, it can reuse hardware components and interface with TCG-compliant software.

While the goal is to add flexibility to and simplify the hardware of the TPM functionality, one requirement for our architecture is to allow for a functionality similar to current TPMs; i.e., the trust model and functionality of the TPM specification must be compatible with the new hardware architecture. We extend the TPM requirements by allowing the architecture to support parallel, independent processes; i.e., it is possible to run several different security co-processors in one hardware block without interference. This is relevant in the areas of virtualization, where one hardware TPM needs to support a number of virtual machines, each of which needs its own virtual TPM, or for the support of multiple TPMs in one platform acting on behalf of different stakeholders. The latter is for example proposed in the MTM specification, where separate logical TPMs are proposed to protect the mobile operator, the service provider and the end user, respectively. Allowing different logical TPMs with different code bases takes this approach further, even allowing individual applications (e.g., a banking application) to have a dedicated, specialized trusted module.

As with classical TPMs, performance is not a priority: TPMs are not meant to work as cryptographic accelerators, but as slow and reliable building blocks. Nevertheless, the rather small communication bandwidth of current TPMs makes communicating large code segments a relatively inefficient operation. We would argue, though, that this is not a real problem; most frequently used commands are rather simple, and thus have relatively short implementations – in the open source TPM emulator [252], most commands can be implemented in a few lines of C-code – with the notable exception of the DAA implementation, which, however, is so far not used in any real setting. As TPMs were never meant to perform fast, the additional delay is usually tolerable. An approach to speed up the scheme is to allow for a library of basic functions (e.g., the hash function) to be either pre-installed, or loaded into the TPM at startup and cached for future use.

## 5.2 $\mu$TPM Architecture

Our proposal for a new flexible Trusted Execution Environment (TrEE), which we call $\mu$TPM, was first presented in [164]. The $\mu$TPM architecture builds upon the X$\mu$P concept of Chevallier-Mames et al. [57] and the BTE proposal of Eisenbarth et al. [87], which will be described extensively in Chapter 6. The process and memory management of our proposal are inspired by the JavaCard architecture [56].

In this section we give a high-level conceptual description of the $\mu$TPM architecture and in the following sections we will describe the low-level interfaces and implementation options in more detail.

## 5.2.1 Design Principles

Figure 5.1 gives a schematic description of the $\mu$TPM architecture and highlights the main design principles. Conceptually the $\mu$TPM is a basic processor operating on dedicated RAM memory, which contains program code and volatile data, and NVM memory, which stores persistent state information. It has two logical I/O ports, denoted IO and XIO.[2] The IO port is the regular communication interface that is used to exchange data between software running on the host processor and a process executing inside the $\mu$TPM's runtime environment; e.g., the CRTM and TSS issue TPM commands and receive responses over the IO port. The XIO port on the other hand is used by an external software component called $\mu$TSS to manage the processes (e.g., creation, activation and suspension) and to load the firmware code (and optionally the externalized non-volatile state) in the $\mu$TPM execution environment.

### Disembedded Firmware

The core idea of our proposal is that program code that runs on the $\mu$TPM processor (i.e., the *firmware*) is stored outside the security perimeter of the $\mu$TPM execution environment and loaded over the XIO interface. This removes the need for on-chip ROM and consequently reduces the required hardware resources. More importantly, the disembedding of the firmware allows for a flexible, customizable and field-updatable TPM implementation. Furthermore, the functionality that runs on the $\mu$TPM is not only limited to the TCG specifications, but it can be arbitrary; consequently the $\mu$TPM architecture can act as a general-purpose secure coprocessor.

Note that on Figure 5.1 the $\mu$TPM contains embedded NVM. This memory is used to persistently store key material and program state information (see below), but it is not used to store program code.

[2] In practice the two interfaces will most likely use the same communication bus, e.g., LPC.

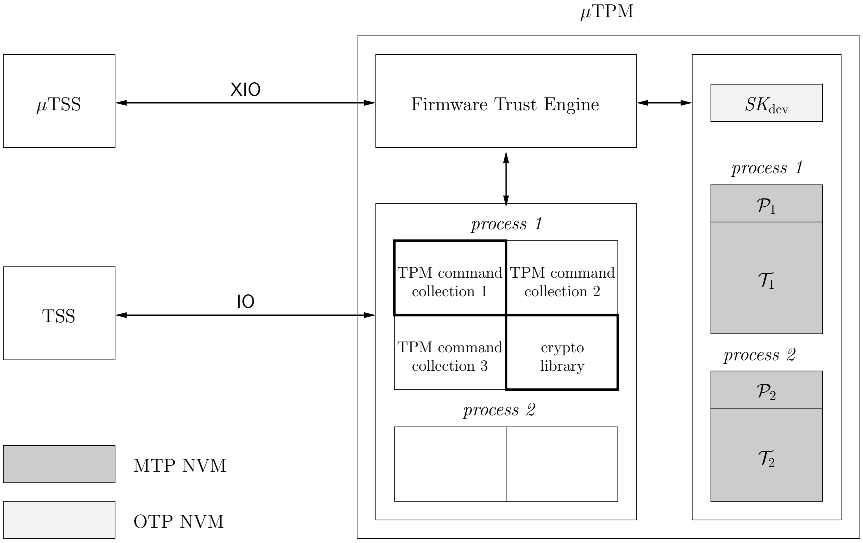


Figure 5.1: Conceptual $\mu$TPM architecture. The firmware of the TPM process is split into three command collections and one cryptographic library. Only one of the three command collections and the library, which are marked bold, are present in the RAM memory.

### Firmware Authentication

Since the firmware is stored in external – and hence potentially untrusted – NVM, it is essential that the authenticity of the disembedded firmware is verified while it is loaded in the RAM and before it is subsequently executed. If no mechanism is in place to authenticate the firmware, it is easy to load malicious code over the XIO interface that compromises (e.g., reads or modifies) the $\mu$TPM's internal state.

We want the $\mu$TPM architecture to be completely open and non-restrictive: there should be no restrictions on the code that can be run. For this reason we adopt the concept of *measured boot* (i.e., measuring code and afterwards reporting it with a protocol) instead of secure boot (i.e., stopping execution when code is not signed by the $\mu$TPM manufacturer).

With this design principle our proposal differs fundamentally from a JavaCard-based TrEE. The management of JavaCard runtime environments is defined in the GlobalPlatform (formerly Visa Open Platform) specifications [188]. In order to install applets on a JavaCard runtime environment, the applet provider must first authenticate to the GlobalPlatform card manager of the environment, which is responsible for card content management. This authentication step is typically based on a symmetric cipher such as 3DES and hence requires the knowledge of a shared secret key. Practically deployed smart cards often implement a fixed functionality and are rarely updated in the field. Hence, in many applications, for instance the Belgian Electronic Identity (eID) card, the functionality is implemented as a single JavaCard applet and the secret key for card content management is only known by the card issuer. The primary example of a JavaCard runtime environment that executes multiple applets of different stakeholders, is the SE of a NFC-enabled mobile phone. Various security applications can rely on the SEs: for identification of the mobile subscriber (i.e., Subscriber Identity Module (SIM)), digital rights management for mobile TV, contactless mobile payments, electronic vouchers (e.g., Google Wallet), etc. The content of the SE will be managed by the mobile network operator, by the phone manufacturer, or by an external party known as the Trusted Service Manager (TSM). For development purposes, the key for card content management can be set to a publicly known default value and any developer can install applets on the TrEE.[3]

We define a new trust anchor called Firmware Trust Engine (FTE) that deals with the firmware externalization and that manages the multiple execution contexts (see below). The FTE plays the same role in the $\mu$TPM as the TCG roots of trust, namely:

- Root of Trust for Storage by providing shielded locations to persistently store key material;
- Root of Trust for Measurement by measuring the identity of the externalized firmware when it is loaded in the $\mu$TPM environment and recording this measurement in a so-called Firmware Configuration Register (FCR); and
- Root of Trust for Reporting by signing the content of the FCR with the signature key $SK_{\text{dev}}$, that is certified by the $\mu$TPM manufacturer. We call this key the Hardware Endorsement Key (HEK).

In Section 5.2.2 and 5.2.3 we will describe how the FTE can isolate the $\mu$TPM processes and their associated states from each other, whereas Section 5.2.4 and 5.2.5 will cover firmware integrity measurement and reporting.

[3] For personal experience we know that it is not always easy to program security applications (in our case a mobile wallet for events) on NFC phones. The SE of the Nokia 6131 NFC phone can be unlocked for development purposes, such that its chip specific keys get revoked and reset to a public value. However, for more recent NFC-enabled Android phones, such as the Google Nexus S and Galaxy Nexus, this feature does not exist.

### Limited RAM Resources

Another important design choice is to keep the embedded RAM of the $\mu$TPM small. In order to minimize the RAM footprint of the TPM firmware, the TPM commands should be grouped into small *collections* of a limited number of commands. The collections can be as fine-grained as individual TPM commands or more coarse-grained, consisting of high-level TPM features such as integrity collection and reporting.[4] The command collections are loaded in and unloaded from the $\mu$TPM when needed, and operate on a common state, which resides in volatile RAM and/or persistent NVM.

We support the ability to load shared libraries, for instance with common cryptographic operations, that remain persistent in RAM memory, when different code collections are loaded. Figure 5.1 visualizes an example where the TPM firmware consists of three collections of commands and one cryptographic library and where the first collection and the library have been loaded in RAM memory; in this example the RAM memory must be big enough to contain the shared library and one collection of TPM commands.

### Multiprocessing Support

The $\mu$TPM architecture can execute multiple processes – either TPM instances or even arbitrary general purpose code – in parallel. At any given moment at most one execution context can be active. We assume that the scheduling of processes is done by a software component outside the trusted computing base of the $\mu$TPM. This external software component, which we call $\mu$TSS, indicates which process should run and it loads the externalized code into the running process. The $\mu$TPM guarantees strong process isolation: each process has access to a dedicated portion of the NVM, and the volatile memory, which stores the program's volatile state and code, is cleared securely during a process switch.

The design choice to schedule the processes outside $\mu$TPM is inspired by the JavaCard architecture, which supports also strictly isolated execution of multiple applets on a physical smart card. In the JavaCard architecture applets are explicitly activated by issuing a selection command with the identifier of the applet. Prior to calling the select method of the indicated applet, the JavaCard runtime environment shall first deselect the previously selected applet.

[4] A possible grouping of commands is the one mentioned in part 3 of the TPM main specification [283].

Note that the JavaCard architecture supports the ability to share objects between applets. We do not support this feature in order to keep the $\mu$TPM architecture simple.

While we design our architecture to allow for an arbitrary number of processes, realistically the number of processes running on a single $\mu$TPM will be relatively low. The only use case that we see for a large number of $\mu$TPM processes, is a server with many virtualized machines each with their own associated TPM. In this case, the manufacturer should choose a high-end $\mu$TPM with a large amount of internal memory and a fast communication bus. In other words, while we do not want to set an upper limit on the number of processes and their memory requirements, we are not trying to build an architecture that scales efficiently to hundreds of processes if the $\mu$TPM only has 1 kbit of RAM memory.

### Non-Volatile State

In Figure 5.1 and in the remainder of this chapter we assume that the $\mu$TPM has abundant embedded non-volatile memory to store the persistent state of the different processes as well as the internal key store of the FTE. It is clear that the state protection schemes that we have discussed in Chapter 4, can be applied to reduce the need for this embedded NVM; e.g., the state $\mathcal{T}_i$ of every process can be encrypted with a fixed key and authenticated with an authentication tree. If the persistent data storage of the $\mu$TPM is externalized, not only the code but also the non-volatile state must be loaded when a process is made active. Moreover, whenever the process modifies its state, the updated state must be returned to $\mu$TSS (as an authenticated, encrypted and replay detectable blob).

Observe that the scheduling of processes is done outside the $\mu$TPM. This has some implications for the development of a TPM implementation and other $\mu$TPM applications, particularly with respect to the use of RAM memory to store volatile state information. When the $\mu$TSS switches from one process to another, the content of the volatile RAM will be cleared. For instance, if the RAM memory is used to store the PCRs of a TPM, their content will be lost during a process switch. Hence, it is necessary to temporarily store the TPM's volatile state in the embedded NVM (e.g., with the TPM_SaveState command) before another process is activated.

Alternatively the whole process state can always be stored in NVM instead of RAM. This design strategy is applied in the JavaCard architecture, which by default stores objects in persistent memory. However, the architecture also support the creation of so-called transient objects that are stored in RAM memory and reset to a default value at the occurrence of certain events, such as card reset or applet deselection. As traditional smart cards are externally powered, the JavaCard runtime environment supports *atomic transactions* to protect the updates on persistent objects against potential card tear failures. In contrast, updates to transient objects are not atomic and hence not affected by transactions.

## 5.2.2 Process Management

The $\mu$TPM architecture supports multiple execution contexts, which we call *processes*. A process typically corresponds with the implementation of a TPM, but, as argued before, it could also be another (arbitrary) security task. The scheduling of the processes is done outside the $\mu$TPM: the $\mu$TPM Software Stack ($\mu$TSS) indicates which process the $\mu$TPM should run. To keep the scheduling simple we assume that process switches are not allowed while the TPM is executing a command; this implies that TPM commands are atomic. Therefore the $\mu$TPM may block operation temporarily if one process is executing a computationally heavy task such as RSA key generation, or block completely if the code running in it is blocking (e.g., infinite loop). However, those blocking events are either rare or suggest that something has gone massively wrong, whereas the ability to switch processes at arbitrary times would lead to greatly increased complexity.

### Process Description

For every process the $\mu$TPM maintains a data structure $\mathcal{P}_i$, that contains the necessary information to manage the particular process. In the literature this data structure is typically called a *process control block* or a *process descriptor*. At least the following basic information must be stored in the process control block of a $\mu$TPM process:

- The *process identifier* PID uniquely identifies the process. It is used by the $\mu$TSS to indicate which process must be activated in the execution environment. This is very similar to the Application Identifier (AID)[5] of smart card applications.

[5] The AID is defined to be a sequence of bytes between 5 and 16 bytes in length. The first 5 bytes of the AID form the Registered application provider Identifier (RID), which is issued by the ISO/IEC 7816-5 registration authority. The following bytes are the Proprietary Application Identifier Extension (PIX) which enables the application provider to differentiate between the different applications offered.

- A *firmware configuration* $\mathcal{F}$ is associated with every process. This value represents the integrity measurement of the program code (i.e., the firmware) that the process will execute. It is recorded in the FCR immediately after the creation of the process and it can later be reported with an interactive remote attestation protocol (see Section 5.2.5).

- Each process has a *firmware authentication key* $k_{\text{auth}}$. Depending on the scheme that is used to authenticate the disembedded firmware (see Section 5.2.4), this is either a symmetric or a public key. A symmetric key will be used to verify a MAC on the externalized firmware, whereas a public key is used to verify a digital signature.

- The process descriptor also includes information about the *non-volatile data space* that stores the persistent state $\mathcal{T}$ of the process. For simplicity reasons we assume that each process has exclusive access to a *contiguous* chunk of non-volatile memory (see below) and that this data space is allocated statically at process creation. This signifies that the *base address* $b$ and *size* $s$ of the memory chunk are included in $\mathcal{P}$. Note that for each process, $\mathcal{P}_i$ is stored in the embedded NVM together with the non-volatile state $\mathcal{T}_i$ (see Figure 5.1).

- The $\mu$TPM also keeps track of the current *process state*. The process lifecycle and the different process state transitions are explained below.

Summarized, the process control block is represented with a tuple

$$\mathcal{P} = \{\mathsf{PID}, \mathcal{F}, k_{\text{auth}}, b, s, \texttt{state}\} .$$

### Process Lifecycle

The lifecycle of $\mu$TPM processes is illustrated in Figure 5.2. A normal process will typically undergo the following steps:

1. The process is `created` when the $\mu$TSS issues the CreateProcess command on the XIO interface. During this operation non-volatile memory is allocated for the descriptor $\mathcal{P}$ and the non-volatile data space $\mathcal{T}$. In this process state, the firmware configuration $\mathcal{F}$ is still uninitialized, but the other values in $\mathcal{P}$ are already valid. If the firmware is authenticated with a symmetric MAC algorithm, the secret key $k_{\text{auth}}$ will randomly generated during the process creation. If the firmware authentication is based on a digital signature, the public key $k_{\text{auth}}$ will be associated with the process in the next process state.

2. After the creation of a process, the process goes into a `measuring` state, in which only Authenticate commands can be issued. The Authenticate command simultaneously measures and authenticates the program code (i.e., firmware) that will later be executed by the process. More concretely, the $\mu$TSS will load a code collection over the XIO interface. The FTE will "measure" the integrity of this code collection and update the firmware configuration $\mathcal{F}$ of the process. Finally, the FTE will compute a MAC on the code collection and return this MAC to the $\mu$TSS.

3. Next, the process can be `selected` with the SelectProcess command.

4. Once the process has been selected, it can finally run program code with the Execute command. In the `executing` state the $\mu$TPM will only execute code that has previously been authenticated during the measuring state of the process. In other words, it will only run code collections that are authenticated with $k_{\text{auth}}$. The process can go back in the `measuring` state to authenticate additional commands, but this feature is optional.

5. As explained in Section 5.2.1, for simplicity reasons the scheduling of processes is performed outside the $\mu$TPM, by the $\mu$TSS. Before another process can run, the currently selected process must be suspended and put into the `deselected` state with the DeselectProcess command. Afterwards, the process can be resumed by reselecting it and reloading its firmware.

6. Finally, the process ends its existence when it is destroyed with the DeleteProcess command. In Figure 5.2 we include a `deleted` state, but in practice this is a dummy state since the deletion of a process frees all its resources.

Remark that in Figure 5.2 the executing state is considered to be blocking. Consequently the process is stuck into this state until the program code is finished or until the $\mu$TPM and/or the platform are reset. However, this is not a strict requirement and the architecture could support a mechanism to interrupt a blocked process.

### Interface for Process Management

The $\mu$TPM processes can be managed with the following commands over the XIO interface:

- CreateProcess(PID, s) creates a new execution context with process identifier PID and a non-volatile data space of size $s$.[6] If the process

[6] This corresponds with the "non-volatile data space limit" parameter of the GlobalPlatform install command. In the $\mu$TPM architecture there is no equivalent for the "non-volatile code

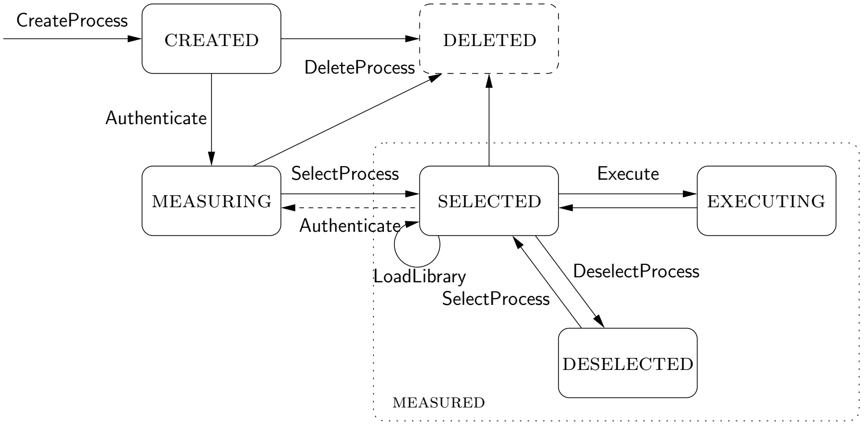


Figure 5.2: Lifecycle of an $\mu$TPM process.

number is no longer available, an error is returned. The $\mu$TSS can decide to delete the process in order to reclaim the desired PID. If the desired amount of non-volatile memory cannot be allocated, an error is returned as well; optionally this error code can indicate the amount of memory that is available.

- SelectProcess(PID) switches context to the process with identifier PID. Once a process is selected, only program code that has been authenticated with the correct firmware authentication key $k_{\text{auth}}$ will be accepted on the XIO interface.
- DeselectProcess temporarily stops the currently selected process. This involves clearing its volatile memory. The DeselectProcess command must be issued before another process can be selected.[7]
- DeleteProcess permanently stops the currently selected process. This command clears the volatile and non-volatile memory of the process and deletes the associated process descriptor $\mathcal{P}$.[8]
- ListProcesses outputs the list of all processes. For each process, the tuple $\{\mathsf{PID}, \mathcal{F}, s, \texttt{state}\}$ with PID the process identifier, $\mathcal{F}$ the firmware

space limit". As the firmware is stored outside the $\mu$TPM, its size can be (theoretically) unlimited.

[7] The DeselectProcess command could also be made implicit: the selection of a different process automatically deselects the currently selected process. The JavaCard architecture works in this manner.

[8] The deletion of a process requires two commands, namely SelectProcess(PID) followed by DeleteProcess. This behavior could be simplified by including the process identifier directly as argument: DeleteProcess(PID).

configuration, $s$ the size of the non-volatile data space and `state` the process state. This command allows the user to identify unwanted $\mu$TPM usage or processes that use too much memory.

The commands that deal with firmware measurement, execution and reporting will be described in Section 5.2.4 and 5.2.5.

### 5.2.3 Memory Management

In order to minimize the hardware resources of the $\mu$TPM, the size of the RAM and NVM memory should be kept small. For this reason, the volatile RAM will only contain the code and data of the currently selected process. While the process is selected, it will have exclusive access to the RAM memory and, when it is deselected, the complete RAM content will be cleared by the $\mu$TPM. Consequently the information that is stored in RAM will not leak during a process switch.

We assumed that the $\mu$TPM has sufficient internal NVM. This requirement can be relaxed by partly externalizing the NVM with the techniques described in Chapter 4. As indicated in Figure 5.1 the FTE ensures that each process has access to its own dedicated, contiguous non-volatile data space, but not to the memory of other processes.

If desired, a mechanism to share NVM between different processes could be provided. This shared memory could then be used for Inter Process Communication (IPC). The JavaCard architecture supports this feature with the shareable interface object mechanism. However, such functionality would greatly increase the complexity of the $\mu$TPM architecture. Moreover, processes can always establish a secure communication channel through the $\mu$TSS.

#### Non-Volatile Memory

To ensure strict process isolation and memory protection, we propose to implement rudimentary *virtual memory management* support. The process will access its non-volatile data space with *logical* addresses and the FTE will map these logical addresses to *physical* NVM addresses. Figure 5.3 provides an example of this mapping mechanism.

In order to keep the memory management as simple as possible, we assume that the allocation of the non-volatile data space is static and contiguous. For each process, the FTE maintains the base address $b$ and the size $s$ of its non-volatile data space. These parameters are stored in the process descriptor $\mathcal{P}$ and they

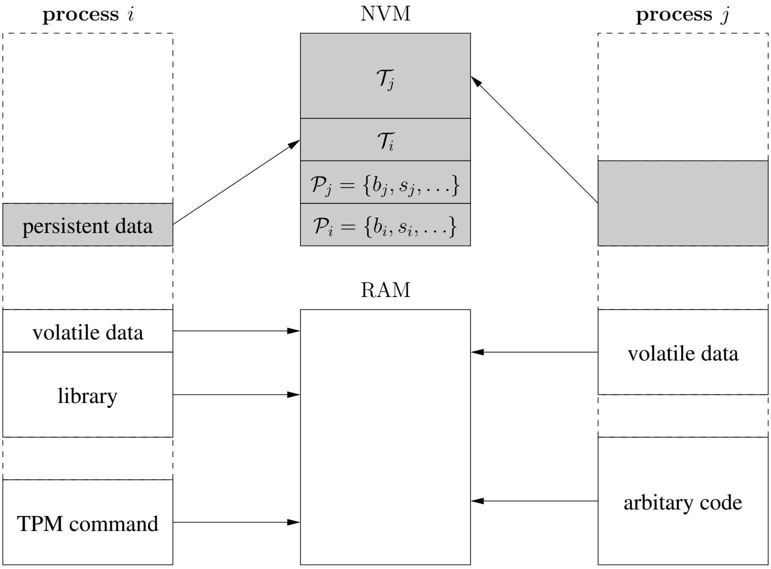


Figure 5.3: Mapping of process memory to physical NVM and RAM.

remain the same during the lifetime of the process. The size $s$ is specified as a parameter of the CreateProcess command and the address $b$ is determined by searching the first memory chunk that is free and big enough to store $\mathcal{T}$.

In Figure 5.3 we assume that the process has a unified memory address space: low addresses (starting from address 0) are mapped to the physical RAM and high addresses (starting from address $h$) to physical NVM. When process $i$ wants to access the high logical address $h + a$, the address gets mapped to the physical NVM address $b_i + a$ on the condition that $a < s_i$. For process $j$ the same memory address is translated to the physical address $b_j + a$ iff $a < s_j$.

The memory address translation is easy to implement and very fast. There are two disadvantages of this simple scheme. Since the non-volatile data space is allocated during process creation, it cannot grow dynamically at runtime. However, this is likely not a big restriction. For instance, the JavaCard language supports dynamic object creation and garbage collection, but the usage of these features is discouraged.[9] Developers typically have a clear idea of how much memory their applet will use. The biggest downside is that the scheme suffers

[9] JavaCard programming guidelines always advise to allocate all objects in the install routine of the applet. Furthermore, when an applet is installed using the GlobalPlatform framework, the “non-volatile data space limit” can be specified as parameter.

from memory fragmentation, because it might not be possible to reuse the non-volatile data space of the deleted processes.

Moreover, a practical realization of the $\mu$TPM architecture will presumably externalize the NVM. This means that the non-volatile data $\mathcal{T}$ of the process will be loaded in the RAM of the $\mu$TPM together with its firmware at runtime. In this case, the memory fragmentation issue does not occur.

### Volatile Memory

As explained earlier, a process has exclusive access to the RAM memory while it is selected, and the $\mu$TPM clears the complete RAM content during a process switch. Consequently, the volatile memory is directly accessible by the process without intervention of the FTE and hence no address translation must be performed. The application developer is personally responsible for the management of the application's volatile memory.

A common usage scenario will be to divide the volatile memory of a process into a static and a dynamic part. The *static* part will contain code and data that is shared between different code collections of a particular process. If data has to be passed from one code collection to another (e.g., an intermediate result in complex calculation or the PCRs of a TPM), the data could be temporarily stored in the $\mu$TPM's non-volatile memory. However, it is much faster to retain the data in RAM memory. Another example is the storage of a shared library in volatile memory, which is used by multiple code collections (see below). The *dynamic* part, on the other hand, is used as scratchpad memory and to store the program code that is being executed.

Figure 5.3 shows a possible memory layout for two processes. The dynamic part of the volatile process memory is located at the bottom of the physical RAM memory and the static part at the top. In the case of process $i$, which represents a TPM implementation, the code of the TPM command gets loaded at address 0, whereas the shared library is loaded at a certain offset. Consequently, when a new TPM command is loaded, the program code of the previous command is overwritten, but the shared library and the volatile state information are untouched.

### Library Support

To minimize the implementation size of each command, the $\mu$TPM could include a library for standard cryptographic operations. In addition, it is possible to load specialized libraries into the static volatile memory that allow for optimizing the size of the individual code collections. Those libraries are uploaded after a process is selected and stay in memory until the next process switch.

In Section 5.2.4 we will describe in detail how program code and libraries get loaded in the $\mu$TPM.

### Process Switch

During a process switch, the physical RAM memory is automatically cleared and overwritten. This guarantees that no secrets leak between processes. The shared library code and the dynamic program code need to be reloaded externally by the $\mu$TSS. However, in some cases the process might need to temporarily store its volatile data to the non-volatile memory before it is suspended, and restore this data afterwards on resumption. This can be done in different ways.

- The process itself is responsible for the storage of sensitive state information. The current TCG specification has commands to suspend and resume the TPM, which are normally used for power saving of the platform. The TPM_SaveState command stores the volatile state, which include the content of PCRs, in NVM and TPM_Startup restores this information. The $\mu$TSS can use these commands to save the volatile data before deselecting the TPM process and to restore it after reselection of the TPM process.
- Each process has an exit and entry routine. The exit routine is executed before the $\mu$TPM switches to a new process, and the entry routine is executed immediately after the process is selected. The exit and entry routine are registered with the FTE during the process creation. This concept is similar to JavaCard applets, which have a select and deselect method.
- The memory management unit of the FTE implements *swapping*. This however adds extra complexity to the $\mu$TPM and hence contradicts the principle of a minimal root of trust.

We prefer the first approach as this requires minimal support by the $\mu$TPM.

The application developer must be aware that the content of the RAM memory can be lost unexpectedly. In fact, the same restriction also exists when developing a smart card application: the power to a smart card can suddenly be interrupted by a so-called *card tear*, i.e., by someone removing the card from the reader.

### 5.2.4 Firmware Integrity Measurement

In Section 5.2.2 we explained that the process goes into a `measuring` state after its creation. In this state the $\mu$TSS will provide the $\mu$TPM with the disembedded firmware such that the integrity of the firmware can be measured and recorded in the FCR of the process (i.e., the value $\mathcal{F}$ in its process control block $\mathcal{P}$).

The FCRs of the $\mu$TPM fulfill a similar role as the PCRs of a TPM. A FCR will store the integrity measurement of the program code that executes in a $\mu$TPM, whereas a PCR contains the integrity measurements of certain software components of a generic computing platform. The content of both configuration registers can be reported with a remote attestation protocol. It should be noted that the number of PCRs of a TPM is fixed (e.g., 16 for TPM 1.1b and 24 for TPM 1.2). Therefore multiple integrity measurements are extended in the same PCR register and a measurement log is needed to determine how the final PCR value was computed. In contrast, the number of FCRs is dynamic, since each $\mu$TPM process has its own dedicated FCR.

A core design principle of the $\mu$TPM architecture is that the program code of an application cannot be stored in the RAM as a whole, since the RAM size is limited. Consequently only a single command or a small collection of commands is loaded at once in the execution environment. We represent the firmware of a $\mu$TPM application as a list of executable commands $c_i$:

$$\{c_0, c_1, c_2, \ldots, c_n\} .$$

The $\mu$TPM architecture differs from the X$\mu$P proposal [57] that also externalizes the firmware: the X$\mu$P only executes code authorized by the device manufacturer, whereas the $\mu$TPM can run arbitrary code but in such a way that executed code can be reported to an external verifier. We propose two schemes for measured execution of program code. The first resembles binary attestation and the second realizes a form of property-based attestation.

#### Binary Measurement

The first option is to use the hash of the complete firmware as the firmware configuration $\mathcal{F}$ of the process:

$$\mathcal{F} = \mathcal{H}(c_0, c_1, \ldots, c_n) .$$

It is important to note that the order in which the commands are measured, determines the resulting hash.

Alternatively, the binary measurement could be computed like the PCR extension operation of a TPM:

$$\mathcal{F} = \mathcal{H}(\mathcal{H}(\mathcal{H}(0, c_0), \ldots), c_n)\,.$$

While the process is in the `measuring` state, the $\mu$TSS provides the $\mu$TPM with the whole firmware image by sequentially loading the code fragments $c_i$. Each time an additional command is loaded, the FTE updates the FCR. Simultaneously the FTE also returns a MAC on the code fragment. The MAC is computed with the firmware authentication key $k_{\mathrm{auth}}$, that is randomly generated during the process creation and that is stored in the process control block $\mathcal{P}$. By measuring the complete firmware, the $\mu$TSS will end up with a list of MACed executable code fragments:

$$\{\{c_0, \mu_0\}, \{c_1, \mu_1\}, \ldots, \{c_n, \mu_n\}\} \quad \text{with} \quad \mu_i = \mathcal{H}_{k_{\mathrm{auth}}}(c_i)\,.$$

Once the whole firmware image has been recorded in FCR, the process can be selected to run the commands that have just been measured. In the `executing` process state, the $\mu$TPM will only execute commands $c_i$ that have a valid MAC.

Figure 5.2, which describes the $\mu$TPM process lifecycle, indicates that the process can optionally return from a `measured` state to the `measuring` state. This means that at a later stage the firmware configuration $\mathcal{F}$ can be updated to include additional commands. In the case of binary firmware measurement, this feature should not be supported as it can be abused by malware to read out the non-volatile state $\mathcal{T}$ of a process. The $\mu$TPM cannot distinguish between additional firmware from the original application developer and malicious code. This implies that once the process has exited the `measuring` state, it can not go back.

With this measurement technique, the only way to securely modify or extend the program code of a deployed $\mu$TPM application, is to create a new process, measure the latest firmware version in this newly created process and finally delete the existing process. Of course the downside of this approach is that the non-volatile state of the old process is lost, unless a state migration mechanism is implemented.

### Interface for Binary Measurement

In order to implement the binary measurement technique, the $\mu$TPM provides the following commands on the XIO interface:

- Authenticate(c) loads the code collection $c$ in the RAM, updates the FCR of the process and returns a MAC on $c$: $\mu = \mathcal{H}_{k_{\mathrm{auth}}}(c)$.

- Execute(c, $\mu$) loads the program code $c$ and verifies the accompanying MAC $\mu$. If the verification is successful, the code $c$ is executed.
- LoadLibrary(l, $\mu$) loads the library code $l$ and verifies the associated MAC $\mu$. If the verification fails, the library $l$ is erased from the memory. Otherwise, the library is kept in memory such that its functions can be called by the process.

Figure 5.3 illustrates that regular program code and library code are loaded at a different location in the volatile memory space of a process. The Execute command loads program code at the bottom of the memory space, namely at address 0. The LoadLibrary command on the other hand loads program code at a certain *offset*. This offset can be a pre-defined value or it can be specified as a parameter of the LoadLibrary command.

### Property Measurement

A second option is to sign the code of every individual command or of every command collection. The complete firmware can then be represented as the public key $PK$ of the firmware developer and a list of executable code fragments $c_i$ and associated digital signatures $\sigma_i$:

$$\{PK, \{c_0, \sigma_0\}, \{c_1, \sigma_1\}, \ldots, \{c_n, \sigma_n\}\} \quad \text{with} \quad \sigma_i = \mathsf{sign}_{SK}(c_i)\,.$$

The private key $SK$ with which the firmware developer signs the firmware, is the same for all commands $c_i$.

The $\mu$TPM will use the hash value of the corresponding public key $PK$ as the firmware configuration $\mathcal{F}$ of the process:

$$\mathcal{F} = \mathcal{H}(PK)\,.$$

Alternatively, the developer's public key $PK$ can be replaced by a *property certificate*. This certificate will contain the public key $PK$ as well as attributes describing certain properties (e.g., version) of the firmware. The property certificate is issued by a trusted third party, that maps binary measurements to properties. In the literature this approach is commonly referred to as *delegation-based* property attestation [161, 224].

It might occur that the main TPM functionality is produced by one entity (e.g., a current TPM vendor), but extensions are added by a different entity (e.g., a government agency). In this situation the second entity has to sign the extensions as well as the original code with its own signature key.

The verification of digital signatures imposes a considerable computational and communication overhead, which will slow down the loading and unloading of commands in the $\mu$TPM execution environment. This bottleneck can be overcome by verifying the signature on every TPM command once and calculating a MAC for subsequent verifications. Similarly to the binary measurement approach, the random firmware authentication key $k_{\text{auth}}$ will be used to MAC the program code.

While the process is in the `measuring` state, the $\mu$TSS can load the signed firmware in $\mu$TPM to get it MACed. By doing so the $\mu$TSS will end up with a list of authenticated executables:

$$\{\{c_0, \mu_0\}, \{c_1, \mu_1\}, \ldots, \{c_n, \mu_n\}\} \quad \text{with} \quad \mu_i = \mathcal{H}_{k_{\text{auth}}}(c_i)\,.$$

The big difference between binary and property measurement is that the FCR of the process is not modified while the firmware is authenticated. There is an initial step that associates the public key *PK* with the process and this will determine the firmware configuration $\mathcal{F}$. Once this association step is done, the $\mu$TPM can either execute signed code, transform signed code into MACed code, or execute MACed code.

With property measurement a process can always return to the `measuring` process state in order to authenticate additional commands, whereas binary measurement does not support this feature. Hence, it is possible to add or replace commands without changing the firmware configuration $\mathcal{F}$, provided that the application developer signs the new commands with the same private key *SK*. This needs to be done carefully though, as mixing different versions of commands might lead to security problems. It is thus recommended to create a new process with a different firmware configuration (i.e., different *SK* and *PK*) if major updates are made, and authenticate the updated firmware in the newly created process.

In the case of a TPM implementation, changing the firmware configuration is equivalent with the creation of a new logical TPM. Consequently, the existing keys of the old TPM will be lost, unless they are migrated with the migration or maintenance functionality.[10] It would be possible to build in some versioning support as well, but this would add unnecessary complexity to the $\mu$TPM architecture.

[10] This complies with the TPM specification, which states that "*When a field upgrade occurs, it is always sufficient to put the TPM into the same state as a successfully executed* TPM_RevokeTrust*.*"

### Interface for Property Measurement

For the signature-based measurement approach there are two implementation options: (1) the $\mu$TPM can always work with signed program code or (2) it can verify the signed firmware once in the `measuring` process state and subsequently operate on MACed program code.

For the first option, the following $\mu$TPM commands need to be supported:

- AssociateKey($PK$) loads the public key $PK$ in volatile memory and stores $\mathcal{F} = \mathcal{H}(PK)$ as firmware configuration in the process descriptor.
- Execute(c, $\sigma$) loads the program code $c$ and verifies its signature $\sigma$. If the verification is successful, the code $c$ is executed.
- LoadLibrary(l, $\sigma$) loads the library code $l$ and verifies the associated signature $\sigma$. If the verification fails, the library $l$ is erased from the memory. Otherwise, the library is kept in memory such that its functions can be called by the process.

The AssociateKey command has to be invoked when the process is in the `measuring` state, directly after the process creation. The association of the key $PK$ to the process can happen only once. Otherwise it is possible for an adversary to program its own public key and run malicious firmware signed with this key to read or modify the state $\mathcal{T}$ of the target application.

If the $\mu$TSS wants to associate a different key with the process, the existing process must be stopped and a new process must be created, with DeleteProcess and CreateProcess respectively.

For the second option, the following $\mu$TPM commands have to be supported:

- AssociateKey($PK$) loads the public key $PK$ in volatile memory and stores $\mathcal{F} = \mathcal{H}(PK)$ as firmware configuration in the process descriptor.
- Authenticate(c, $\sigma$) loads the program code $c$ in the RAM and verifies its signature $\sigma$. If the verification is successful, the $\mu$TPM returns a MAC on $c$: $\mu = \mathcal{H}_{k_{\mathrm{auth}}}(c)$.
- Execute(c, $\mu$) loads the program code $c$ and verifies its MAC $\mu$. If the verification is successful, the code $c$ is executed.
- LoadLibrary(l, $\mu$) loads the library code $l$ and verifies its MAC $\mu$. If the verification fails, the library $l$ is erased from the memory. Otherwise, the library is kept in memory such that its functions can be called by the process.

With this implementation option, it is possible to authenticate additional code securely, as long as the application developer's key $PK$ remains the same.

### 5.2.5 Firmware Integrity Reporting

The FTE guarantees that a process only executes authenticated code and that processes run isolated from each other. Therefore, the secrets stored by an application (e.g., SRK of a TPM, monotonic counters, owner credential, etc.) stay confidential. The TPM process can generate its own EK, but a mechanism is desired to link this key to the $\mu$TPM hardware and the firmware configuration $\mathcal{F}$. Otherwise remote parties cannot distinguish between a TPM process executing in a $\mu$TPM processor and a software emulator on a PC [252].

For this reason, every $\mu$TPM ships with an asymmetric key pair called HEK that uniquely identifies the device. The private part of the HEK, which we denote $SK_{\text{dev}}$, is stored in the $\mu$TPM's on-chip OTP NVM and never leaves the device. The public part of the HEK is signed by $\mu$TPM producer during manufacturing, yielding the hardware endorsement certificate. The HEK certificate can be stored outside the $\mu$TPM.

The FTE provides the following attestation routines:

- FTE_FCRRead() returns the firmware configuration $\mathcal{F}$, which is stored in the FCR.
- FTE_Quote(b) uses the HEK to create a signature on the blob $b$ and the FCR content:
$$q = \mathsf{sign}_{SK_{\text{dev}}}(b, \mathcal{F})\,.$$

It is important to note that the attestation feature is not exposed externally over the XIO interface, but only internally to $\mu$TPM processes.

With this functionality, it is possible to bind the EK of a TPM process to the HEK of the underlying $\mu$TPM process.

The TPM process will generate its own endorsement certificate by performing the FTE_Quote operation on the public EK: $\texttt{cert}_{EK} = \mathsf{sign}_{SK_{\text{dev}}}(EK, \mathcal{F})$. This endorsement certificate will be provided later to a privacy CA when the TPM registers an AIK certificate (see Section 2.1.2). The privacy CA should include the firmware configuration $\mathcal{F}$ as an attribute in the AIK certificates.

## 5.3 Discussion

### 5.3.1 Implementation Options

Figure 5.1 gave a conceptual representation of the $\mu$TPM architecture. A number of options exists to realize the proposed architecture in practice. The $\mu$TPM can be implemented as a discrete chip or embedded in an existing platform component, as is the case with a traditional TPM or an SE. Alternatively, the functionality can be realized, like a MTM, as a software component that runs logically isolated from the rest of the platform. As illustrated in Section 4.1 the isolation between a legacy and a trusted execution environment can be achieved with hardware extensions such as ARM TrustZone or TI M-Shield or with a security kernel.

#### Hardware Requirements

The $\mu$TPM requires similar hardware components as a regular TPM or a smart card: a secure microcontroller possibly assisted by a cryptographic coprocessor, volatile RAM memory and MTP non-volatile memory. The cryptographic primitives that must at least be supported, are a MAC algorithm to authenticate the disembedded firmware and a public key signature scheme (both verification and generation).

The major difference between a conventional TPM and the $\mu$TPM architecture is the lack of ROM. For current TPMs, the ROM contains the firmware that implements the fixed functionality defined by the TCG specifications.[11] The $\mu$TPM architecture conversely is ROM-less and stores its firmware outside its security perimeter. The $\mu$TPM contains an OTP key store for the private hardware endorsement key $SK_{\text{dev}}$. Furthermore the $\mu$TPM also contains reprogrammable NVM to store the process control block $\mathcal{P}_i$ and the persistent state $\mathcal{T}_i$ of the processes that operate in the $\mu$TPM environment. As shown in Chapter 4 this memory can be securely externalized with an adequate state protection scheme.

The communication interface of the $\mu$TPM architecture differs from a traditional TPM. A standard TPM has a single I/O bus to transfer data from and to the rest of the platform. The $\mu$TPM on the other hand has a second interface to manage the processes and load program code. We denoted the interface to transfer data IO[12] and the control interface XIO. In a real implementation both

[11] The TCG specifies an optional TPM_FieldUpgrade command to update the firmware. This implies that the firmware can be stored (partially) in reprogrammable NVM.

[12] Optionally this port supports locality.

(logical) interfaces will use the same physical bus. The TPM's communication interface is ordinarily a low speed bus (e.g., LPC or SMBus). However, when Ekberg and Bugiel implemented a software MTM with disembedded firmware and externalized non-volatile state [90], they found out that the communication interface becomes a bottleneck. Whenever code collections are loaded onto the $\mu$TPM execution environment, the firmware must be transferred over the XIO interface and subsequently its integrity must be verified. Similarly the non-volatile state will be swapped in and out over this communication bus. The bandwidth and latency of the XIO interface could pose a problem, especially if the $\mu$TPM is used to run a security application that requires a fast transaction speed (e.g., NFC payments in a large event).

### Realization of Firmware Trust Engine

The core component of the $\mu$TPM architecture is the FTE, which is responsible for process and memory management and for firmware authentication.

In [164] we originally envisioned the FTE to be implemented as hard-coded logic that is separated from the microcontroller on which the firmware is executed. The FTE will receive requests over the XIO interface, e.g., to load a different code collection in the current execution context or to switch to another process. Based on the request it has to control the microcontroller (e.g., start/stop the processor) and the RAM (e.g., load another code collection into memory or clear the memory during a process switch). Additionally, it also must control access to the NVM such that a process can only access its own persistent state. These requirements imply that the FTE must be tightly integrated in the memory controller and the interrupt controller of the microcontroller.

A more natural way to realize the FTE functionality is with a basic operating system. This operating system will closely resemble a traditional JavaCard runtime environment, especially considering that the $\mu$TPM's multiprocessing and memory management are strongly inspired by the JavaCard architecture. The main differences between the $\mu$TPM architecture and a JavaCard runtime environment is the card context management: the $\mu$TPM offers an open environment where arbitrary applications can be executed, albeit in a measured fashion that can later be cryptographically verified, whereas the GlobalPlatform framework is essentially closed since a third party, such as the TSM, controls which applications can be installed on the runtime environment. A secondary difference is the fact that the applet code and its associated state are stored externally. This implies that the size of the NVM (typically 16-32 kB) of the JavaCard SE no longer forms a restriction.

Note that a software realization mandates that the FTE is stored reliably and that its integrity is protected. This requirement can be met by storing the operating system in on-chip ROM, hence inside the $\mu$TPM security perimeter. Alternatively a secure boot loader, that is stored in on-chip ROM, can be used to retrieve the operating system from an external storage medium. In the latter case, a chain of trust will be created:

1. The boot ROM will load the operating system and check its integrity, either by comparing the hash on the operating system's image with an expected value in ROM or by verifying the digital signature on the image with the manufacturer's public key in ROM. This step applies the secure boot principle.

2. The FTE, which is part of the operating system, will load the disembedded firmware that gets executed in the $\mu$TPM. It will measure the integrity of the firmware and afterwards report the firmware's integrity with a remote attestation protocol. This step applies the measured boot principle.

#### $\mu$TPM Software Stack

In Section 5.2.1 we explained that the $\mu$TPM needs an accompanying software component on the platform, the $\mu$TSS. The most obvious implementation option is to extend a conventional TSS with the functionality to deal with the externalized firmware. The TSS has to store the disembedded firmware and load the appropriate program code over the XIO port before issuing a TPM command over the IO port. The integrity of the TPM's firmware is protected by the authentication key $k_{\mathrm{auth}}$, but its availability cannot be guaranteed. If the external firmware gets deleted (purposely or involuntary), the corresponding $\mu$TPM process becomes unusable. However this is not a big concern as other DoS attacks can be applied on current TPMs (e.g., deleting part of the TSS or the TPM driver), and TPMs were never meant to withstand a DoS attack.

Although not indicated on Figure 5.1, it is also necessary that the platform's CRTM stores the code of the TPM commands that are used during the platform startup (e.g., TPM_Extend). The CRTM might use another non-volatile storage medium, such as the BIOS Flash memory, than the TSS.

### 5.3.2 Memory Externalization

Up until now, we have assumed that the $\mu$TPM has abundant on-chip memory to store the non-volatile state of all processes that run in its trusted execution environment. This may seem strange because we do a lot of effort to disembed the program code from the $\mu$TPM, while the non-volatile data space of processes is still stored inside the $\mu$TPM. The assumption of on-chip storage of the process data was made to simplify the description of the $\mu$TPM architecture. However, we envision that a practical implementation of the $\mu$TPM architecture will externalize the storage of the non-volatile process data as well as the process code.

There are two major challenges with externalized memory. Firstly, the memory management becomes more complex, especially if the internal RAM of the $\mu$TPM is too small to hold the complete non-volatile data space of a single process. Secondly, a scheme is needed to protect the confidentiality, integrity and freshness of the externalized non-volatile storage. In Chapter 4 we developed a variety of solutions for the latter challenge. Briefly summarized, the $\mu$TPM requires a small amount of reprogrammable NVM to store either a replay detection nonce or an updatable symmetric key.

### Multiprocessing Support

As shown in Figure 5.1, the non-volatile memory of the $\mu$TPM contains the process descriptor $\mathcal{P}_i$ and non-volatile state $\mathcal{T}_i$ for every process and the device specific key $SK_{\text{dev}}$. The HEK must be programmed during the manufacturing of the $\mu$TPM and, because it does not change afterwards, it can be stored with OTP fuses. The other non-volatile data can be stored outside the $\mu$TPM's security perimeter, provided that it is protected with a secret key that is only known by the $\mu$TPM. This key, which we called the state protection key in Chapter 4, fulfills a comparable role as the SRK of a conventional TPM. The keys maintained by a TPM are encrypted by the SRK, which forms the root of a key hierarchy, and they are stored outside the TPM, for instance on hard disk. Similarly the state protection key protects the non-volatile memory of processes in external memory.

In Section 4.2.1 of the previous chapter we identified the security requirements for the externalized non-volatile storage. The non-volatile data must be encrypted and authenticated to protect its confidentiality and integrity and the state protection key has to be different for every device such that the externalized state is uniquely bound at a specific $\mu$TPM. Furthermore, the $\mu$TPM needs to make sure that all old versions of the externalized memory become unavailable whenever the memory is updated. This last requirement is important in order to guarantee that the control block $\mathcal{P}$ of a process contains the up-to-date firmware configuration $\mathcal{F}$ or if a process maintains a monotonic counter in its persistent state $\mathcal{T}$. In Chapter 4 we explained that state replay can be detected

(1) by including a nonce in the externalized memory and maintaining a local copy of this nonce inside the $\mu$TPM or (2) by changing the state protection key every time the memory content is updated.

As motivated in Section 5.2.1 multiprocessing support is an essential feature of the proposed $\mu$TPM architecture. The memory externalization scheme must adequately isolate the non-volatile data of the different $\mu$TPM processes. In general, one can choose to protect the persistent data of each process with an individual key $k_{\mathcal{T}_i}$ or to use one global state protection key $k_{\mathcal{T}}$ for all processes. With the latter approach, it is important that the externalized memory of different processes cannot be interchanged. This can for instance be done by checking whether the process identifier PID matches the value in the encrypted process description.

In a practical realization of the $\mu$TPM architecture the amount of embedded MTP NVM will be limited. Therefore it is not possible to store an updateable key $k_{\mathcal{T}_i}$ or a replay detection nonce $n_{\mathcal{T}_i}$ for an unlimited number of processes. Consequently the number of $\mu$TPM processes could be limited in order to not run into this restriction. Alternatively, the keys and/or nonces that protect the non-volatile memory of the different processes, could be externalized as well. This can be done by encrypting the keys $k_{\mathcal{T}_i}$ with one global key $k_{\mathcal{T}}$ or by protecting the integrity of the nonces $n_{\mathcal{T}_i}$ with an authentication tree, whose root nonce $n_{\mathcal{T}}$ is kept inside the $\mu$TPM.

### Memory Management

In Section 5.2.3 we explained that the $\mu$TPM clears its RAM memory during a process switch. The application developer must anticipate this feature by temporarily storing the volatile data of a process to the embedded NVM before the process is suspended. Therefore the process exposes commands to save and restore its volatile data and the $\mu$TSS issues these commands before and after a process switch respectively.

When we take into consideration that the on-chip NVM of the $\mu$TPM is not big enough to simultaneously hold the persistent state of all processes, the memory management must be modified. In Figure 5.3 we described how the memory layout of a process gets mapped to the volatile RAM memory and the persistent NVM memory. When implementing NVM externalization, the strict distinction between the volatile code and data space and the non-volatile data space is no longer necessary. The $\mu$TPM only needs a small amount of NVM for persistent key storage and the rest of its on-chip memory can be volatile RAM. The program code of a process as well as its data are stored in the internal

RAM when the process is selected, and in the external NVM when the process is deselected.

The authenticity and integrity of the program code is protected by the schemes that are discussed in Section 5.2. The confidentiality, integrity and freshness of the process state will be protected by the state protection schemes that we proposed in Section 4.2. Note that we do not consider confidentiality and version management of the firmware as essential requirements. If these properties are desirable, the state protection scheme can also be used to secure the disembedded firmware.

In Section 5.2.3 we explained that rudimentary virtual memory management is needed to map the logical process memory to physical RAM and NVM addresses. This requirement is no longer necessary since the memory of an active process will only reside in the on-chip RAM. The running process will have full access to the RAM memory, except for a dedicated portion of the memory in which the FTE will maintain the process descriptor.

In a simple setting, we can assume that the state of each process fits completely into the embedded RAM of the $\mu$TPM, but that the size of this memory is insufficient to simultaneously hold the data space of all processes. Consequently, the process state must be swapped during a process switch. As an answer to a process creation/switch command, the $\mu$TPM returns the memory content of the current process as an authenticated and encrypted blob, and expects the content of the next process in return. This functionality does not necessarily need to be implemented in the $\mu$TPM hardware itself; the process can provide commands to the $\mu$TSS for the retrieval and cleaning of its memory (e.g., with the TPM_SaveState and TPM_Startup command).

If the memory that is needed to execute a process exceeds the size of the hardware memory in the $\mu$TPM, memory management gets more complicated. In theory, one can design a scheme that simulates page faults as TPM responses to the $\mu$TSS. For instance it is possible for the $\mu$TPM to communicate to the $\mu$TSS during command execution by using error codes; the $\mu$TPM is a passive device, hence error codes are the only mechanism to notify the $\mu$TSS about problems. This would allow us to implement a function that can swap in and out blocks of internal memory manually while commands are being processed. We do, however, feel that such design is an abuse of error codes and the support for this functionality contradicts the principle of a minimal root of trust.

### Memory Availability

It should be noted that externalization of the NVM allows for new forms of DoS attacks. While the externalized firmware can easily be replaced if it got lost, an attacker may be able to remove the externalized NVM for good, and the only recovery from such an attack is the reset of the $\mu$TPM to its default setting. While this attack is not critical in most TPM usage scenarios – to execute it, the attacker already requires a level of control over the host platform that allows for many other ways of DoS attacks – this is an attack that is not possible for a normal TPM, and some TPM-based protocols may assume it impossible.

One solution to this issue is for the $\mu$TPM to collaborate with secure storage, as specified by the TCG Storage Work Group [280]. In this case, the $\mu$TPM only releases its internal memory once it receives an acknowledgement from its counterpart in the hard disk that the memory content has successfully been stored in a hard disk section that is unavailable for the operating system.

## 5.3.3 Security Considerations

Our approach attempts to stick closely to the original TPM attack model. Nevertheless, the exposure of the internal TPM data does create small differences, which in some applications may require extra care to prevent an attack.

### Denial of Service

In our system, the attacker can delete the internal state of the TPM process, and even its firmware. While the program code and data cannot be replaced by something meaningful, such an attack may disable the TPM or force a reset into its native state. This is a stronger attack than possible on a normal TPM, where deletion of external data can only destroy key material stored under the SRK, but no state information or functionality. We would argue though that in practice, there is little difference between an attack on a $\mu$TPM and an attack on a classical one. If a classical TPM looses all key blobs, most crucial state information is lost as well, and an application using this TPM needs to find a way to recover without opening other avenues of attack.

The only setting where a difference may occur is in a virtualized environment. It must not be possible for one virtual environment to destroy state information of the virtual TPM of another one. This means that each virtual TPM must have a local copy of its relevant state information, and shared information must be stored by a trusted program (e.g., the hypervisor), where it cannot be deleted by individual virtual machines.

### Access Analysis

While our $\mu$TPM architecture protects TPM data from being read by an attacker, the attacker may be able to see the encrypted internal state, or – for example, by analyzing the cache of the $\mu$TPM or the untrusted storage – obtain information about the last commands submitted to the $\mu$TPM. While this does not endanger security in most settings in which a TPM is used, this increased visibility should be taken into account, and critical applications may need to apply additional measures to hide the activity of the $\mu$TPM, for example by adding fake encrypted state blocks and clearing all caches after usage.

### Vendor Backdoor

One of the motivations for a more flexible, alternative TPM architecture was verifiability of the TPM firmware. By storing the firmware outside the $\mu$TPM, it can be publicly inspected by the community and therefore users can get more assurance that the TPM functionality is implemented correctly and that it does not contain any backdoor functionality. This implies that the firmware developer makes the source code of the TPM implementation available as well as the tools to compile the code to the binary firmware image.

However, in the case of signature-based property measurement, it is still possible for the vendor to authorize TPM commands that are invisible to the user (i.e., not supplied with the original $\mu$TPM on delivery), but used later as a backdoor to violate the security of the TPM. Consequently, in security settings where the firmware developer is untrusted, the signature-based scheme must not be used, or the user should sign the firmware with his own public key and thus lock the original firmware developer out of the TPM.

## 5.4 Conclusion

We have shown that many of the current issues with TPM implementations that stem from complexity and inflexibility can be overcome by redefining the trust boundaries. By putting the firmware outside of the secure hardware and securing it cryptographically, our architecture allows for simplified hardware, while gaining flexibility in the supported command set and even allowing multiple secure coprocessors to share the same hardware. Our architecture is largely compatible with the current specification, provided an additional software layer is added between the classical TSS and the $\mu$TPM; thus allowing for the improved architecture without having to adapt the TCG specification.

More concretely, we introduced a novel flexible architecture for a secure coprocessor, which we called $\mu$TPM. The $\mu$TPM architecture combines a number of techniques that have been proposed in the literature. The firmware that is executed on the $\mu$TPM, is stored externally, like in the X$\mu$P proposal of Chevallier-Mames et al. [57]. However, whereas the X$\mu$P architecture only runs code that is signed by the manufacturer (or by a trusted third party), our architecture applies the trusted computing principle of executing arbitrary code, but in a (remotely) verifiable way. We proposed a scheme for binary measurement of the firmware integrity and one for delegation-based property measurement. The $\mu$TPM proposal supports multiprocessing in a similar way as the JavaCard architecture.

# Chapter 6

# Reconfigurable Trusted Computing

The implementation of a trusted module on reconfigurable hardware is beneficial because it allows for updates of the firmware (like in Chapter 5) as well as updates of the hardware of the trusted module (e.g., the cryptographic coprocessor) after deployment in the field. However, SRAM-based FPGAs, which form the majority of the FPGAs sold today, store their configuration bitstream in external NVM in order to persist across power cycles. This requirement makes the implementation of a trusted module on SRAM-based FPGAs challenging, especially with respect to the storage of the persistent state.

In this chapter we discuss how the techniques from Chapter 4 can be used to protect the persistent state of a trusted module on currently available FPGAs. This research was published, in part, in [228]. We also describe a novel FPGA architecture that defines a root of trust to measure and report the integrity of partial bitstreams. This scheme, which we presented in [87], goes a step further than the $\mu$TPM architecture of Chapter 5 as it allows to attest not only the integrity of the TPM's firmware, but also the integrity of the configuration bitstream that represents the TPM's hardware.

The research in [87] was started by the co-authors from the Ruhr-Universität Bochum and the author of this thesis contributed at a later stage by helping to refine and improve the architecture.

## 6.1 FPGA Security

In this chapter we focus on security aspects of FPGAs. The bulk of these devices are SRAM-based FPGAs. The configuration program/bitstream that defines their functionality is stored in non-volatile memory and loaded onto the chip at start-up. Due to their flexibility and low cost, they are very attractive for prototyping and developing new innovative applications. Although their flexibility is one of their main advantages, it also poses one of the main threats to the security of these devices. In this section we survey various protection technologies for volatile SRAM-based FPGAs. For an extensive overview on this topic we refer the reader to the survey paper of Wollinger et al. [305] and the thesis of Drimer [79].

### 6.1.1 Attacker Objectives

The value of the products and applications in which FPGAs are used, resides mainly in their functionality which is embedded in the bitstream. Therefore, development houses like to protect their bitstream, as its functionality represents a significant development investment or because it has certain security requirements. Broadly speaking, an attacker can have two main objectives.

1. **Stealing intellectual property.** The attacker may have commercial interests to steal the *intellectual property* (IP) contained in the bitstream. This allows him to create a competing product derived from the original at a reduced cost, either by making a one-to-one copy (i.e., *cloning*) or by *reverse engineering* the functionality and reusing it in his own product. In addition, the third party facility that manufactures or assembles the product, may produce more devices than contractually agreed. This is known as *overbuilding*. Observe that this last attack is usually very easy to carry out from a technical point since the manufacturing facility usually has all the information required to manufacture the products.

2. **Obtaining security sensitive information.** The adversary may want to attack certain security functionality present in the bitstream. A common threat is the extraction of a secret cryptographic key, that is, either a symmetric key or the private key of an asymmetric key pair. In other applications, the attacker may try to bypass an access control mechanism or a license check. Another possible attack is to reverse engineer an obfuscated cipher. Although not an FPGA application, the reverse engineering of the Keeloq algorithm [88], the MIFARE Classic cipher [100], and the Hitag2

cipher [293], which led to the complete collapse of the system's security, illustrates the power and consequences of this type of attacks.

## 6.1.2 Attacks

Depending on the approach that the attacker takes, his objectives, and budget we can identify the following attack strategies:

### Bitstream Copying

Very often the configuration bitstream of an SRAM-based FPGA is stored unprotected in external non-volatile memory. It is fairly easy for a competent attacker to tap the bus over which the bitstream is loaded onto the FPGA (see Chapter 3). As FPGAs are generic devices, the recorded bitstream can be used to program any other FPGA of the same type and size. The attacker can regard the FPGA as a black box and only needs to invest effort in copying the printed circuit board on which the FPGA is mounted. This can be done with reasonable effort and cost, since these boards consist usually of standard components. Clearly, such a cloning attack is a serious vulnerability of today's volatile FPGAs.

### Bitstream Readback

Readback is a debugging feature provided by some FPGA families. It allows to retrieve the FPGA's internal configuration memory, after start-up and while in operation. The snapshot returned by a readback operation includes the FPGA configuration and the contents of the Look-Up Tables (LUTs) and memory of the FPGA. It differs slightly from the original bitstream. In particular, some padding and header/footer information is missing from the readback version. If an attacker can add this missing information, he has a copy of the bitstream and he can clone the device.

In addition, the attacker can perform a *readback difference attack* to bypass an IP protection or security mechanism [79]. He can for instance take multiple snapshots to determine which internal signal activates a protected IP core. The readback functionality also has security implications, as it can be used to read secret information from the FPGA's built-in RAM memory. In this context, we can mention that experiments on personal computers have shown that it is easy to identify cryptographic keys in memory because they have high entropy [122, 239]. On the other hand, internal readback can also be used as a protection mechanism [278]. The design can verify a checksum or cryptographic hash on its own configuration in order to detect tampering.[1]

Most FPGA manufacturers provide a mechanism to prevent readback. Xilinx uses security bits inside the bitstream to disable external and/or internal readback. However, these bits can be easily found and overwritten in the external configuration memory. If the bitstream is stored in integrated non-volatile memory though, these security bits cannot be modified. When bitstream encryption (see below) is used on Lattice and Xilinx FPGAs, readback is automatically disallowed; otherwise it could be used to get the decrypted bitstream. Finally, Altera does not offer any readback capability and consequently is not susceptible to this type of attacks.

### Reverse Engineering of Bitstream

The attacks described thus far provide the attacker with access to the FPGA's bitstream. Next, he can try to transform the encoded bitstream into a logical functional description, expressed as a netlist or in a Hardware Description Language (HDL). This reverse engineering process is defined as *full bitstream reversal* by Drimer [79]. In practice, the encoding of bitstream formats is largely undocumented and obscure. In the 1990s NeoCAD reverse engineered Xilinx's bitstream generation software to generate compatible bitstreams, and Clear Logic was able to program laser configured ASICs based on existing Altera bitstreams. Since then, FPGAs have become more sophisticated and no successful reverse engineering attacks have been reported. This suggests that, given the size and complexity of modern FPGAs, full reversal of (large) bitstreams is currently very difficult, time consuming, and (most probably) not economically viable. Notice however that reverse engineering difficulty does not correspond to "difficult" in the cryptographic sense (e.g., exponentially small in some security parameter) but rather to a very large engineering effort.

*Partial bitstream reversal*, which is defined as decoding the look-up tables and initial RAM content from bitstreams, is much easier. The Debit project[2] of Note and Rannaud [203] aimed at full netlist recovery from closed FPGA bitstream formats, but it was only able to decode LUTs and RAM contents from Xilinx bitstreams before the project was discontinued in 2008. In August 2012 Benz et al. [23] presented a tool called Bitfile Interpretation Library (BIL)[3] that improves upon the Debit work and that can reverse certain sections of a Virtex-5

[1] The interface that is used for internal readback, can often also be used for dynamic reconfiguration. If the FPGA supports this functionality, a part of the design can be decrypted and loaded using the internal reconfiguration interface [318].

[2] `http://code.google.com/p/debit/`

[3] `http://florianbenz.github.com/bil/`

bitstream. However, the authors conclude that full bitstream reversal remains infeasible for the time being.

Given the existence of tools for partial bitstream reversal, hiding cryptographic keys inside the bitstream (as LUTs or RAM content) should be avoided. Similarly, executable code of a soft microprocessor can be extracted from the bitstream. The knowledge gained from a partial bitstream reversal can be used to make targeted modifications to the bitstream; e.g., to replace program code, a root public key or a substitution box of a block cipher [149].

### Side-Channel Attacks

*Side-channel attacks* exploit the fact that a physical implementation of a cryptographic algorithm leaks unwanted information while processing secret data (see also Section 3.4). The physical leakage, including timing [154], power consumption [155] and EM radiation [2, 219], can be measured externally and used to recover the secret keys with statistical methods. In recent years, various side-channel analysis attacks have been demonstrated on FPGA implementations of cryptographic algorithms and protocols [41, 43, 67, 68, 204, 211, 253, 254, 255, 256, 257, 258].

A lot of research has been done to strengthen hardware implementations against these attacks. For example, the execution time of the algorithm has to be made data independent to prevent timing analysis. For most cryptographic algorithms this is relatively easy. However, power and EM analysis are much harder to prevent and the development of effective countermeasures is still ongoing research. In general, the approaches can be divided into software and hardware countermeasures. Software-based countermeasures adapt the algorithm such that the occurrence of predictable intermediate results is avoided. Typically, the data representation of secret information is masked with random values. These algorithmic countermeasures can be easily applied to FPGA implementations. Hardware-based countermeasures include noise generation (to decrease the signal-to-noise ratio of the measurements) and special logic styles that make the power consumption of the circuit constant. Only some logic level changes can be implemented on FPGAs [274, 275, 315, 316]. Cryptographic implementations can also be attacked by the injection of faults (e.g., glitch in power or clock supply [6], EM radiation) during the computation. Such *fault attacks* can be applied to FPGAs.

### Physical Invasive Attacks

Finally, an attacker can remove the package of the FPGA and attack the chip physically. He can attempt for instance to read data on internal buses with microprobes, inspect the fuses that store the bitstream encryption key with a microscope, or (re-)enable readback with a fault injection. Given the feature size[4] and the complexity of state-of-the-art FPGAs, the cost of such physical attack will be very high. Currently, no reports are available on a successful (semi-)invasive attack against volatile FPGAs.

## 6.1.3 Defenses

As illustrated with the attacks previously described, the configuration bitstream is the most common and possibly, the simplest attack target. An adversary can copy it to make a clone, analyze it to learn secret information (e.g., a key or a proprietary algorithm) or alter it to modify the functionality of the design. FPGA vendors acknowledge this threat and have extended some of their devices with defense mechanisms. Firstly, the bitstream can be protected by storing it externally in an encrypted form or by storing it in internal non-volatile memory that is not accessible from the outside. Finally, *node locking* prevents cloning and overbuilding by binding the bitstream (and its embedded IP) to a unique FPGA.

The theft of intellectual property can also be retroactively detected with bitstream *watermarking* and *fingerprinting* [45, 140, 143, 144, 145, 166, 167, 168, 169, 170, 317, 321, 322]. These countermeasures do not actively prevent the theft, but provide digital evidence in a court case against a fraudster. Some of the proposed schemes are not robust and can be circumvented after a partial bitstream reversal, as shown by Van Le and Desmedt in [291].

### Bitstream Encryption

*Bitstream encryption* is a mechanism that provides confidentiality of the bitstream while binding it at the same time to the platform. The bitstream is encrypted with a user-defined key that is stored in on-chip non-volatile memory on the FPGA. When the encrypted bitstream is loaded onto the FPGA, it is first fed to a decryption module. The decrypted bitstream is then used to configure the FPGA. If the attacker eavesdrops on the communication bus to the external non-volatile memory, he obtains an encrypted bitstream, which he

[4] For instance, the Xilinx Virtex-7 family is produced in 28 nm CMOS technology.

cannot reverse engineer nor use in another device (because he does not know the correct decryption key).

If the cleartext bitstream contains redundancy checks, tampering of the encrypted bitstream will be detected by the FPGA to some extent. Academic researchers have proposed to explicitly protect the integrity and authenticity of the bitstream with a MAC [78] or an AE scheme [208]. Modern high-end Xilinx FPGAs, starting from the Virtex 6 family, support HMAC-SHA256-based *bitstream authentication*, in combination with AES-based bitstream encryption.

This defense mechanism requires on-chip non-volatile memory for secret key storage. Originally Xilinx relied on battery backed RAM, which complicates (semi-)invasive attacks to recover the secret decryption key, but poses maintenance problems if the battery fails [272, 286]. In Lattice's volatile FPGAs the key is stored with one-time-programmable fuses. Altera supports both battery backed volatile storage and non-volatile storage with polyfuses.[5] The programming of fuses generally requires higher voltages. Due to the necessity of this persistent key storage, the cost of these FPGAs is higher than that of the pure volatile FPGAs. Therefore, this countermeasure is mainly present in high-end FPGA families. Since the Virtex-6 family, Xilinx FPGAs also offer the option to store the bitstream encryption keys with eFuse technology as an alternative for battery backed RAM.

In 2011 Moradi et al. demonstrated the real-world feasibility of a side channel analysis on Xilinx's bitstream decryption engine. In [200] they describe a successful power attack on the 3DES engine of Virtex II Pro FPGAs and in [201] they outline a power attack on the AES engine of the Virtex 4, Virtex 5 and Spartan 6 family. It is reasonable to assume that similar attacks can also be mounted the products of other FPGA vendors. Independent security test laboratories have also demonstrated practical power and EM attacks on the bitstream decryption engine of commercial FPGAs.

### Node Locking

The basic idea of node locking is that the bitstream is bound to the platform by an identifier that cannot be modified by an attacker. This identifier is stored either in an external chip (such as the Dallas/Maxim Secure EEPROM [3, 14]) or internally as a unique serial number (e.g., Xilinx's Device DNA [245]) or a PUF [117, 118, 119]. In order to activate an IP core or the full FPGA design,

[5] An extra fuse can be programmed to put the device into a "tampering protection" mode. From this moment on, the FPGA can only be configured with encrypted bitstreams. In contrast, Xilinx FPGAs will always accept unencrypted bitstreams, even if a bitstream decryption key has been programmed in the device.

an activation code is computed based on the device identifier and it is written to the external non-volatile memory, which also stores the bitstream. Afterwards the design can determine whether it runs on the expected device or not, by verifying if the external activation code corresponds with its device identifier.

The security requirements for the generation of the activation code differ depending on the identifier used. Device serial numbers such as the Device DNA of Xilinx FPGAs can be read by anyone (e.g., over JTAG) and hence the generation algorithm has to be based on some secret (i.e., a proprietary algorithm, a secret symmetric key, or a private key). When a PUF is used and its response cannot be read externally,[6] the algorithm can be kept public.[7] The verification algorithm is part of the bitstream and consequently it is a potential attack target. This is why the security of node locking strongly depends on the obscurity of the bitstream encoding to thwart reverse engineering and the ability to prevent or harden readback difference attacks.

The combination of partial reconfiguration, bitstream encryption and a device identifier allows for novel IP protection schemes. For instance, in [183] we presented a pay-per-use licensing scheme for hardware IP on modern SRAM-based FPGAs that support this combination of features. In our scheme a bootstrapping bitstream loads encrypted partial bitstreams that represent IP cores of different providers, and a remote activation protocol is used to acquire a license key for the different IP bitstreams.

### Non-Volatile FPGAs

The configuration of a non-volatile FPGA is inherently protected as it is stored in internal non-volatile memory. As a result, bitstream cloning, reverse engineering and tampering are only feasible with expensive physical attacks. A number of manufacturers produce devices of this kind. Microsemi (formerly Actel) produces OTP antifuse-based FPGAs and reprogrammable Flash-based FPGAs [1]. The non-volatile devices manufactured by Lattice [174] and Xilinx are hybrid Flash/SRAM-based FPGAs: on startup the content of the SRAM cells is loaded from Flash memory that is integrated on die or in package, respectively.

[6] A separate enrollment bitstream should be used to read out the PUF response.

[7] In the simplest case, the PUF response is used directly as activation code. The bitstream just has to check if the response measured at a later time is sufficiently close to the response during enrollment.

## 6.2 Trusted Computing on Commercial FPGAs

In this section we will study how a TPM can be implemented on commercial FPGAs that are available today, whereas in Section 6.3 we will propose a novel FPGA architecture with built-in support for trusted computing.

More specifically, we will investigate how the techniques from Chapter 4 can be combined with PUF-based key storage to protect the TPM's non-volatile state on different FPGA types. Since volatile FPGAs do not have non-volatile memory on board, we propose to extract the secret keys that are used in the state protection scheme, from an intrinsic PUF. Our solution only relies on the complexity of full bitstream reversal and can hence be realized with current low-end FPGA technology which does not have any additional built-in security mechanisms. This difficulty is not exactly measurable and does not achieve nowadays cryptographic standards, but this approach is superior to embedding the keys directly in the configuration bitstream. Clearly, the security of our proposal can be strengthened if the FPGA supports bitstream encryption or if the configuration bitstream can be stored internally in the FPGA.[8]

The fact that a reconfigurable system-on-chip design is stored as a configuration bitstream, enables field upgrades of the trusted module. In this section we will also cover how this can be done securely.

### 6.2.1 Protection of Non-Volatile State

In Chapter 4 we analyzed how a trusted module can be integrated in computing platforms that lack on-chip NVM and, more specifically, we provided solutions to protect the persistent state in external non-volatile memory. The solutions typically encrypt and authenticate the externalized state with a symmetric key that is embedded in the trusted module. The main challenge, however, is the detection of state replay as this requires a source of freshness within the trusted module.

In [228] we showed that PUFs are well suited to embed a secret key in the configuration bitstream of an FPGA and hence we are convinced that PUFs are an important enabling technology to achieve secure persistent storage on today's commercial FPGAs. In order to detect state replay we made use of external authenticated NVM, which is included in the security perimeter of the trusted module with a minimal cryptographic protocol. The concept of extending the trusted module's security perimeter was also covered in detail in Section 4.4.

[8] In this scenario it is not strictly necessary to derive the secret keys from a PUF.

### Internal NVM with Bitstream Restriction

Reprogrammable non-volatile FPGAs store their configuration bitstream in integrated Flash memory. This removes the need for an external configuration NVM chip and it improves the security of the device by eliminating the possibility of a bitstream interception. Often a dedicated part of the internal non-volatile memory can be used by the FPGA application as persistent storage. This makes these devices highly suited for reconfigurable trusted computing. However, not all Flash-based FPGAs offer the same security level.

Modern Lattice and Microsemi FPGAs have advanced options to limit the reconfigurability [1, 174]:

- The device can be *temporarily locked* with a password such that it can only be unlocked and reprogrammed by providing the correct password.
- A *permanent lock* disables reconfiguration, effectively turning it into a one-time-programmable FPGA.
- When bitstream encryption is enabled, only bitstreams encrypted with the correct key will be loaded.

The protection of the state of a (reconfigurable) trusted module is straightforward on this type of devices. Firstly, reprogramming of the FPGA has to be restricted, either with the locking mechanism or with the bitstream encryption functionality. The latter approach facilitates upgrades of the trusted module in the field. Secondly, the persistent state $\mathcal{T}$ can be stored directly (in plain) into the application accessible NVM. If the size of the NVM is too small to fit the whole persistent state, $\mathcal{T}$ can be stored externally, albeit encrypted and authenticated with the techniques described in Section 4.2. In this case, only the state protection key $k_{\mathcal{T}}$ and (optionally) a replay detection nonce $n_{\mathcal{T}}$ have to be stored in the FPGA's internal NVM.

### Internal NVM without Bitstream Restriction

The Xilinx Spartan-3AN is an example of a hybrid SRAM/Flash-based FPGA, that consists of a volatile Spartan-3 FPGA and a Flash memory chip integrated in the same device.[9] The integrated NVM is primarily attractive for cost saving, as there is no need for a separate off-chip NVM for the storage of the

[9] The FPGA chip and the Flash memory chip are bundled in the same package and not integrated on the same die.

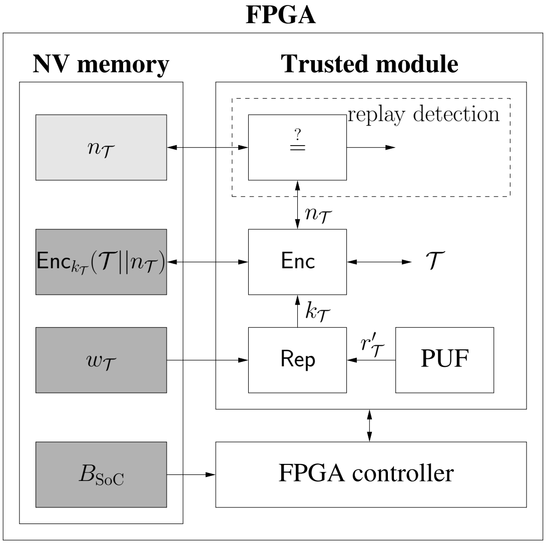


Figure 6.1: Trusted computing on non-volatile FPGAs without configuration locking.

configuration bitstream. The embedded NVM can also be used by the design that is programmed on the FPGA, to persistently store state information.

However, unlike the non-volatile FPGAs mentioned above, the Spartan-3AN does not support any mechanism to restrict its configuration: it does not provide a locking mechanism for the configuration bitstream, nor does the Spartan-3 family support bitstream encryption [311]. Consequently the security level offered by this product is limited.

Xilinx FPGAs support different configuration modes and the selection of the mode is done with external pins. There are security bits inside the bitstream to disable readback and (partial) reconfiguration, but these restrictions are only enforced once the bitstream has been loaded onto the FPGA. An attacker can always put the device into a different mode (e.g., Joint Test Action Group (JTAG) configuration) with the external FPGA pins and subsequently load a malicious bitstream that reads out and/or overwrites the content of the user Flash memory.

This signifies that an FPGA application – in our case a reconfigurable trusted module – must not store sensitive information in the internal NVM without an appropriate state protection scheme. Figure 6.1 illustrates a possible solution that relies on PUF-based key storage and that protects the state $\mathcal{T}$ in the following manner:

- The trusted module's persistent state $\mathcal{T}$ is encrypted and authenticated with the $\mathsf{Enc}()$ algorithm.
- The secret key $k_{\mathcal{T}}$ is derived from a PUF that is embedded in the bitstream $B_{SoC}$. The challenge $c_{\mathcal{T}}$ to the PUF, which is not shown explicitly in Figure 6.1, is also embedded in the bitstream. Alternatively, the key $k_{\mathcal{T}}$ can be derived with a secret algorithm from the FPGA's serial number.
- The associated helper data $w_{\mathcal{T}}$ is also stored in the internal NVM. This allows the trusted module to reconstruct the key $k_{\mathcal{T}}$ from the noisy PUF response $r'_{\mathcal{T}}$ with the $\mathsf{Rep}$ algorithm of the fuzzy extractor.
- In order to detect state replay, a monotonic counter $n_{\mathcal{T}}$ is included in the encrypted state and stored in the integrated Flash memory.

The nonce $n_{\mathcal{T}}$ must not be kept secret and therefore it can be read unrestrictedly. However, it is essential that the nonce is protected against replay (i.e., decreasing the counter to an earlier, lower value). We propose to protect the monotonicity of the counter with the *sector lockdown* mechanism of the integrated Flash memory.[10] This functionality permanently and irreversibly protects the contents of an individual Flash sector against program and erase cycles. Every time the state $\mathcal{T}$ is changed, the counter $n_{\mathcal{T}}$ is incremented and a new sector of the NVM is locked down, effectively emulating the blowing of a fuse.

Sector lockdown is a very expensive operation because valuable Flash memory is rendered unusable. However, it does make sense when implementing the MTM bootstrap counter, which is used to detect firmware rollback. This counter is only 5 bit long and thus it can be implemented with 31 sector lockdown operations.

### Authenticated External NVM

In Section 4.4 we proposed two protocols to extend the security perimeter of the trusted module to the external NVM module. The first realizes an authenticated memory interface and the second an encrypted memory interface. In both protocols a secret key ($k_{\text{auth}}$ and $k_{\mathcal{T}}$ respectively) is shared between the trusted module and the external memory, and this key is programmed during a paring phase. The read operation is protected against replay with a nonce that

[10] The integrated Flash memory also has a security register, which contains a 64-byte unique identifier and a 64-byte user-defined field. The complete user-defined field can only be programmed once. Therefore it is not suited to realize a counter.

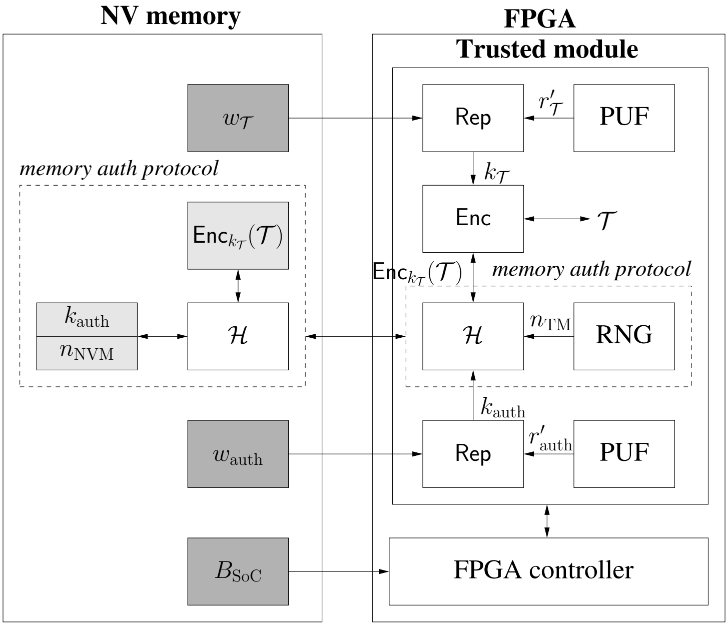


Figure 6.2: Trusted computing on volatile FPGAs with authenticated external NVM.

is generated randomly by the trusted module, and the write operation with a monotonic counter that is stored persistently in the external memory.

Most FPGAs that are sold today, are truly volatile and have no internal reprogrammable non-volatile memory. Devices that support bitstream encryption, do contain some memory to store a bitstream decryption key, but this memory is often only one-time-programmable and can never be used as persistent storage for the FPGA application. In [228] we proposed to realize a reconfigurable trusted module on standard volatile FPGAs by combining a memory authentication protocol with PUF-based key storage.

Figure 6.2 gives a schematic overview of the resulting solution. The following components can be identified:

- The state $\mathcal{T}$ is encrypted and (optionally) authenticated with a secret key $k_{\mathcal{T}}$. For efficiency reasons, we propose to use AES in an AE mode as encryption algorithm $\mathsf{Enc}$, even though this symmetric cipher is not part of the TCG specifications (see Section 4.2.3).

- The state encryption key $k_{\mathcal{T}}$ is stored with a PUF that is embedded inside the bitstream $B_{\text{SoC}}$.
- The corresponding helper data $w_{\mathcal{T}}$ is stored in the external NVM such that $k_{\mathcal{T}}$ can be reliably reconstructed from a noisy PUF response $r'_{\mathcal{T}}$.
- A challenge-response protocol guarantees the freshness of an interaction between the trusted module and the external NVM chip. The trusted module uses a random challenge $n_{\text{TM}}$ when reading for the NVM and the monotonic counter $n_{\text{NVM}}$ prevents replay of previous write operations. We propose to use the HMAC algorithm for $\mathcal{H}$ since this is already present in a TCG compliant trusted module. Another option is a MAC algorithm based on a block cipher (e.g., AES), since we also propose to implement the AES algorithm in the trusted module for encryption of the state $\mathcal{T}$.
- On the trusted module the secret key $k_{\text{auth}}$ used in the memory authentication protocol is derived from a PUF response $r'_{\text{auth}}$, while on the NVM module it is stored internally in its non-volatile memory. This shared authentication key must be programmed in both devices during a pairing phase.

Even though two PUFs are shown in Figure 6.2, the secret keys $k_{\mathcal{T}}$ and $k_{\text{auth}}$ can be derived from the same PUF by applying different challenges. In this case, the two challenges $c_{\mathcal{T}}$ and $c_{\text{auth}}$ are embedded in the bitstream $B_{\text{SoC}}$ or they are stored separately in the external NVM. The PUF guarantees that, given the challenges, its responses are still unpredictable.

### Security Assumptions

The security of the three schemes presented above relies on a number of assumptions:

- The system designer who creates the configuration bitstream $B_{\text{SoC}}$ has to be trusted. This entity can always generate a malicious bitstream to extract sensitive data protected by the trusted module or include a backdoor in the original bitstream.
- Full bitstream reversal is practically infeasible. If an adversary can successfully reverse engineer the bitstream, he will precisely know the type and exact location of the PUF in the reconfigurable logic. With this knowledge he can make a malicious bitstream (containing the same PUF) that outputs the PUF responses ($r'_{\mathcal{T}}$ and $r'_{\text{auth}}$), the secret keys ($k_{\mathcal{T}}$

and $k_{\mathrm{auth}}$) or the decrypted state $\mathcal{T}$. Note that this assumption is not necessary when the bitstream is encrypted or stored in internal NVM.

- Readback is not supported or disabled. This prevents attacks that directly read the unencrypted state or the secret keys directly from the FPGA's memory.
- The attacker does not perform physical attacks. Otherwise he could for instance re-enable readback or read/modify the content of the internal as well as the authenticated external NVM. We argue that (semi-)invasive attacks are expensive on modern FPGAs. In addition, observe that the TPM and MTM are not required to withstand hardware attacks.
- Appropriate countermeasures are taken in the implementation of the cryptographic algorithms $\mathsf{Enc}$ and $\mathcal{H}$ and the fuzzy extractor in order to resist side-channel attacks.

### Enrollment Phase

In order to use a PUF to generate keys, an enrollment phase has to be carried out. During this phase the PUF is challenged for the first time with the challenges $c_{\mathcal{T}}$ and $c_{\mathrm{auth}}$ and the responses $r_{\mathcal{T}}$ and $r_{\mathrm{auth}}$ are measured. The $\mathsf{Gen}$ function of the fuzzy extractor is used to generate the keys $k_{\mathcal{T}}$ and $k_{\mathrm{auth}}$ for the first time together with their accompanying helper data $w_{\mathcal{T}}$ and $w_{\mathrm{auth}}$. The helper data are then stored in the non-volatile memory. We note that in the case of authenticated external NVM this phase can be carried out during the pairing phase of the memory authentication protocol.

The system designer can choose to create a separate enrollment bitstream $B_{\mathrm{PUF}}$ that contains the same PUF as the bitstream $B_{\mathrm{SoC}}$, that will be deployed afterwards.

## 6.2.2 Protection of the Bitstream

The bitstream $B_{\mathrm{SoC}}$ contains the system-on-chip design including the trusted module. The following security requirements should be considered:

1. **Design integrity:** Unauthorized modification of the system-on-chip design must be impossible. More specifically the integrity of a number of components should be guaranteed: the trusted module, especially its firmware, the main processor, the CRTM code, and the communication interface between the trusted module and the CPU.

2. **Design confidentiality:** The design can contain cores whose intellectual property has to be protected. Additionally, the cryptographic keys that are embedded in the trusted module, must remain secret.

3. **Design freshness:** Reconfigurable hardware allows field upgrades of the design. It must not be possible to replay an older and insecure version of the design.

The bitstream is stored in the configuration memory of the FPGA. This memory is internal NVM, in the case of a non-volatile FPGA or an SRAM-based FPGA with integrated Flash memory, and external NVM, for of a truly volatile FPGA. In the latter case, the regular memory interface must be used, because commercial FPGAs do not support the memory authentication protocol and because there exists no standard for authenticated NVM.

### Bitstream Obfuscation

On low-end reconfigurable devices where the configuration bitstream can be intercepted, we can only rely on the reverse engineering complexity of the bitstream encoding for security purposes. According to the above mentioned literature, this gives a decent level of assurance that IP cores cannot easily be reverse engineered and that directed modification of the logic is difficult.

An adversary can extract the challenge $c_{\mathcal{T}}$ and $c_{\text{auth}}$ from the bitstream, but due to the unpredictability of the PUF responses, this knowledge is insufficient to learn any information about the keys $k_{\mathcal{T}}$ and $k_{\text{auth}}$. In order to be successful, he must perform a full bitstream reversal and create a malicious design with exactly the same PUF as $B_{\text{SoC}}$ to output the secret keys. According to the state of the art [79], this is currently infeasible.

### Embedded ROMs

If the CRTM code and the trusted module's firmware are embedded inside the bitstream $B_{\text{SoC}}$, partial bitstream reversal will possibly reveal the contents of these embedded ROMs, which we denote with $M_{\text{ROM}}$. This is a serious threat as this could enable an attacker to create a bitstream with a modified TPM firmware or CRTM, and circumvent the TCG chain of trust or to extract the persistent state.

The easiest way to overcome this problem is by storing these embedded ROMs in the authenticated non-volatile memory. The system-on-chip design needs to be extended with some extra logic that performs the cryptographic protocol to

Table 6.1: Content of external NVM.

| name | type | description |
|---|---|---|
| $B_{\text{SoC}}$ | regular | FPGA bitstream containing SoC design |
| $w_{\mathcal{T}}$ | regular | helper data for state encryption key $k_{\mathcal{T}}$ |
| $w_{\text{auth}}$ | regular | helper data for NVM authentication key $k_{\text{auth}}$ |
| $\mathsf{Enc}_{k_{\mathcal{T}}}(\mathcal{T})$ | authenticated | encrypted persistent state $\mathcal{T}$ |
| $M_{\text{ROM}}$ | authenticated | trusted module firmware ROM and CRTM |
| $PK_{\text{ROM}}$ | authenticated | public key to verify signed ROM updates |

access the authenticated NVM. This logic has to have access to $k_{\text{auth}}$, that is stored with the intrinsic PUF.

#### Bitstream Encryption

On high-end FPGAs bitstream encryption can be used to obtain better design confidentiality. Additionally, it is difficult to make meaningful modifications to the design if the bitstream is encrypted. Typically the plain bitstream contains redundancy checks (e.g., CRC), so bit flips get detected. Ideally, the bitstream should be cryptographically authenticated as well, for instance with an additional MAC algorithm or by using an AE scheme. Modern commercial FPGA hardware, such as Xilinx Virtex-6 and Virtex-7, support this functionality.

Bitstream encryption support does not necessarily imply that the design freshness requirement is satisfied. Even if the bitstream is encrypted in the configuration memory, an attacker can revert a field update by overwriting the encrypted bitstream with an older version.

### 6.2.3 Field Updates

The TCG specifications define the TPM_FieldUpgrade command, but the implementation is free. Typically this command is used to update the firmware of the trusted module. However, the advantage of an FPGA is the possibility to also update the hardware implementation of the TPM. Hence two type of field updates can be distinguished, namely *firmware updates* and *bitstream updates*.

We proposed to use the authenticated external NVM in order to implement a trusted module on a purely volatile FPGA. Table 6.1 summarizes the content of the external non-volatile memory and which memory interface is used to access the data. The firmware of the trusted module is stored in authenticated NVM as $M_{\text{ROM}}$, whereas the trusted module's hardware is stored separately in regular NVM as part of the bitstream $B_{\text{SoC}}$.

### Firmware Updates

Firmware updates are fairly straightforward as we can rely on the authenticated memory interface of the external NVM. The trusted module must offer an extra command to verify a signed firmware image $\mathsf{sign}_{SK_{\text{ROM}}}(M'_{\text{ROM}})$. We propose to store the public key $PK_{\text{ROM}}$, which will be used to verify the digital signature on the firmware update, in authenticated NVM in order to protect its integrity. Once the trusted module has verified the digital signature, it can overwrite $M_{\text{ROM}}$ with the authenticated memory interface.

In order to protect against rollback,[11] a version number $v_{\text{ROM}}$ needs to be included in the firmware ROM: $v_{\text{ROM}} \subset M_{\text{ROM}}$. The trusted module has to check whether the version of the new firmware is higher than its own version: $v'_{\text{ROM}} > v_{\text{ROM}}$.

Alternatively, the update key $PK_{\text{ROM}}$ and the firmware version $v_{\text{ROM}}$ can be stored inside $\mathcal{T}$. The version number should be included explicitly in the firmware update: $\mathsf{sign}_{SK_{\text{ROM}}}(M'_{\text{ROM}} || v'_{\text{ROM}})$. With both approaches, the authenticated memory interface assures that only the trusted module can update its firmware.

### Bitstream Updates

In some situations (e.g., to replace a cryptographic coprocessor), it might be necessary to update the FPGA bitstream $B_{\text{SoC}}$, instead of only the firmware image $M_{\text{ROM}}$. It is crucial that the new bitstream includes exactly the same PUF. Otherwise, the secret keys $k_{\text{auth}}$ and $k_{\mathcal{T}}$ become inaccessible and consequently the trusted module can no longer access its persistent state.

Because $B_{\text{SoC}}$ is stored in regular external NVM, it can always be overwritten by an older version. A possible solution to prevent this is to bind the bitstream $B_{\text{SoC}}$ to the firmware $M_{\text{ROM}}$. Every bitstream update will then be accompanied by a firmware update. The binding mechanism has to assure that the new firmware $M'_{\text{ROM}}$ does not function correctly with the old bitstream $B_{\text{SoC}}$. For instance, some extra logic in $B_{\text{SoC}}$ checks whether an expected identifier is present in $M_{\text{ROM}}$ and halts the design if this is not the case. Like the node

[11] Version rollback could be desirable because updates can cause issues.

locking schemes described in Section 6.1, this solution relies on the difficulty of bitstream reversal.

## 6.3 Trusted FPGA Architecture

As illustrated in the Section 6.1, manufacturers are aware of the security issues of volatile FPGAs and they are slowly adding more security functionalities. In [87] we proposed a novel FPGA architecture for realizing trusted computing functionalities in reconfigurable hardware.

The research started from the existing concept of having a small security engine in the static FPGA fabric to decrypt and authenticate the bitstream before it is configured on the reconfigurable logic. We went a step further by transforming this security engine into a TCG-like root of trust that can attest the loaded bitstream and seal and unseal sensitive data when a specific bitstream has been configured on the FPGA. With our trusted FPGA architecture we can realize a reconfigurable TPM that can include its own hardware implementation (i.e., the bitstream) in the TCG chain of the trust.

A possible extension to our architecture would be to make the security engine responsible for binding (partial) bitstreams to specific FPGAs based on a license, and thus allowing more elaborate pay-per-use IP licensing schemes. Such extension is elaborated in the TinyTPM work [97] of Feller et al.

### 6.3.1 Underlying Model

The main parties involved are FPGA *manufacturers*, *hardware IP developers* (e.g., developing the application logic synthesized to a bitstream), *software IP developers* who implement software that runs on the loaded bitstream on the FPGA, *system developers* who integrate hardware and software IP onto an FPGA platform and the *user* who employs the device. All parties trust the FPGA hardware manufacturer since there is no publicly known efficient mechanism to verify an ASIC implementation for correctness or potential trapdoors. However, IP developers have only limited trust in systems developers, and users have only limited trust in IP and system developers. It is obvious that the entity issuing the update (usually the TPM designer) needs to be trustworthy, or the TPM implementation is subject to certification by some trusted organization.

We assume an *adversary* who can eavesdrop and modify all external communication lines of the FPGA, eavesdrop and modify all memories external to the FPGA, arbitrarily reconfigure the FPGA, but *cannot* eavesdrop or modify internal states of the FPGA. Particularly, we exclude invasive attacks such as glitch attacks, microprobing attacks or attacks using laser or FIB to gain or modify FPGA internals. Precautions against other physical attacks such as side channel attack or non-invasive tampering must be taken when implementing the TPM. Furthermore, we do not consider any destructive adversaries which are focusing on denial-of-service attacks, destroying components or the entire system.

### 6.3.2 Basic Idea and Design

The basic idea is to include the hardware configuration bitstream(s) of the FPGA in the chain of trust. The main security issue, besides protection of the application logic, is to protect the TPM against manipulations, replays and cloning (see security requirements in Section 6.2.2). Hence, appropriate measures are required to securely store and access the sensitive TPM state $\mathcal{T}$.

In the following we denote a hardware configuration bitstream as $B_X$ with $X \in \{\mathrm{TPM}, \mathrm{App}\}$ such that $B_{\mathrm{TPM}}$ denotes a TPM bitstream and $B_{\mathrm{App}}$ an application bitstream. We further define $E_X$ as the encryption of $B_X$ using a symmetric encryption algorithm and a symmetric encryption key $k_X^{\mathrm{enc}}$ such that $E_X = \mathsf{Enc}_{k_X^{\mathrm{enc}}}(B_X)$. We define $A_X$ as an authenticator of a bitstream $B_X$ with $A_X = \mathsf{Auth}_{k_X^{\mathrm{auth}}}(B_X)$ where $\mathsf{Auth}_{k_X^{\mathrm{auth}}}$ could be for instance a MAC based on the key $k_X^{\mathrm{auth}}$. We denote the corresponding verification algorithm of an authenticator $A_X$ with $\mathsf{Verify}_{k_X^{\mathrm{auth}}}(B_X, A_X)$. If a bitstream has been encrypted to preserve design confidentiality, $B_X$ is replaced by $E_X$. Thus, the corresponding authenticator $A_X$ becomes $A_X = \mathsf{Auth}_{k_X^{\mathrm{auth}}}(E_X)$. As already mentioned in Section 4.2.3, such an Encrypt-then-MAC scheme provides the strongest security (with respect to the two other possible schemes MAC-then-Encrypt and Encrypt-and-MAC).[12] We finally define $C_X$ as an unique representative of $B_X$'s configuration, e.g., a cryptographic hash value which can be based on a block cipher [30, 171, 218, 222, 251]. If $C_{\mathrm{TPM}}$ is a hash value, it represents a measurement that conforms to the TCG approach. However, alternative approaches may use for $C_{\mathrm{TPM}}$, e.g., a property certificate about $B_{\mathrm{TPM}}$ signed by a trusted third party that is included within the corresponding authenticator.

Figure 6.3 shows our high-level reconfigurable architecture. The bitstreams $B_{\mathrm{App}}$ and $B_{\mathrm{TPM}}$ of the application and the TPM core (without any state $\mathcal{T}$) respectively are stored authenticated (and encrypted) in the external (untrusted) memory. The FPGA control logic allows partial hardware configuration of the

[12] Xilinx on the other hand uses the MAC-then-Encrypt option in the Virtex-6 family.

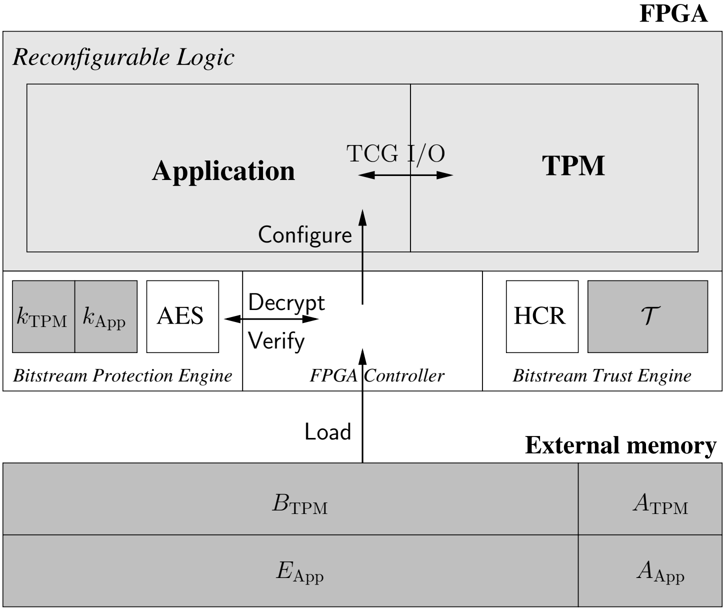


Figure 6.3: Trusted FPGA architecture.

FPGA fabric to load the TPM and the application independently using the Load and Configure interfaces.

The Bitstream Trust Engine (BTE) provides means to decrypt and verify the authenticity and integrity of bitstreams using the Decrypt and Verify interfaces, which are already present in modern high-end FPGAs. Furthermore, the BTE includes a protected and non-volatile key storage to store the keys for bitstream decryption and authentication. Finally, the BTE provides a volatile memory location called Hardware Configuration Registers (HCRs) to store the configuration information of loaded bitstreams. These registers are used later on by the TPM to set up its internal PCRs. In the following we define two stages in our protocol, the *setup* and the *operational* phase.

### 6.3.3 Setup Phase

To enable an FPGA with trusted computing functionality, a TPM issuer designs a TPM and synthesizes it to a (partial) bitstream $B_{\mathrm{TPM}}$ for use on an FPGA. Furthermore, we assume that an application designer provides a trusted computing enabled FPGA application delivered as partial bitstream $B_{\mathrm{App}}$ which can interact with the TPM architecture using a well-defined interface. Of course, particularly when using an open TPM implementation, it is possible that both components are developed by a single party, e.g., by the system developer itself.

1. The system developer verifies the authenticity of $B_{\mathrm{TPM}}$ and $B_{\mathrm{App}}$, encrypts $B_{\mathrm{App}}$ to $E_{\mathrm{App}}$ and then creates bitstream authenticators $A_{\mathrm{TPM}}$ and $A_{\mathrm{App}}$ using the keys $k^{\mathrm{auth}}_{\mathrm{TPM}}$ and $k^{\mathrm{auth}}_{\mathrm{App}}$ respectively. Note that if TPM bitstream $B_{\mathrm{TPM}}$ is also provided by the system integrator itself, he can choose $k^{\mathrm{auth}}_{\mathrm{TPM}} = k^{\mathrm{auth}}_{\mathrm{App}}$.

2. The TPM bitstream $B_{\mathrm{TPM}}$, its authenticator $A_{\mathrm{TPM}}$, the encrypted application bitstream $E_{\mathrm{App}}$, and its authenticator $A_{\mathrm{App}}$ are stored in the external memory.

3. The system developer writes the appropriate authentication keys $k^{\mathrm{auth}}_{\mathrm{TPM}}$ and $k^{\mathrm{auth}}_{\mathrm{App}}$ (and the encryption key $k^{\mathrm{enc}}_{\mathrm{App}}$) to the key store of the BTE.

### 6.3.4 Operational Phase

Remember that on each power-up the FPGA needs to reload its hardware configuration from the external memory. Hence, for loading a trusted computing enabled application, the following steps need to be accomplished:

1. On device startup the FPGA controller reads the TPM bitstream $B_{\mathrm{TPM}}$ and the corresponding authentication information $A_{\mathrm{TPM}}$ from the external memory. BTE verifies the authenticity and integrity of $B_{\mathrm{TPM}}$ based on the authenticator $A_{\mathrm{TPM}}$ by using $\mathsf{Verify}_{k^{\mathrm{auth}}_{\mathrm{TPM}}}(B_{\mathrm{TPM}}, A_{\mathrm{TPM}})$. After successful verification, BTE computes the configuration value $C_{\mathrm{TPM}}$ of the TPM bitstream and writes $C_{\mathrm{TPM}}$ into the first Hardware Configuration Register (HCR) before the FPGA's fabric is finally configured with $B_{\mathrm{TPM}}$.

2. The TPM requires exclusive access to a non-volatile memory location to store its sensitive state $\mathcal{T}$. Furthermore, the access to this storage location is protected by the BTE which provides access to sensitive data only when a specific bitstream (i.e., the TPM) is loaded. This access control function is equivalent to TCG *sealed storage*. For full flexibility, the BTE implements an interface with which a currently configured bitstream can request a reset (and implicitly, a clear) of the non-volatile memory to reassign the access to the storage for its own exclusive use. The access authorization to the memory for a loaded bitstream $X$ can easily be performed by BTE by checking its $C_X$ stored in the first HCR.

This step necessitates on-chip NVM, which, as discussed earlier, in practice is rarely available in volatile FPGA. However, this restriction can be overcome with the state protection techniques described in Chapter 4, e.g., by extending the security perimeter to the external NVM (see also Section 6.2).

3. After the TPM has been loaded onto the fabric, the application bitstream $E_{\text{App}}$ and its authenticator $A_{\text{App}}$ are read from the external memory, verified and decrypted in the same way. The BTE stores the configuration value $C_{\text{App}}$ of the verified application in the second HCR register.
4. After the application bitstream has been configured in the fabric, the first call of the application to use the trusted computing functionality will initialize the TPM as follows. Based on the content of the two HCR registers, the TPM initializes its own PCRs. It reads the recorded bitstream configurations $(C_{\text{TPM}}, C_{\text{App}})$ and extends them in the PCRs. In this way the (unique) configurations of all bitstreams can be included in the chain of trust. This is similar to the initialization of a desktop TPM via CRTM. However, now the PCR includes the hardware measurement results of the TPM itself.

After loading the hardware configuration of the TPM and the application into the FPGA, the chain of trust can be extended by the measurements of other specific platform components such as the operating system and high-level application software. This allows to bind any higher level application (of the IP provider) to the underlying FPGA by binding the application (or its data) using the subset of the PCR registers that contain the corresponding measurements of the underlying FPGA.

### 6.3.5 TPM Updates

The update of the current $\text{TPM}_1$ to another $\text{TPM}_2$ on an FPGA is quite easy when the sensitive state $\mathcal{T}$ does not need to be migrated. The $\text{TPM}_2$ needs to be loaded and will obviously not be able to access the BTE's protected storage containing $\mathcal{T}$ of $\text{TPM}_1$ (since $\text{TPM}_2$ cannot provide $C_{\text{TPM}_1}$). Hence, $\text{TPM}_2$ reassigns the protected storage to be able to create and store its own $\mathcal{T}$. With the reset of the protected storage, the previous $\mathcal{T}$ in the non-volatile memory is cleared so that no confidential information of $\text{TPM}_1$ will be accessible for $\text{TPM}_2$. To prevent denial-of-service attacks against $\mathcal{T}$, BTE can additionally implement a mechanism such that $\text{TPM}_1$ has to clear its $\mathcal{T}$ before $\text{TPM}_2$ is able to reassigns the protected storage for its own $\mathcal{T}$.

However, for migrating $\mathcal{T}$ from TPM$_1$ to TPM$_2$ without loss of $\mathcal{T}$, we propose to extend existing TPM implementations by a migration function[13] $\mathsf{Migrate}(A_{\text{update}}, C_{\text{TPM}_2})$ where $A_{\text{update}}$ is an *update authenticator* and $C_{\text{TPM}_2}$ a unique reference to the corresponding TPM$_2$. For a TPM update, a system developer (who has set $k^{\text{auth}}_{\text{TPM}}$ for the corresponding FPGA) generates an update authenticator

$$A_{\text{update}} = \mathsf{sign}_{\text{SK}_{\text{update}}}(C_{\text{TPM}_2}, P_{\text{TPM}}),$$

where SK$_{\text{update}}$ denotes an update signing key. TPM$_1$ trusts the corresponding update verification key PK$_{\text{update}}$, e.g., as it is pre-installed in TPM$_1$.

Thus TPM$_1$ knows a set of trusted update authorities (the system developer, etc.) who are allowed to perform the migration of $\mathcal{T}$ for use with TPM$_2$. $P_{\text{TPM}}$ denotes a reference to the class of TPMs that provides a certain (minimum) set of security properties. Note that $P_{\text{TPM}}$ can also be replaced by individual update signing keys each representing a single security property.

When the user requests a TPM update, he invokes the migration function of TPM$_1$ using the parameters $A_{\text{update}}$ and $C_{\text{TPM}_2}$ received from the corresponding system developer (over an untrusted channel). Then, the migration function $\mathsf{Migrate}(A_{\text{update}}, C_{\text{TPM}_2})$ of TPM$_1$ performs the following steps:

1. It verifies $A_{\text{update}}$ using the update verification key PK$_{\text{update}}$ and checks whether $P_{\text{TPM}_2}$ provides the same (minimum) set of security properties as $P_{\text{TPM}_1}$.

2. After successful verification, it reassigns the BTE's protected storage, which contains $\mathcal{T}$, for use with TPM$_2$. The BTE needs to grant TPM$_2$ access to the protected NVM without erasing its content. More precisely, the BTE provides an interface so that only TPM$_1$ can associate the protected storage with $C_{\text{TPM}_2}$. After reassignment of the protected memory, only the new TPM$_2$ is able to access $\mathcal{T}$.

After the migration function has terminated, the application (or manually, the user) overwrites TPM$_1$ stored in the external memory with $B_{\text{TPM}_2}$ and the corresponding authenticator $A_{\text{TPM}_2}$. Now, the user restarts the FPGA to reload the updated TPM and application.

[13] Note, our migration functionality does *not* replace the TCG mechanisms for migrating internals called TPM_Migration and TPM_Maintenance.

### 6.3.6 Discussion

#### Advantages

Enhancing an FPGA with trusted computing mechanisms in reconfigurable logic can provide the following benefits.

**Enhancing the Chain of Trust.** As explained in Section 2.1, TPM-enabled systems establish the chain of trust by starting from the CRTM, which is currently part of the BIOS. For FPGA hosted TPMs, the BTE can begin with the hardware configuration of the application and even with the TPM itself. Therefore, the chain of trust can include the underlying hardware as well as the TPM hardware configuration, i.e., the chain of trust paradigm can be moved to the hardware level.

**Flexible Usage of TPM Functionality.** The developer may also utilize the basic functionality of the TPM in his application which can make the development of additional cryptographic units obsolete. This includes the generation of true random numbers, the asymmetric cryptographic engine as well as protected non-volatile memory. Furthermore, a flexible FPGA design allows to use only the specific TPM functionality that is required for the application, yielding a smaller and consequently easier to certify implementation.

**Flexible Update of TPM Functionality.** A TPM implemented in reconfigurable logic of an FPGA can easily be adapted to new requirements or versions. For example, if the default hash function turns out to be not secure enough, an FPGA hosted TPM could include a self-modification feature which updates the corresponding hash function component, in particular no new hardware design is needed. Moreover, patches fixing potential implementation errors or changes/updates enhancing interoperability could be applied quickly and very easily.

**Improved Communication Security.** The integration of CPU, ROM, RAM and TPM into a single chip enhances the protection of communication links between these security critical components from being intercepted or manipulated. Having the boot ROM and RAM integrated on the FPGA chip, makes the injection of malicious boot code or RAM manipulations more difficult.

**Vendor Independence.** Platform owners can select which TPM implementation is operated on their platforms. This allows even the usage of fully vendor independent open TPM implementations providing more assurance regarding trapdoors and Trojans. Moreover, since we can easily implement a TPM soft core into hardware, a multitude of vendors can offer a variety of TPM implementations. Thus, users are not only restricted to a few TPM ASIC manufacturers as today, they even can implement their own TPM instances and have it certified for TCG compliance by a trustworthy authority. In this scenario the users do not have to trust any external entity, except for the FPGA manufacturer.

### Implementation Aspects

Our trusted FPGA architecture uses a number of features that are already present in some commercial FPGAs: partial reconfiguration, bitstream authentication, bitstream encryption and embedded Flash memory. This suggests that a practical implementation of our proposal is technically feasible today. The main components of the BTE that are currently missing, are the hardware configuration registers and the access control mechanism for the embedded NVM.

In order to realize the former, the FPGA controller must be extended to measure and record the loaded bitstreams and an interface must be provided to the reconfigurable logic to read the content of the HCRs. Xilinx FPGAs already have a so-called Internal Configuration Access Port (ICAP) primitive that provides the reconfigurable user logic access to the FPGA's configuration interface and that enables internal readback and (partial) self-reconfiguration. It seems natural to expose the HCRs using this ICAP interface.

The protected storage on the other hand requires a more tight integration of the embedded Flash memory and the FPGA controller. Access to the memory addresses that are used to store the persistent state $\mathcal{T}$ must only be granted if the expected bitstream $B_{\text{TPM}}$ has been recorded in the HCR register.

## 6.4 Conclusion

In Chapter 4 we explained that it is sometimes desirable to embed a trusted computing module in a hardware component that lacks reprogrammable non-volatile memory. One of the examples that we gave, is trusted computing on reconfigurable hardware. In this chapter we explored this example in more detail and we investigated how a trusted module can be implemented on modern

FPGAs, mainly focussing on the protection of the trusted module's persistent state.

We first provided a brief overview of the attacks on FPGAs, such as bitstream cloning and reverse engineering, and of the defense mechanisms that are presently available in commercial FPGA products. Most security problems arise from the fact that the configuration bitstream of volatile FPGAs, which contains intellectual property or security sensitive information, is stored in external NVM, and that it is consequently possible to read and/or modify the bitstream that is loaded on the FPGA. The most popular defense mechanism are schemes to bind the bitstream to a unique device and the encryption of the bitstream. The latter requires some embedded non-volatile memory, typically fuses or battery-backed RAM, to store the bitstream decryption key.

Next we discussed how to protect the persistent state when implementing a trusted module on commercial FPGAs. The state protection schemes, which we described in Chapter 4, require the storage of a secret key and a source of freshness inside the trusted module. We observed that PUFs can uniquely fingerprint an FPGA and hence that they are well suited to derive the secret key(s) of the state protection scheme. In our discussion we postulated that it is complex to extract a PUF-derived key by reverse engineering a configuration bitstream. If the underlying FPGA supports bitstream encryption, this assumption must not be made. The detection of state replay proves challenging on a volatile FPGA that does not have embedded non-volatile memory and the only proper solution is to extend the security perimeter to the external NVM with a cryptographic protocol.

Finally, we described a trusted FPGA architecture that improves upon solutions for bitstream encryption and bitstream authentication which are present in today's commercial FPGAs. We defined a bitstream trust engine that acts as a root of trust to measure and report the integrity of partial bitstreams. The scheme goes a step further than the $\mu$TPM architecture that was described in the previous chapter, as it allows to attest not only the integrity of the TPM's firmware, but also the integrity of the configuration bitstream that represents the TPM's hardware.

# Chapter 7

# Conclusions and Future Work

## 7.1 Conclusions

This thesis deals with the analysis and design of trusted computing platforms. In our analysis we focussed primarily on the TCG specifications, which introduced the concept of a trusted computing platform that supports remote attestation and sealed storage. Trusted computing technology is considered as a promising enabling technology to improve the trustworthiness of computing platforms.

More than 600 million PCs and laptops have been sold equipped with a TPM, but enterprises have yet to embrace the technology on a large scale. One of the reasons for this limited success is the lack of adequate operating system support. This situation might changed drastically with the arrival of Windows 8. In earlier versions of Windows, the TPM was only used by BitLocker drive encryption to record the integrity of the early boot process (i.e., BIOS and bootloader) and to unseal the disk encryption key. Windows 8 goes much further: it records the integrity of all boot components (including bootloader, kernel, device drivers, and anti-malware software), it supports remote attestation, it offers TPM-based certificate storage, it enables the TPM to acts like a permanently inserted smart card, and it has TPM provisioning software.

The MTM specification of the TCG tried to adapt the TPM to requirements of the mobile phone platform and it can be seen as an attempt to standardize secure boot on mobile phones. This specification has been far less successful than the TPM standard, and it has not been adopted by manufacturers. We believe that there are two main reasons for the failure of this attempt. On the one hand, the mobile phone market is a lot more competitive than the PC world where Intel and Microsoft define the platform. Various processor manufacturers and multiple operating systems (e.g., Android, iOS, Windows Phone, Blackberry OS) exist, which presumably makes it more difficult to agree on a common specification. On the other hand, smart phone platforms are more modern and hence hampered less by backward compatibility requirements. Many mobile phone manufacturers already implement a secure boot mechanism without an MTM, e.g., in order to restrict the phone to a specific mobile operator or to exercise strict control on the applications that get executed on the phones. iOS is the prime example of a mobile ecosystem that relies heavily on code signing. Apple wants that all third party applications are installed through its App Store. A bonus of its closed model is that very few malware exists for the iOS platform.

In Chapter 2 we explained that the TCG's binary attestation approach has some inherent shortcomings. First, it is hard to manage on a large scale because a large number of valid platform configurations could exist and because every software update will yield a different integrity measurement. Second, it only provides assurance about software components that get loaded at boot time, and consequently the platform can still be compromised at runtime. In Chapter 2 we also described an alternative attestation scheme that combines the computation of a runtime memory checksum with the timestamping functionality of the TPM. This scheme requires minimal software support as it only relies on a TPM device driver and optionally a trusted bootloader.

Chapter 3 dealt with the resilience of trusted computing platforms against hardware attacks. Although the TCG specifications only consider software attacks, most TPM implementations are based on a secure microcontroller series and hence apply countermeasures against physical attacks. In this chapter, we investigated which operations, at least in theory, can be targeted with a side channel attack in order to extract the TPM's internal secrets. It remains to be seen how difficult this type of attacks is in practice, but it is clear that any cryptographic implementation can be attacked with sufficient resources. The main conclusion of this chapter, however, is that the communication interface of the TPM is a more attractive attack target. Physical attacks on this interface are not that difficult to perform and require relatively inexpensive equipment, yet they have severe security implications. For instance, cryptographic keys can be captured on the communication bus when they are unsealed, or a TPM reset attack can be used to circumvent the static CRTM. An effective countermeasure against this type of attacks is the integration of the TPM in the chipset, like done by Intel, since it is much more difficult to access and monitor the front-side bus of a PC.

In Chapter 4 we addressed one of the main challenges for integrated TPMs and MTMs, namely the secure storage of its persistent state in external non-volatile memory. We proposed to encrypt and authenticate the externalized state with a secret key that is embedded inside the trusted module. In addition, we proposed to detect replay of older versions of the state either by including a nonce in the state or by updating the secret key. With either approach, the trusted module still needs a small amount of on-chip reprogrammable non-volatile memory. Since Flash memory can often not be embedded in a cost effective manner, the required on-chip NVM must be implemented with another technology. Battery-backed SRAM is a popular choice; this option is for instance used in IBM secure coprocessor products and in FPGAs with bitstream encryption support. If the expected number of state updates is finite and rather small, fuses can be used to implement a monotonic counter; this option is for instance used in game consoles to prevent software downgrading. In Chapter 4 we introduced two alternative solutions that do not require embedded reprogrammable NVM. The first solution derives an updatable state protection key from a reconfigurable PUF, whereas the second extends the security perimeter of the trusted module to the external NVM chip with a cryptographic protocol. For the time being, the first approach remains theoretical in nature, because good practical realizations of the RPUF concept have yet to be found.

The TPM enables novel functionalities such as remote attestation and sealed storage, but it can also be used as a traditional cryptographic coprocessor, e.g., to generate RSA signatures, to maintain a monotonic counter or to generate random numbers. On most platforms third-party software can access the TPM using a Public Key Cryptography Standards (PKCS)#11 interface and with Microsoft Crypto API. For some applications, however, it would be useful if the platform had a secure coprocessor that is freely programmable and that can host multiple applications. In Chapter 5 we introduced the $\mu$TPM architecture for a secure coprocessor. The proposed architecture is inspired by the JavaCard framework, particularly with regard to its multiprocessing support. It can be used to implement a conventional TPM, but it can also run arbitrary security tasks. Two important features differentiate the $\mu$TPM architecture from the JavaCard architecture. First, the program code of the $\mu$TPM is stored in external NVM and gets loaded in internal RAM when needed. In order to keep the size of the embedded RAM small, we propose to split the TPM firmware into different code fragments, each operating on a shared state, and to only load one fragment at once. Second, the $\mu$TPM architecture adopts the trusted computing principle of measured boot. The $\mu$TPM can execute arbitrary program code, but the integrity of code gets measured and it can be reported afterwards.

FPGA devices have become big enough to implement complex reconfigurable SoC designs that contain a softcore processor (e.g., MicroBlaze) and that are capable of hosting an operating system such as Linux. In Chapter 6 we investigated how a TPM or MTM can be integrated into a reconfigurable SoC design. With an FPGA implementation of the TPM standard, it is possible to upgrade the TPM firmware as well as its hardware. Two main topics were covered in this chapter: the protection of the persistent state and the protection of the configuration bitstream on commercially available FPGAs. Since SRAM-based FPGAs typically lack on-chip non-volatile memory, the techniques of Chapter 4 were used to securely externalize the trusted module's persistent module. Furthermore, we introduced PUFs as a technology to derive cryptographic keys on an FPGA, instead of embedding them directly in the configuration bitstream. Finally, we also designed a trusted FPGA architecture that adopts the TCG principle of measured boot and sealed storage.

## 7.2 Directions for Future Research

The publication of the TCG specifications has lead to an active research community in the field of trusted computing. Significant contributions have been made in this research domain, including the analysis of the TCG specifications, applications of trusted computing technology, alternative approaches for remote attestation, improvements to the DAA protocol, software support for trusted computing, etc. In this thesis we covered a number of research topics in the area of trusted computing, most of which deal with hardware aspects. We see that there are still some open research questions and interesting directions for future research.

**Attacks on Trusted Computing Platforms.** In Chapter 3, we did a theoretical analysis of side channel attacks on the TPM. It remains to be investigated how difficult these attacks are in practice on existing TPM implementations. The experiments Kovah et al. [159] with the TPM tick stamping functionality indicate that the Broadcom TPM might be vulnerable to timing attacks. Of course the question can be raised whether the scientific community should perform a security analysis of commercial products. However, this analysis can provide users more confidence in trusted computing technology in general and in specific solutions.

The TCG is currently working on specifications for a next generation of trusted modules, namely for a TPM 2.0 and for an MTM 2.0. In October 2012 a draft of the TPM 2.0 specification has been published for public review. As pointed out in Chapter 3, some minor flaws were found in the TPM 1.1b and 1.2 specifications. So it is essential that the 2.0 specification gets scrutinized by the community.

**Lightweight Trusted Computing.** The TCG specifications and the accompanying extensions to the x86 processor enable new security features such as measured boot, remote attestation and sealed storage on the PC platform. Even though the MTM specification has not been adopted, mobile phone manufacturers are starting to include hardware support (e.g., ARM TrustZone, TI M-Shield, embedded SE) for a trusted execution environment. The TCG Embedded Systems Work Group has yet to release any specifications, but, in the meantime, manufacturers are offering TPM variants with a communication interface (i.e., I2C or SPI) that is more suited for non-PC platforms.

We believe that there is a need for a more minimal root of trust for embedded systems. It is not viable to add a TPM, which in essence is a microcontroller with a RSA coprocessor, to a low-end embedded device. Several software-based remote attestation schemes (e.g., [238]) have been proposed for embedded systems, but they are vulnerable to simple hardware attacks (e.g., increasing the microcontroller's clock frequency) and to the proxy attack that was described in Chapter 2. However, with minor hardware changes, better schemes can be devised. For instance, Schulz et al. [232, 233] constructed a lightweight solution that combines software-based attestation with a PUF. Other promising approaches are the self-protecting modules of Strackx et al. [261, 262] and the SMART scheme of El Defrawy et al. [70], which both use a program-counter dependent memory access control model.

**Physical Unclonable Functions.** In this thesis, we repeatedly used PUFs to derive cryptographic keys. PUF-based key storage is interesting for cost reasons, as it provides an alternative for OTP NVM technology, and from a security respective, because the key is not present when the device is powered off. Various PUF constructions have been proposed in the literature. However, an thorough security analysis of these PUF constructions, in particular regarding their resilience to physical attacks, is missing. Initial work on side channel analysis of PUFs and fuzzy extractors has been presented by Merli et al. [197, 198], but more elaborate research efforts are needed. One concrete idea that is worth investigating, is to verify whether the frequency injection techniques that have been proposed by Markettos and Moore [189] and by Bayon et al. [17] to attack ring oscillator based TRNGs can also be applied on ring oscillator PUFs.

We believe that there is a need for more research on PUFs that are intrinsically present in existing platforms. Intrinsic PUFs are for instance useful to derive a secret key for a software TPM or to bind software to a specific platform. Memory based PUFs (e.g., SRAM PUF) have been found in existing platforms. Holcomb et al. [132, 133] used the SRAM startup state to derive a fingerprint on the microcontroller of an RFID tag. In September 2012 the PUFFIN project, which looks for PUFs in standard computer components, announced[1] that SRAM PUFs can be found in graphics cards of computers. Other interesting new results are the work on extracting a fingerprint from Flash memory [217, 295] and the microprocessor-intrinsic PUF construction of Maiti and Schaumont [186].

In Chapter 4 we introduced the concept of a reconfigurable PUF and we illustrated how this new security primitive can be used to protect the persistent state of a trusted module in external NVM. Practical realizations of this theoretical concept, preferably in silicon, remain an open issue. One possible approach could be to exploit transistor aging. Logically reconfigurable PUFs are a good emulation from a behavioral perspective, but they do not achieve the strong security properties of a physically reconfigurable PUF and they require embedded NVM to store an internal state.

**FPGA Security.** In Chapter 6, we assumed that it is practically infeasible to reverse engineer a PUF from a configuration bitstream. This assumption is motivated by the fact that existing bitstream reversal tools such as Debit [203] and BIL [23] can only reverse engineer bitstreams partially. Full bitstream reversal remains an open problem.

In general, more research is needed to evaluate the security of PUF-based key storage on FPGAs. Reverse engineering of the bitstream in order to determine the type and location of the PUF in the reconfigurable logic is one attack scenario. In his thesis [79] Drimer described how internal state changes can be observed with the FPGA's readback functionality. Perhaps this active readback difference attack can be used to read the PUF response or the derived key. Finally, as mentioned above, the PUF and the fuzzy extractor can also be targeted with a side channel attack.

[1] http://puffin.eu.org/20120927.html

# Bibliography

[1] Actel. The Importance of Design Security: Protecting Your Investment, 2005. http://www.actel.com/documents/Security_PIB.pdf.

[2] Agrawal, D., Archambeault, B., Rao, J. R., and Rohatgi, P. The EM Side-Channel(s). In *4th International Workshop on Cryptographic Hardware and Embedded Systems (CHES 2002)* (2002), B. S. Kaliski Jr., Çetin Kaya Koç, and C. Paar, Eds., vol. 2523 of *Lecture Notes in Computer Science*, Springer, pp. 29–45.

[3] Altera. An FPGA Design Security Solution Using a Secure Memory Device, Oct. 2007. http://www.altera.com/literature/wp/wp-01033.pdf.

[4] Alves, T., and Rudelic, J. ARM Security Solutions and Intel Authenticated Flash, 2007.

[5] Anderson, R. J., and Kuhn, M. G. Tamper Resistance – a Cautionary Note. In *2nd USENIX Workshop on Electronic Commerce (EC 1996)* (1996), USENIX, pp. 1–11.

[6] Anderson, R. J., and Kuhn, M. G. Low Cost Attacks on Tamper Resistant Devices. In *5th International Workshop on Security Protocols* (1998), B. Christianson, B. Crispo, T. M. A. Lomas, and M. Roe, Eds., vol. 1361 of *Lecture Notes in Computer Science*, Springer, pp. 125–136.

[7] Arbaugh, W. A. *Chaining Layered Integrity Checks*. PhD thesis, University of Pennsylvania, 1999.

[8] Arbaugh, W. A., Farber, D. J., and Smith, J. M. A Secure and Reliable Bootstrap Architecture. In *1997 IEEE Symposium on Security and Privacy (S&P 1997)* (1997), IEEE, pp. 65–71.

[9] Arbaugh, W. A., Keromytis, A. D., Farber, D. J., and Smith, J. M. Automated Recovery in a Secure Bootstrap Process. In *Network and Distributed System Security Symposium (NDSS 1998)* (1998), The Internet Society.

[10] Armknecht, F., Maes, R., Sadeghi, A.-R., Sunar, B., and Tuyls, P. Memory Leakage-Resilient Encryption Based on Physically Unclonable Functions. In *Advances in Cryptology – ASIACRYPT 2009* (2009), M. Matsui, Ed., vol. 5912 of *Lecture Notes in Computer Science*, Springer, pp. 685–702.

[11] Asokan, N., Ekberg, J.-E., and Paatero, L. Method, system and computer program product for a trusted counter in an external security element for secure a personal communication device, Feb. 2007. US patent 7178041 B2.

[12] Aucsmith, D. Tamper Resistant Software: An Implementation. In *1st International Workshop on Information Hiding (IH 1996)* (1996), R. J. Anderson, Ed., vol. 1174 of *Lecture Notes in Computer Science*, Springer, pp. 317–333.

[13] Azema, J., and Fayad, G. M-Shield™Mobile Security Technology White Paper. Tech. rep., Texas Instruments, Feb. 2008.

[14] Baetoniu, C., and Sheth, S. FPGA IFF Copy Protection Using Dallas Semiconductor/Maxim DS2432 Secure EEPROMs, May 2010. http://www.xilinx.com/support/documentation/application_notes/xapp780.pdf.

[15] Bar-El, H., Choukri, H., Naccache, D., Tunstall, M., and Whelana, C. The Sorcerer's Apprentice Guide to Fault Attacks. *Proceedings of the IEEE 94*, 2 (2006), 370–382.

[16] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T. L., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. Xen and the Art of Virtualization. In *19th ACM Symposium on Operating Systems Principles (SOSP 2003)* (2003), M. L. Scott and L. L. Peterson, Eds., ACM, pp. 164–177.

[17] Bayon, P., Bossuet, L., Aubert, A., Fischer, V., Poucheret, F., Robisson, B., and Maurine, P. Contactless Electromagnetic Active Attack on Ring Oscillator Based True Random Number Generator. In *3rd International Workshop on Constructive Side-Channel Analysis and Secure Design (COSADE 2012)* (2012), W. Schindler and S. A. Huss, Eds., vol. 7275 of *Lecture Notes in Computer Science*, Springer, pp. 151–166.

[18] Becher, M., Dornseif, M., and Klein, C. N. FireWire: all your memory are belong to us. In *CanSecWest 2005* (2005).

[19] Bellare, M., Canetti, R., and Krawczyk, H. Keying Hash Functions for Message Authentication. In *Advances in Cryptology – CRYPTO 1996* (1996), N. Koblitz, Ed., vol. 1109 of *Lecture Notes in Computer Science*, Springer, pp. 1–15.

[20] Bellare, M., Goldreich, O., and Goldwasser, S. Incremental Cryptography: The Case of Hashing and Signing. In *Advances in Cryptology – CRYPTO 1994* (1994), Y. Desmedt, Ed., vol. 839 of *Lecture Notes in Computer Science*, Springer, pp. 216–233.

[21] Bellare, M., and Namprempre, C. Authenticated Encryption: Relations among Notions and Analysis of the Generic Composition Paradigm. In *Advances in Cryptology – ASIACRYPT 2000* (2000), T. Okamoto, Ed., vol. 1976 of *Lecture Notes in Computer Science*, Springer, pp. 531–545.

[22] Bellare, M., Rogaway, P., and Wagner, D. The EAX Mode of Operation. In *11th International Workshop on Fast Software Encryption (FSE 2004)* (2004), B. K. Roy and W. Meier, Eds., vol. 3017 of *Lecture Notes in Computer Science*, Springer, pp. 389–407.

[23] Benz, F., Seffrin, A., and Huss, S. A. BIL: A Tool-Chain for Bitstream Reverse-Engineering. In *22nd International Conference on Field Programmable Logic and Applications (FPL 2012)* (2012), IEEE, pp. 735–738.

[24] Berger, S., Cáceres, R., Goldman, K. A., Perez, R., Sailer, R., and van Doorn, L. vTPM: Virtualizing the Trusted Platform Module. In *15th USENIX Security Symposium* (2006), USENIX, pp. 305–320.

[25] Biham, E., and Shamir, A. Differential Fault Analysis of Secret Key Cryptosystems. In *Advances in Cryptology – CRYPTO 1997* (1997), B. S. Kaliski Jr., Ed., vol. 1294 of *Lecture Notes in Computer Science*, Springer, pp. 513–525.

[26] Billet, O., Gilbert, H., and Ech-Chatbi, C. Cryptanalysis of a White Box AES Implementation. In *11th International Workshop on Selected Areas in Cryptography (SAC 2004)* (2004), H. Handschuh and M. A. Hasan, Eds., vol. 3357 of *Lecture Notes in Computer Science*, Springer, pp. 227–240.

[27] Black, J. Authenticated encryption. In *Encyclopedia of Cryptography and Security*, H. C. A. van Tilborg, Ed. Springer, 2005.

[28] Black, J., Halevi, S., Krawczyk, H., Krovetz, T., and Rogaway, P. UMAC: Fast and Secure Message Authentication. In *Advances in Cryptology – CRYPTO 1999* (1999), M. J. Wiener, Ed., vol. 1666 of *Lecture Notes in Computer Science*, Springer, pp. 216–233.

[29] Black, J., and Rogaway, P. CBC MACs for Arbitrary-Length Messages: The Three-Key Constructions. In *Advances in Cryptology – CRYPTO 2000* (2000), M. Bellare, Ed., vol. 1880 of *Lecture Notes in Computer Science*, Springer, pp. 197–215.

[30] Black, J., Rogaway, P., and Shrimpton, T. Black-Box Analysis of the Block-Cipher-Based Hash-Function Constructions from PGV. In *Advances in Cryptology – CRYPTO 2002* (2002), M. Yung, Ed., vol. 2442 of *Lecture Notes in Computer Science*, Springer, pp. 320–335.

[31] Boileau, A. Hit by a Bus: Physical Access Attacks with Firewire. In *Ruxcon 2006* (2006).

[32] Bond, M. Attacks on Cryptoprocessor Transaction Sets. In *3rd International Workshop on Cryptographic Hardware and Embedded Systems (CHES 2001)* (2001), Çetin Kaya Koç, D. Naccache, and C. Paar, Eds., vol. 2162 of *Lecture Notes in Computer Science*, Springer, pp. 220–234.

[33] Boneh, D., DeMillo, R. A., and Lipton, R. J. On the Importance of Checking Cryptographic Protocols for Faults (Extended Abstract). In *Advances in Cryptology – EUROCRYPT 1997* (1997), W. Fumy, Ed., vol. 1233 of *Lecture Notes in Computer Science*, Springer, pp. 37–51.

[34] Borghoff, J., Canteaut, A., Güneysu, T., Kavun, E. B., Knezevic, M., Knudsen, L. R., Leander, G., Nikov, V., Paar, C., Rechberger, C., Rombouts, P., Thomsen, S. S., and Yalçin, T. PRINCE – A Low-Latency Block Cipher for Pervasive Computing Applications. In *Advances in Cryptology – ASIACRYPT 2012* (2012), X. Wang and K. Sako, Eds., vol. 7658 of *Lecture Notes in Computer Science*, Springer, pp. 208–225.

[35] Bösch, C., Guajardo, J., Sadeghi, A.-R., Shokrollahi, J., and Tuyls, P. Efficient Helper Data Key Extractor on FPGAs. In *10th International Workshop on Cryptographic Hardware and Embedded Systems (CHES 2008)* (2008), E. Oswald and P. Rohatgi, Eds., vol. 5154 of *Lecture Notes in Computer Science*, Springer, pp. 181–197.

[36] Boyen, X. Robust and Reusable Fuzzy Extractors. In *Security with Noisy Data: On Private Biometrics, Secure Key Storage and Anti-Counterfeiting.* Springer, 2007, ch. 6, pp. 101–112.

[37] Bratus, S., D'Cunha, N., Sparks, E., and Smith, S. W. TOCTOU, Traps, and Trusted Computing. In *1st International Conference on Trusted Computing and Trust in Information Technologies (Trust 2008)* (2008), P. Lipp, A.-R. Sadeghi, and K.-M. Koch, Eds., vol. 4968 of *Lecture Notes in Computer Science*, Springer, pp. 14–32.

[38] Brickell, E. F., Camenisch, J., and Chen, L. Direct Anonymous Attestation. In *11th ACM Conference on Computer and Communications Security (CCS 2004)* (2004), V. Atluri, B. Pfitzmann, and P. D. McDaniel, Eds., ACM, pp. 132–145.

[39] Bruschi, D., Cavallaro, L., Lanzi, A., and Monga, M. Replay Attack in TCG Specification and Solution. In *21th Annual Computer Security Applications Conference (ACSAC 2005)* (2005), IEEE, pp. 127–137.

[40] BSI. Federal Government's Comments on the TCG and NGSCB in the Field of Trusted Computing. https://www.bsi.bund.de/cae/servlet/contentblob/483110/publicationFile/30569/StellungnahmeTCG1_2a_e_pdf.pdf.

[41] Buysschaert, P., De Mulder, E., Örs, S. B., Delmotte, P., Preneel, B., Vandenbosch, G., and Verbauwhede, I. Electromagnetic Analysis Attack on an FPGA Implementation of an Elliptic Curve Cryptosystem. In *The International Conference on "Computer as a Tool" (EUROCON 2005)* (2005), vol. 2, IEEE, pp. 1879–1882.

[42] Camenisch, J. Better Privacy for Trusted Computing Platforms: (Extended Abstract). In *9th European Symposium on Research In Computer Security (ESORICS 2004)* (2004), P. Samarati, P. Y. A. Ryan, D. Gollmann, and R. Molva, Eds., vol. 3193 of *Lecture Notes in Computer Science*, Springer, pp. 73–88.

[43] Carlier, V., Chabanne, H., Dottax, E., and Pelletier, H. Electromagnetic Side Channels of an FPGA Implementation of AES. Cryptology ePrint Archive, Report 2004/145, 2004.

[44] Carrier, B. D., and Grand, J. A hardware-based memory acquisition procedure for digital investigations. *Digital Investigation 1*, 1 (2004), 50–60.

[45] Castillo, E., Parrilla, L., García, A., Lloris, A., and Meyer-Baese, U. Intellectual Property Protection of HDL IP Cores Through Automated Signature Hosting. In *17th International Conference on Field*

*Programmable Logic and Applications (FPL 2007)* (Aug. 2007), IEEE, pp. 183–188.

[46] Ceccato, M., Preda, M. D., Nagra, J., Collberg, C., and Tonella, P. Barrier Slicing for Remote Software Trusting. In *7th IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2007)* (2007), IEEE, pp. 27–36.

[47] Ceccato, M., Preda, M. D., Nagra, J., Collberg, C., and Tonella, P. Trading-off Security and Performance in Barrier Slicing for Remote Software Entrusting. *Automated Software Engineering 16*, 2 (2009), 235–261.

[48] Ceccato, M., and Tonella, P. CodeBender: Remote Software Protection Using Orthogonal Replacement. *IEEE Software 28*, 2 (2011), 28–34.

[49] Ceccato, M., Tonella, P., Preda, M. D., and Majumdar, A. Remote software protection by orthogonal client replacement. In *24th ACM Symposium on Applied Computing (SAC 2009)* (2009), S. Y. Shin and S. Ossowski, Eds., ACM, pp. 448–455.

[50] Cesena, E., Ramunno, G., Sassu, R., Vernizzi, D., and Lioy, A. On Scalability of Remote Attestation. In *6th ACM Workshop on Scalable Trusted Computing (STC 2011)* (2011), Y. Chen, S. Xu, A.-R. Sadeghi, and X. Zhang, Eds., ACM, pp. 25–30.

[51] Challener, D., Yoder, K., Catherman, R., Safford, D., and van Doorn, L. *A Practical Guide to Trusted Computing.* IBM Press, 2008.

[52] Chang, H., and Atallah, M. J. Protecting Software Code by Guards. In *1st ACM Workshop on Security and Privacy in Digital Rights Management (DRM 2001)* (2002), vol. 2320 of *Lecture Notes in Computer Science*, Springer, pp. 160–175.

[53] Chari, S., Rao, J. R., and Rohatgi, P. Template Attacks. In *4th International Workshop on Cryptographic Hardware and Embedded Systems (CHES 2002)* (2003), B. S. Kaliski Jr., Çetin Kaya Koç, and C. Paar, Eds., vol. 2523 of *Lecture Notes in Computer Science*, Springer, pp. 13–28.

[54] Chen, L., and Ryan, M. D. Offline dictionary attack on TCG TPM weak authorisation data, and solution. In *1st International Conference on Future of Trust in Computing* (2008), Vieweg & Teubner, pp. 193–196.

[55] Chen, L., and Ryan, M. D. Attack, Solution and Verification for Shared Authorisation Data in TCG TPM. In *6th International Workshop on Formal Aspects in Security and Trust (FAST 2009)* (2010), P. Degano and J. D. Guttman, Eds., vol. 5983 of *Lecture Notes in Computer Science*, Springer, pp. 201–216.

[56] Chen, Z. *Java Card™Technology for Smart Cards: Architecture and Programmer's Guide.* Addison-Wesley Professional, 2000.

[57] Chevallier-Mames, B., Naccache, D., Paillier, P., and Pointcheval, D. How to Disembed a Program? In *6th International Workshop on Cryptographic Hardware and Embedded Systems (CHES 2004)* (2004), M. Joye and J.-J. Quisquater, Eds., vol. 3156 of *Lecture Notes in Computer Science*, Springer, pp. 441–454.

[58] Chevallier-Mames, B., Naccache, D., Paillier, P., and Pointcheval, D. How to Disembed a Program? Cryptology ePrint Archive, Report 2004/138, 2004.

[59] Chiu, Y.-H., Liao, Y.-B., Chiang, M.-H., Lin, C.-L., Hsu, W.-C., Chiang, P.-C., Hsu, Y.-Y., Liu, W.-H., Sheu, S.-S., Su, K.-L., Kao, M.-J., and Tsai, M.-J. Impact of Resistance Drift on Multilevel PCM Design. In *2010 IEEE International Conference on IC Design and Technology (ICICDT 2010)* (2010), pp. 20–23.

[60] Chow, S., Eisen, P. A., Johnson, H., and van Oorschot, P. C. A White-Box DES Implementation for DRM Applications. In *2nd ACM Workshop on Security and Privacy in Digital Rights Management (DRM 2002)* (2003), J. Feigenbaum, Ed., vol. 2696 of *Lecture Notes in Computer Science*, Springer, pp. 1–15.

[61] Chow, S., Eisen, P. A., Johnson, H., and van Oorschot, P. C. White-Box Cryptography and an AES Implementation. In *9th Annual International Workshop on Selected Areas in Cryptography (SAC 2002)* (2003), K. Nyberg and H. M. Heys, Eds., vol. 2595 of *Lecture Notes in Computer Science*, Springer, pp. 250–270.

[62] Clarke, D. E., Suh, G. E., Gassend, B., Sudan, A., van Dijk, M., and Devadas, S. Towards Constant Bandwidth Overhead Integrity Checking of Untrusted Data. In *2005 IEEE Symposium on Security and Privacy (S&P 2005)* (2005), IEEE, pp. 139–153.

[63] Collberg, C. S., and Thomborson, C. Watermarking, Tamper-Proofing, and Obfuscation – Tools for Software Protection. Tech. Rep. TR00-03, Department of Computer Science, University of Arizona, Feb. 2000.

[64] Collberg, C. S., Thomborson, C., and Low, D. A Taxonomy of Obfuscating Transformations. Tech. Rep. 148, Department of Computer Science, University of Auckland, July 1997.

[65] Costan, V., Sarmenta, L. F. G., van Dijk, M., and Devadas, S. The Trusted Execution Module: Commodity General-Purpose Trusted Computing. In *8th Smart Card Research and Advanced Applications IFIP Conference (CARDIS 2008)* (2008), G. Grimaud and F.-X. Standaert, Eds., vol. 5189 of *Lecture Notes in Computer Science*, Springer, pp. 133–148.

[66] Costan, V. M. A Commodity Trusted Computing Module. Master's thesis, Massachusetts Institute of Technology, May 2008.

[67] De Mulder, E., Örs, S. B., Preneel, B., and Verbauwhede, I. Differential Electromagnetic Attack on an FPGA. In *World Automation Congress 2006* (2006), pp. 1–6.

[68] De Mulder, E., Örs, S. B., Preneel, B., and Verbauwhede, I. Differential power and electromagnetic attacks on a FPGA implementation of elliptic curve cryptosystems. *Computers & Electrical Engineering 33*, 5-6 (2007), 367–382.

[69] De Vries, A., and Ma, Y. A logical approach to NVM integration in SOC design. *EDN Magazine*, 2 (Jan. 2007).

[70] Defrawy, K. E., Francillon, A., Perito, D., and Tsudik, G. SMART: Secure and Minimal Architecture for (Establishing a Dynamic) Root of Trust. In *19th Annual Network and Distributed System Security Symposium (NDSS 2012)* (2012), The Internet Society.

[71] Devine, C., and Vissian, G. PCI bus based operating system attack and protections. In *EUSecWest 2009* (2009).

[72] Dietrich, K. An Integrated Architecture for Trusted Computing for Java enabled Embedded Devices. In *2nd ACM Workshop on Scalable Trusted Computing (STC 2007)* (2007), P. Ning, V. Atluri, S. Xu, and M. Yung, Eds., ACM, pp. 2–6.

[73] Dietrich, K., and Winter, J. Secure Boot Revisited. In *9th International Conference for Young Computer Scientists (ICYCS 2008)* (2008), IEEE, pp. 2360–2365.

[74] Dietrich, K., and Winter, J. Implementation Aspects of Mobile and Embedded Trusted Computing. In *2nd International Conference on Trusted Computing (Trust 2009)* (2009), L. Chen, C. J. Mitchell, and

A. Martin, Eds., vol. 5471 of *Lecture Notes in Computer Science*, Springer, pp. 29–44.

[75] Dietrich, K., and Winter, J. Towards Customizable, Application Specific Mobile Trusted Modules. In *5th ACM Workshop on Scalable Trusted Computing (STC 2010)* (2010), S. Xu, N. Asokan, and A.-R. Sadeghi, Eds., ACM, pp. 31–40.

[76] Dodis, Y., Reyzin, L., and Smith, A. Fuzzy Extractors: How to Generate Strong Keys from Biometrics and Other Noisy Data. In *Advances in Cryptology – EUROCRYPT 2004* (2004), C. Cachin and J. Camenisch, Eds., vol. 3027 of *Lecture Notes in Computer Science*, Springer, pp. 523–540.

[77] Dornseif, M. 0wned by an iPod – hacking by Firewire. In *PacSec 2004* (2004).

[78] Drimer, S. Authentication of FPGA Bitstreams: Why and How. In *3rd International Workshop on Applied Reconfigurable Computing (ARC 2007)* (2007), P. C. Diniz, E. Marques, K. Bertels, M. M. Fernandes, and J. ao M. P. Cardoso, Eds., vol. 4419 of *Lecture Notes in Computer Science*, Springer, pp. 73–84.

[79] Drimer, S. *Security for volatile FPGAs.* PhD thesis, University of Cambridge, Nov. 2009.

[80] Duflot, L., Etiemble, D., and Grumelard, O. Using CPU System Management Mode to Circumvent Operating System Security Functions. In *CanSecWest 2006* (2006).

[81] Duflot, L., Grumelard, O., Levillain, O., and Morin, B. Getting into the SMRAM: SMM Reloaded. In *CanSecWest 2009* (2009).

[82] Duflot, L., Levillain, O., Levillain, O., and Morin, B. ACPI and SMI handlers: some limits to trusted computing. *Journal in Computer Virology* (2009).

[83] Duflot, L., Levillain, O., Morin, B., and Grumela, O. System Management Mode Design and Security Issues. In *IT-Defense 2010* (2010).

[84] Duflot, L., Perez, Y.-A., Valadon, G., and Levillain, O. Can you still trust your network card? In *CanSecWest 2010* (2010).

[85] Dvir, O., Herlihy, M., and Shavit, N. Virtual Leashing: Internet-Based Software Piracy Protection. In *25th International Conference on Distributed Computing Systems (ICDCS 2005)* (2005), IEEE, pp. 283–292.

[86] Dyer, J. G., Lindemann, M., Perez, R., Sailer, R., van Doorn, L., Smith, S. W., and Weingart, S. Building the IBM 4758 Secure Coprocessor. *IEEE Computer 34*, 10 (2001), 57–66.

[87] Eisenbarth, T., Güneysu, T., Paar, C., Sadeghi, A.-R., Schellekens, D., and Wolf, M. Reconfigurable Trusted Computing in Hardware. In *2nd ACM Workshop on Scalable Trusted Computing (STC 2007)* (2007), P. Ning, V. Atluri, S. Xu, and M. Yung, Eds., ACM, pp. 15–20.

[88] Eisenbarth, T., Kasper, T., Moradi, A., Paar, C., Salmasizadeh, M., and Shalmani, M. T. M. On the Power of Power Analysis in the Real World: A Complete Break of the KeeLoqCode Hopping Scheme. In *Advances in Cryptology – CRYPTO 2008* (2008), D. Wagner, Ed., vol. 5157 of *Lecture Notes in Computer Science*, Springer, pp. 203–220.

[89] Ekberg, J.-E., and Asokan, N. External Authenticated Non-volatile Memory with Lifecycle Management for State Protection in Trusted Computing. In *1st International Conference on Trusted Systems (INTRUST 2009)* (2010), L. Chen and M. Yung, Eds., vol. 6163 of *Lecture Notes in Computer Science*, Springer, pp. 16–38.

[90] Ekberg, J.-E., and Bugiel, S. Trust in a Small Package: Minimized MRTM Software Implementation for Mobile Secure Environments. In *4th ACM Workshop on Scalable Trusted Computing (STC 2009)* (2009), S. Xu, N. Asokan, C. Nita-Rotaru, and J.-P. Seifert, Eds., ACM, pp. 9–18.

[91] Ekberg, J.-E., and Kylänpää, M. Mobile Trusted Module (MTM) – an introduction, Nov. 2007. http://research.nokia.com/files/NRCTR2007015.pdf.

[92] Elbaz, R., Champagne, D., Gebotys, C. H., Lee, R. B., Potlapally, N. R., and Torres, L. Hardware Mechanisms for Memory Authentication: A Survey of Existing Techniques and Engines. In *Transactions on Computational Science IV – Special Issue on Security in Computing*, M. L. Gavrilova, C. J. K. Tan, and E. D. Moreno, Eds., vol. 5430 of *Lecture Notes in Computer Science*. Springer, 2009, pp. 1–22.

[93] Elbaz, R., Champagne, D., Lee, R. B., Torres, L., Sassatelli, G., and Guillemin, P. TEC-Tree: A Low-Cost, Parallelizable Tree for Efficient Defense Against Memory Replay Attacks. In *9th International Workshop on Cryptographic Hardware and Embedded Systems (CHES 2007)* (2007), P. Paillier and I. Verbauwhede, Eds., vol. 4727 of *Lecture Notes in Computer Science*, Springer, pp. 289–302.

[94] Elbaz, R., Torres, L., Sassatelli, G., Guillemin, P., Bardouillet, M., and Martinez, A. Block-Level Added Redundancy Explicit Authentication for Parallelized Encryption and Integrity Checking of Processor-Memory Transactions. In *Transactions on Computational Science X – Special Issue on Security in Computing, Part I*, M. L. Gavrilova, C. J. K. Tan, and E. D. Moreno, Eds., vol. 6340 of *Lecture Notes in Computer Science*. Springer, 2010, pp. 231–260.

[95] England, P. Practical Techniques for Operating System Attestation. In *1st International Conference on Trusted Computing and Trust in Information Technologies (Trust 2008)* (2008), P. Lipp, A.-R. Sadeghi, and K.-M. Koch, Eds., vol. 4968 of *Lecture Notes in Computer Science*, Springer, pp. 1–13.

[96] England, P., Lampson, B. W., Manferdelli, J., Peinado, M., and Willman, B. A Trusted Open Platform. *IEEE Computer 36*, 7 (2003), 55–62.

[97] Feller, T., Malipatlolla, S., Meister, D., and Huss, S. A. TinyTPM: A Lightweight Module aimed to IP Protection and Trusted Embedded Platforms. In *2011 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST 2011)* (2011), IEEE, pp. 6–11.

[98] Gandolfi, K., Mourtel, C., and Olivier, F. Electromagnetic Analysis: Concrete Results. In *3rd International Workshop on Cryptographic Hardware and Embedded Systems (CHES 2001)* (2001), Çetin Kaya Koç, D. Naccache, and C. Paar, Eds., vol. 2162 of *Lecture Notes in Computer Science*, Springer, pp. 251–261.

[99] Garay, J. A., and Huelsbergen, L. Software Integrity Protection Using Timed Executable Agents. In *1st ACM Symposium on Information, Computer and Communications Security (ASIACCS 2006)* (2006), F.-C. Lin, D.-T. Lee, B.-S. Lin, S. Shieh, and S. Jajodia, Eds., ACM, pp. 189–200.

[100] Garcia, F. D., de Koning Gans, G., Muijrers, R., van Rossum, P., Verdult, R., Schreur, R. W., and Jacobs, B. Dismantling MIFARE Classic. In *13th European Symposium on Research in Computer Security (ESORICS 2008)* (2008), S. Jajodia and J. López, Eds., vol. 5283 of *Lecture Notes in Computer Science*, Springer, pp. 97–114.

[101] Garfinkel, T., Pfaff, B., Chow, J., Rosenblum, M., and Boneh, D. Terra: A Virtual Machine-based Platform for Trusted Computing. In *19th ACM Symposium on Operating Systems Principles (SOSP 2003)* (2003), pp. 193–206.

[102] Garfinkel, T., Rosenblum, M., and Boneh, D. Flexible OS Support and Applications for Trusted Computing. In *9th Workshop on Hot Topics in Operating Systems (HotOS 2003)* (2003), M. B. Jones, Ed., USENIX, pp. 145–150.

[103] Gassend, B. Physical Random Functions. Master's thesis, Massachusetts Institute of Technology, 2003.

[104] Gassend, B., Clarke, D. E., van Dijk, M., and Devadas, S. Controlled Physical Random Functions. In *18th Annual Computer Security Applications Conference (ACSAC 2002)* (2002), IEEE, pp. 149–160.

[105] Gassend, B., Clarke, D. E., van Dijk, M., and Devadas, S. Silicon Physical Unknown Functions. In *9th ACM Conference on Computer and Communications Security (CCS 2002)* (2002), V. Atluri, Ed., ACM, pp. 148–160.

[106] Gassend, B., Lim, D., Clarke, D., van Dijk, M., and Devadas, S. Identification and authentication of integrated circuits. *Concurrency – Practice and Experience 16*, 11 (2004), 1077–1098.

[107] Gassend, B., Suh, G. E., Clarke, D., van Dijk, M., and Devadas, S. Caches and Hash Trees for Efficient Memory Integrity Verification. In *9th International Symposium on High-Performance Computer Architecture (HPCA 2003)* (2003), IEEE, pp. 295–306.

[108] Gasser, M., Goldstein, A., Kaufman, C., and Lampson, B. The Digital Distributed System Security Architecture. In *12th National Computer Security Conference (NCSC 1989)* (1989), pp. 305–319.

[109] Giffin, J. T., Christodorescu, M., and Kruger, L. Strengthening Software Self-Checksumming via Self-Modifying Code. In *21st Annual Computer Security Applications Conference (ACSAC 2005)* (2005), IEEE, pp. 23–32.

[110] Goodin, D. Ex-Army man cracks popular security chip: How to open Infineon's Trusted Platform Module, Feb. 2010. http://www.theregister.co.uk/2010/02/17/infineon_tpm_crack/.

[111] Goodman, J. W. Statistical properties of laser speckle patterns. In *Laser Speckle and Related Phenomena*, J. W. Dainty, Ed. Springer, 1975.

[112] Goubin, L., Masereel, J.-M., and Quisquater, M. Cryptanalysis of White Box DES Implementations. In *14th International Workshop on Selected Areas in Cryptography (SAC 2007)* (2007), C. M. Adams, A. Miri, and M. J. Wiener, Eds., vol. 4876 of *Lecture Notes in Computer Science*, Springer, pp. 278–295.

[113] Grawrock, D. *The Intel Safer Computing Initiative: Building Blocks for Trusted Computing.* Intel Press, 2006.

[114] Grawrock, D. *Dynamics of a Trusted Platform: A building block approach.* Intel Press, 2009.

[115] Gross, M. Vertrauenswürdiges Booten als Grundlage authentischer Basissysteme. In *GI-Fachtagung Verläßliche Informationssysteme (VIS'91)* (1991), A. Pfitzmann and E. Raubold, Eds., vol. 271 of *Informatik-Fachberichte*, Springer, pp. 190–207.

[116] Großschädl, J., Vejda, T., and Page, D. Reassessing the TCG Specifications for Trusted Computing in Mobile and Embedded Systems. In *IEEE International Workshop on Hardware-Oriented Security and Trust (HOST 2008)* (2008), M. Tehranipoor and J. Plusquellic, Eds., IEEE, pp. 84–90.

[117] Guajardo, J., Kumar, S. S., Schrijen, G.-J., and Tuyls, P. FPGA Intrinsic PUFs and Their Use for IP Protection. In *9th International Workshop on Cryptographic Hardware and Embedded Systems (CHES 2007)* (September 10-13, 2007), P. Paillier and I. Verbauwhede, Eds., vol. 4727 of *Lecture Notes in Computer Science*, Springer, pp. 63–80.

[118] Guajardo, J., Kumar, S. S., Schrijen, G.-J., and Tuyls, P. Physical Unclonable Functions and Public Key Crypto for FPGA IP Protection. In *17th International Conference on Field Programmable Logic and Applications (FPL 2007)* (August 27-30, 2007), IEEE, pp. 189–195.

[119] Guajardo, J., Kumar, S. S., Schrijen, G.-J., and Tuyls, P. Brand and IP Protection with Physical Unclonable Functions. In *2008 International Symposium on Circuits and Systems (ISCAS 2008)* (2008), IEEE, pp. 3186–3189.

[120] Gürgens, S., Rudolph, C., Scheuermann, D., Atts, M., and Plaga, R. Security Evaluation of Scenarios Based on the TCG's TPM Specification. In *12th European Symposium On Research In Computer Security (ESORICS 2007)* (2007), J. Biskup and J. Lopez, Eds., vol. 4734 of *Lecture Notes in Computer Science*, Springer, pp. 438–453.

[121] Haldar, V., Chandra, D., and Franz, M. Semantic Remote Attestation – Virtual Machine Directed Approach to Trusted Computing. In *3rd Virtual Machine Research and Technology Symposium (VM 2004)* (2004), USENIX, pp. 29–41.

[122] Halderman, J. A., Schoen, S. D., Heninger, N., Clarkson, W., Paul, W., Calandrino, J. A., Feldman, A. J., Appelbaum, J., and Felten, E. W. Lest We Remember: Cold Boot Attacks on Encryption Keys. In *17th USENIX Security Symposium* (2008), pp. 45–60.

[123] Hall, W. E., and Jutla, C. S. Parallelizable Authentication Trees. In *12th International Workshop on Selected Areas in Cryptography (SAC 2005)* (2006), B. Preneel and S. E. Tavares, Eds., vol. 3897 of *Lecture Notes in Computer Science*, Springer, pp. 95–109.

[124] Hammouri, G., Öztürk, E., Birand, B., and Sunar, B. Unclonable Lightweight Authentication Scheme. In *10th International Conference on Information and Communications Security (ICICS 2008)* (2008), L. Chen, M. D. Ryan, and G. Wang, Eds., vol. 5308 of *Lecture Notes in Computer Science*, Springer, pp. 33–48.

[125] Hammouri, G., and Sunar, B. PUF-HB: A Tamper-Resilient HB Based Authentication Protocol. In *6th International Conference on Applied Cryptography and Network Security (ACNS 2008)* (2008), S. M. Bellovin, R. Gennaro, A. D. Keromytis, and M. Yung, Eds., vol. 5037 of *Lecture Notes in Computer Science*, Springer, pp. 346–365.

[126] Handschuh, H., Paillier, P., and Stern, J. Probing Attacks on Tamper-Resistant Devices. In *1st International Workshop on Cryptographic Hardware and Embedded Systems (CHES 1999)* (1999), Çetin Kaya Koç and C. Paar, Eds., vol. 1717 of *Lecture Notes in Computer Science*, Springer, pp. 303–315.

[127] Handschuh, H., and Trichina, E. Securing Flash Technology. In *4th International Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC 2007)* (2007), L. Breveglieri, S. Gueron, I. Koren, D. Naccache, and J.-P. Seifert, Eds., IEEE, pp. 3–17.

[128] Handschuh, H., and Trichina, E. Securing Flash Technology: How Does It Look From Inside? In *ISSE 2008 – Securing Electronic Busines Processes, Highlights of the Information Security Solutions Europe 2008 Conference, 7-9 October 2008, Madrid, Spain* (2009), N. Pohlmann, H. Reimer, and W. Schneider, Eds., Vieweg+Teubner, pp. 380–389.

[129] Härtig, H., Kowalski, O. C., and Kühnhauser, W. E. The BirliX Security Architecture. *Journal of Computer Security 2* (1993), 5–22.

[130] Heiser, G., Elphinstone, K., Kuz, I., Klein, G., and Petters, S. M. Towards Trustworthy Computing Systems: Taking Microkernels to the Next Level. *ACM SIGOPS Operating Systems Review 41*, 4 (2007), 3–11.

[131] Heiser, G., and Leslie, B. The OKL4 Microvisor: Convergence Point of Microkernels and Hypervisors. In *1st ACM SIGCOMM Asia-Pacific Workshop on Systems (ApSys 2010)* (2010), C. A. Thekkath, R. Kotla, and L. Zhou, Eds., ACM, pp. 19–24.

[132] Holcomb, D. E., Burleson, W. P., and Fu, K. Initial SRAM State as a Fingerprint and Source of True Random Numbers for RFID Tags. In *Conference on RFID Security 2007* (July 2007).

[133] Holcomb, D. E., Burleson, W. P., and Fu, K. Power-up SRAM State as an Identifying Fingerprint and Source of True Random Numbers. *IEEE Transactions on Computers 58*, 9 (2009), 1198–1210.

[134] Hu, Y., Hammouri, G., and Sunar, B. A Fast Real-time Memory Authentication Protocol. In *3rd ACM Workshop on Scalable Trusted Computing (STC 2008)* (2008), S. Xu, C. Nita-Rotaru, and J.-P. Seifert, Eds., ACM, pp. 31–40.

[135] Huang, A. *Hacking the Xbox: An Introduction to Reverse Engineering.* No Starch Press, July 2003.

[136] Huang, A. Keeping Secrets in Hardware: The Microsoft Xbox Case Study. In *4th International Workshop on Cryptographic Hardware and Embedded Systems (CHES 2002)* (2003), B. S. Kaliski Jr., Çetin Kaya Koç, and C. Paar, Eds., vol. 2523 of *Lecture Notes in Computer Science*, Springer, pp. 213–227.

[137] Intel. Low Pin Count (LPC) Interface Specification, Aug. 2002. http://www.intel.com/design/chipsets/industry/25128901.pdf.

[138] Iwata, T., and Kurosawa, K. OMAC: One-Key CBC MAC. In *10th International Workshop on Fast Software Encryption (FSE 2003)* (2003), T. Johansson, Ed., vol. 2887 of *Lecture Notes in Computer Science*, Springer, pp. 129–153.

[139] Jacob, M., Boneh, D., and Felten, E. W. Attacking an Obfuscated Cipher by Injecting Faults. In *2nd ACM Workshop on Security and Privacy in Digital Rights Management (DRM 2002)* (2003), J. Feigenbaum, Ed., vol. 2696 of *Lecture Notes in Computer Science*, Springer, pp. 16–31.

[140] Jain, A. K., Yuan, L., Pari, P. R., and Qu, G. Zero overhead watermarking technique for FPGA designs. In *13th ACM Great Lakes Symposium on VLSI* (2003), ACM, pp. 147–152.

[141] Jonsson, J. On the Security of CTR + CBC-MAC. In *9th Annual International Workshop on Selected Areas in Cryptography (SAC 2002)*

(2003), K. Nyberg and H. M. Heys, Eds., vol. 2595 of *Lecture Notes in Computer Science*, Springer, pp. 76–93.

[142] Jutla, C. S. Encryption Modes with Almost Free Message Integrity. In *Advances in Cryptology – EUROCRYPT 2001* (2001), B. Pfitzmann, Ed., vol. 2045 of *Lecture Notes in Computer Science*, Springer, pp. 529–544.

[143] Kahng, A. B., Lach, J., Mangione-Smith, W. H., Mantik, S., Markov, I. L., Potkonjak, M., Tucker, P., Wang, H., and Wolfe, G. Watermarking Techniques for Intellectual Property Protection. In *35th Conference on Design Automation (DAC 1998)* (1998), pp. 776–781.

[144] Kahng, A. B., Lach, J., Mangione-Smith, W. H., Mantik, S., Markov, I. L., Potkonjak, M., Tucker, P., Wang, H., and Wolfe, G. Constraint-Based Watermarking Techniques for Design IP Protection. *IEEE Transactions on CAD of Integrated Circuits and Systems 20*, 10 (2001), 1236–1252.

[145] Kahng, A. B., Mantik, S., Markov, I. L., Potkonjak, M., Tucker, P., Wang, H., and Wolfe, G. Robust IP Watermarking Methodologies for Physical Design. In *35th Conference on Design Automation (DAC 1998)* (1998), pp. 782–787.

[146] Katzenbeisser, S., Koçabas, U., van der Leest, V., Sadeghi, A.-R., Schrijen, G.-J., Schröder, H., and Wachsmann, C. Recyclable PUFs: Logically Reconfigurable PUFs. In *13th International Workshop on Cryptographic Hardware and Embedded Systems (CHES 2011)* (2011), B. Preneel and T. Takagi, Eds., vol. 6917 of *Lecture Notes in Computer Science*, Springer, pp. 374–389.

[147] Kauer, B. OSLO: Improving the Security of Trusted Computing. In *16th USENIX Security Symposium* (2007), USENIX, pp. 229–237.

[148] Kennell, R., and Jamieson, L. H. Establishing the Genuinity of Remote Computer Systems. In *12th USENIX Security Symposium* (2003), USENIX, pp. 295–308.

[149] Kerins, T., and Kursawe, K. A Cautionary Note on Weak Implementations of Block Ciphers. In *1st Benelux Workshop on Information and System Security (WISSec 2006)* (Nov. 2006).

[150] Kim, M., Ju, H., Kim, Y., Park, J., and Park, Y. Design and Implementation of Mobile Trusted Module for Trusted Mobile Computing. *IEEE Transactions on Consumer Electronics 56*, 1 (Feb. 2010), 134–140.

[151] Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., and Winwood, S. seL4: Formal Verification of an OS Kernel. In *22nd ACM Symposium on Operating Systems Principles (SOSP 2009)* (2009), J. N. Matthews and T. E. Anderson, Eds., ACM, pp. 207–220.

[152] Klug, B. Motorola Droid X: Thoroughly Reviewed, July 2010. http://www.anandtech.com/show/3826/motorola-droid-x-thoroughly-reviewed/.

[153] Knezevic, M., Nikov, V., and Rombouts, P. Low-Latency Encryption – Is "Lightweight = Light + Wait"? In *14th International Workshop on Cryptographic Hardware and Embedded Systems (CHES 2012)* (2012), E. Prouff and P. Schaumont, Eds., vol. 7428 of *Lecture Notes in Computer Science*, Springer, pp. 426–446.

[154] Kocher, P. C. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Advances in Cryptology – CRYPTO 1996* (1996), N. Koblitz, Ed., vol. 1109 of *Lecture Notes in Computer Science*, Springer, pp. 104–113.

[155] Kocher, P. C., Jaffe, J., and Jun, B. Differential Power Analysis. In *Advances in Cryptology – CRYPTO 1999* (1999), M. J. Wiener, Ed., vol. 1666 of *Lecture Notes in Computer Science*, Springer, pp. 388–397.

[156] Kohno, T., Viega, J., and Whiting, D. CWC: A High-Performance Conventional Authenticated Encryption Mode. In *11th International Workshop on Fast Software Encryption (FSE 2004)* (2004), B. K. Roy and W. Meier, Eds., vol. 3017 of *Lecture Notes in Computer Science*, Springer, pp. 408–426.

[157] Kömmerling, O., and Kuhn, M. G. Design Principles for Tamper-Resistant Smartcard Processors. In *1st USENIX Workshop on Smartcard Technology (Smartcard '99)* (1999), USENIX, pp. 9–20.

[158] Kostiainen, K., Reshetova, E., Ekberg, J.-E., and Asokan, N. Old, New, Borrowed, Blue – A Perspective on the Evolution of Mobile Platform Security Architectures. In *1st ACM Conference on Data and Application Security and Privacy (CODASPY 2011)* (2011), ACM, pp. 13–24.

[159] Kovah, X., Kallenberg, C., Weathers, C., Herzog, A., Albin, M., and Butterworth, J. New Results for Timing-Based Attestation. In *2012 IEEE Symposium on Security and Privacy (S&P 2012)* (2012), IEEE, pp. 239–253.

[160] Kuhlmann, D., Landfermann, R., Ramasamy, H. V., Schunter, M., Ramunno, G., and Vernizzi, D. An Open Trusted Computing Architecture – Secure Virtual Machines Enabling User-Defined Policy Enforcement. http://www.opentc.net, June 2006.

[161] Kühn, U., Selhorst, M., and Stüble, C. Realizing Property-Based Attestation and Sealing with Commonly Available Hard- and Software. In *2nd ACM Workshop on Scalable Trusted Computing (STC 2007)* (2007), P. Ning, V. Atluri, S. Xu, and M. Yung, Eds., ACM, pp. 50–57.

[162] Kumar, S. S., Guajardo, J., Maes, R., Schrijen, G.-J., and Tuyls, P. The Butterfly PUF: Protecting IP on every FPGA. In *IEEE International Workshop on Hardware-Oriented Security and Trust (HOST 2008)* (2008), M. Tehranipoor and J. Plusquellic, Eds., IEEE, pp. 67–70.

[163] Kursawe, K., Sadeghi, A.-R., Schellekens, D., Škorić, B., and Tuyls, P. Reconfigurable Physical Unclonable Functions – Enabling Technology for Tamper-Resistant Storage. In *2009 IEEE International Workshop on Hardware-Oriented Security and Trust (HOST 2009)* (2009), M. Tehranipoor and J. Plusquellic, Eds., IEEE, pp. 22–29.

[164] Kursawe, K., and Schellekens, D. Flexible $\mu$TPMs through Disembedding. In *4th ACM Symposium on Information, Computer and Communications Security (ASIACCS 2009)* (2009), W. Li, W. Susilo, U. K. Tupakula, R. Safavi-Naini, and V. Varadharajan, Eds., ACM, pp. 116–124.

[165] Kursawe, K., Schellekens, D., and Preneel, B. Analyzing trusted platform communication. In *ECRYPT Workshop on Cryptographic Advances in Secure Hardware (CRASH)* (Leuven, Belgium, 2005).

[166] Lach, J., Mangione-Smith, W. H., and Potkonjak, M. Fingerprinting Digital Circuits on Programmable Hardware. In *3rd International Workshop on Information Hiding (IH 1998)* (1998), D. Aucsmith, Ed., vol. 1525 of *Lecture Notes in Computer Science*, Springer, pp. 16–31.

[167] Lach, J., Mangione-Smith, W. H., and Potkonjak, M. Signature Hiding Techniques for FPGA Intellectual Property Protection. In *1998 IEEE/ACM International Conference on Computer-Aided Design (ICCAD 1998)* (1998), pp. 186–189.

[168] Lach, J., Mangione-Smith, W. H., and Potkonjak, M. Enhanced Intellectual Property Protection for Digital Circuits on Programmable Hardware. In *4th International Workshop on Information Hiding (IH 1999)* (1999), A. Pfitzmann, Ed., vol. 1768 of *Lecture Notes in Computer Science*, Springer, pp. 286–301.

[169] Lach, J., Mangione-Smith, W. H., and Potkonjak, M. Robust FPGA Intellectual Property Protection Through Multiple Small Watermarks. In *36th Conference on Design Automation (DAC 1999)* (1999), pp. 831–836.

[170] Lach, J., Mangione-Smith, W. H., and Potkonjak, M. Fingerprinting Techniques for Field-Programmable Gate Array Intellectual Property Protection. *IEEE Transactions on CAD of Integrated Circuits and Systems 20*, 10 (2001), 1253–1261.

[171] Lai, X., and Massey, J. L. Hash Function Based on Block Ciphers. In *Advances in Cryptology – EUROCRYPT 1992* (1993), R. A. Rueppel, Ed., vol. 658 of *Lecture Notes in Computer Science*, Springer, pp. 55–70.

[172] Lao, Y., and Parhi, K. K. Novel Reconfigurable Silicon Physical Unclonable Functions. In *Workshop on Foundations of Dependable and Secure Cyber-Physical Systems (FDSCPS 2011)* (2011), pp. 30–36.

[173] Lao, Y., and Parhi, K. K. Reconfigurable Architectures for Silicon Physical Unclonable Functions. In *2011 IEEE International Conference on Electro/Information Technology (EIT 2011)* (2011), pp. 1–7.

[174] Lattice. FPGA Design Security Issues: Using Lattice FPGAs to Achieve High Design Security, Sept. 2007. http://www.latticesemi.com/documents/doc18329x45.pdf.

[175] Levy, O., Kumar, A., and Goel, P. Advanced Security Features of Intel® vPro™ Technology. *Intel Technology Journal 12*, 4 (Dec. 2009), 229–238.

[176] Lim, D. Extracting Secret Keys from Integrated Circuits. Master's thesis, Massachusetts Institute of Technology, 2004.

[177] Limited, A. ARM Security Technology: Building A Secure System Using TrustZone Technology. Tech. Rep. PRD29-GENC-009492C, ARM Limited, Apr. 2009.

[178] Lin, A. H. Automated Analysis of Security APIs. Master's thesis, Massachusetts Institute of Technology, 2005.

[179] Linn, C., and Debray, S. K. Obfuscation of Executable Code to Improve Resistance to Static Disassembly. In *10th ACM Conference on Computer and Communications Security (CCS 2003)* (2003), S. Jajodia, V. Atluri, and T. Jaeger, Eds., ACM, pp. 290–299.

[180] Linnartz, J.-P. M. G., and Tuyls, P. New Shielding Functions to Enhance Privacy and Prevent Misuse of Biometric Templates. In *4th International Conference on Audio-and Video-Based Biometrie Person Authentication (AVBPA 2003)* (2003), J. Kittler and M. S. Nixon, Eds., vol. 2688 of *Lecture Notes in Computer Science*, Springer, pp. 393–402.

[181] Löhr, H., Ramasamy, H. V., Sadeghi, A.-R., Schulz, S., Schunter, M., and Stüble, C. Enhancing Grid Security Using Trusted Virtualization. In *4th International Conference on Autonomic and Trusted Computing (ATC 2007)* (2007), B. Xiao, L. T. Yang, J. Ma, C. Müller-Schloer, and Y. Hua, Eds., vol. 4610 of *Lecture Notes in Computer Science*, Springer, pp. 372–384.

[182] Maes, R. *Physically Unclonable Functions: Constructions, Properties and Applications*. PhD thesis, Katholieke Universiteit Leuven, Aug. 2012.

[183] Maes, R., Schellekens, D., and Verbauwhede, I. A Pay-per-Use Licensing Scheme for Hardware IP Cores in Recent SRAM-Based FPGAs. *IEEE Transactions on Information Forensics and Security 7*, 1 (2012), 98–108.

[184] Maes, R., Tuyls, P., and Verbauwhede, I. Intrinsic PUFs from Flip-flops on Reconfigurable Devices. In *3rd Benelux Workshop on Information and System Security (WISSec 2008)* (2008).

[185] Maes, R., Tuyls, P., and Verbauwhede, I. Low-Overhead Implementation of a Soft Decision Helper Data Algorithm for SRAM PUFs. In *11th International Workshop on Cryptographic Hardware and Embedded Systems (CHES 2009)* (2009), C. Clavier and K. Gaj, Eds., vol. 5747 of *Lecture Notes in Computer Science*, Springer, pp. 332–347.

[186] Maiti, A., and Schaumont, P. A Novel Microprocessor-intrinsic Physical Unclonable Function. In *22nd International Conference on Field Programmable Logic and Applications (FPL 2012)* (2012), IEEE, pp. 380–387.

[187] Majzoobi, M., Koushanfar, F., and Potkonjak, M. Techniques for Design and Implementation of Secure Reconfigurable PUFs. *ACM Transactions on Reconfigurable Technology and Systems 2*, 1 (2009), 1–33.

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

[189] Markettos, A. T., and Moore, S. W. The Frequency Injection Attack on Ring-Oscillator-Based True Random Number Generators. In

*11th International Workshop on Cryptographic Hardware and Embedded Systems (CHES 2009)* (2009), C. Clavier and K. Gaj, Eds., vol. 5747 of *Lecture Notes in Computer Science*, Springer, pp. 317–331.

[190] McCune, J. M. *Reducing the Trusted Computing Base for Applications on Commodity Systems.* PhD thesis, Carnegie Mellon University, Jan. 2009.

[191] McCune, J. M., Li, Y., Qu, N., Zhou, Z., Datta, A., Gligor, V., and Perrig, A. TrustVisor: Efficient TCB Reduction and Attestation. In *2010 IEEE Symposium on Security and Privacy (S&P 2010)* (2010), IEEE, pp. 143–158.

[192] McCune, J. M., Parno, B., Perrig, A., Reiter, M. K., and Isozaki, H. Flicker: An Execution Infrastructure for TCB Minimization. In *3th European Conference on Computer Systems (EuroSys 2008)* (2008), J. S. Sventek and S. Hand, Eds., ACM, pp. 315–328.

[193] McCune, J. M., Parno, B., Perrig, A., Reiter, M. K., and Seshadri, A. Minimal TCB Code Execution (Extended Abstract). In *2007 IEEE Symposium on Security and Privacy (S&P 2007)* (2007), IEEE, pp. 267–272.

[194] McCune, J. M., Parno, B., Perrig, A., Reiter, M. K., and Seshadri, A. How Low Can You Go?: Recommendations for Hardware-Supported Minimal TCB Code Execution. In *13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2008)* (2008), S. J. Eggers and J. R. Larus, Eds., ACM, pp. 14–25.

[195] McGrew, D. A., and Viega, J. The Security and Performance of the Galois/Counter Mode (GCM) of Operation. In *Progress in Cryptology – INDOCRYPT 2004* (2004), A. Canteaut and K. Viswanathan, Eds., vol. 3348 of *Lecture Notes in Computer Science*, Springer, pp. 343–355.

[196] Merkle, R. C. Protocols for Public Key Cryptosystems. In *1998 IEEE Symposium on Security and Privacy (S&P 1998)* (1980), pp. 122–134.

[197] Merli, D., Schuster, D., Stumpf, F., and Sigl, G. Semi-invasive EM Attack on FPGA RO PUFs and Countermeasures. In *6th Workshop on Embedded Systems Security (WESS 2011)* (2011), ACM, pp. 2:1–2:9.

[198] Merli, D., Schuster, D., Stumpf, F., and Sigl, G. Side-Channel Analysis of PUFs and Fuzzy Extractors. In *4th International Conference on Trust and Trustworthy Computing (TRUST 2011)* (2011), J. M. McCune, B. Balacheff, A. Perrig, A.-R. Sadeghi, A. Sasse, and Y. Beres, Eds., vol. 6740 of *Lecture Notes in Computer Science*, Springer, pp. 33–47.

[199] Michiels, W., and Gorissen, P. Mechanism for Software Tamper Resistance: An Application of White-Box Cryptography. In *7th ACM Workshop on Digital Rights Management (DRM 2007)* (2007), M. Yung, A. Kiayias, and A.-R. Sadeghi, Eds., ACM, pp. 82–89.

[200] Moradi, A., Barenghi, A., Kasper, T., and Paar, C. On the Vulnerability of FPGA Bitstream Encryption against Power Analysis Attacks – Extracting Keys from Xilinx Virtex-II FPGAs. In *18th ACM Conference on Computer and Communications Security (CCS 2011)* (2011), Y. Chen, G. Danezis, and V. Shmatikov, Eds., ACM, pp. 111–124.

[201] Moradi, A., Kasper, M., and Paar, C. Black-Box Side-Channel Attacks Highlight the Importance of Countermeasures – An Analysis of the Xilinx Virtex-4 and Virtex-5 Bitstream Encryption Mechanism. In *Topics in Cryptology – CT-RSA 2012* (2012), O. Dunkelman, Ed., vol. 7178 of *Lecture Notes in Computer Science*, Springer, pp. 1–18.

[202] Murray, D. G., Milos, G., and Hand, S. Improving Xen Security through Disaggregation. In *4th International Conference on Virtual Execution Environments (VEE 2008)* (2008), D. Gregg, V. S. Adve, and B. N. Bershad, Eds., ACM, pp. 151–160.

[203] Note, J.-B., and Rannaud, E. From the bitstream to the netlist. In *ACM/SIGDA 16th International Symposium on Field Programmable Gate Arrays (FPGA 2008)* (2008), M. Hutton and P. Chow, Eds., ACM, p. 264.

[204] Örs, S. B., Oswald, E., and Preneel, B. Power-Analysis Attacks on an FPGA – First Experimental Results. In *5th International Workshop on Cryptographic Hardware and Embedded Systems (CHES 2003)* (2003), C. D. Walter, Çetin Kaya Koç, and C. Paar, Eds., vol. 2779 of *Lecture Notes in Computer Science*, Springer, pp. 35–50.

[205] Papandreou, N., Pantazi, A., Sebastian, A., Breitwisch, M., Lamt, C., Pozidis, H., and Eleftheriou, E. Multilevel Phase-Change Memory. In *17th IEEE International Conference on Electronics, Circuits, and Systems (ICECS 2010)* (2010), IEEE, pp. 1017–1020.

[206] Pappu, R. S. *Physical One-Way Functions*. PhD thesis, Massachusetts Institute of Technology, Mar. 2001.

[207] Pappu, R. S., Recht, B., Taylor, J., and Gershenfeld, N. Physical One-Way Functions. *Science 297* (2002), 2026–2030.

[208] Parelkar, M. M., and Gaj, K. Implementation of EAX Mode of Operation for FPGA Bitstream Encryption and Authentication. In *2005 IEEE International Conference on Field-Programmable Technology (FPT*

*2005)* (2005), G. J. Brebner, S. Chakraborty, and W.-F. Wong, Eds., IEEE, pp. 335–336.

[209] Parno, B., McCune, J. M., and Perrig, A. Bootstrapping Trust in Commodity Computers. In *2010 IEEE Symposium on Security and Privacy (S&P 2010)* (2010), IEEE, pp. 414–429.

[210] Parno, B., McCune, J. M., and Perrig, A. *Bootstrapping Trust in Modern Computers*, vol. 10 of *SpringerBriefs in Computer Science.* Springer, 2011.

[211] Peeters, E., Standaert, F.-X., Donckers, N., and Quisquater, J.-J. Improved Higher-Order Side-Channel Attacks with FPGA Experiments. In *7th International Workshop on Cryptographic Hardware and Embedded Systems (CHES 2005)* (2005), J. R. Rao and B. Sunar, Eds., vol. 3659 of *Lecture Notes in Computer Science*, Springer, pp. 309–323.

[212] Peinado, M., Chen, Y., England, P., and Manferdelli, J. NGSCB: A Trusted Open System. In *9th Australasian Conference on Information Security and Privacy (ACISP 2004)* (2004), H. Wang, J. Pieprzyk, and V. Varadharajan, Eds., vol. 3108 of *Lecture Notes in Computer Science*, Springer, pp. 86–97.

[213] Petrank, E., and Rackoff, C. CBC MAC for Real-Time Data Sources. *Journal of Cryptology 13*, 3 (2000), 315–338.

[214] Petroni Jr., N. L., Fraser, T., Molina, J., and Arbaugh, W. A. Copilot – a Coprocessor-based Kernel Runtime Integrity Monitor. In *13th USENIX Security Symposium* (2004), USENIX, pp. 179–194.

[215] Pirker, M., Toegl, R., Hein, D. M., and Danner, P. A PrivacyCA for Anonymity and Trust. In *2nd International Conference on Trusted Computing (Trust 2009)* (2009), L. Chen, C. J. Mitchell, and A. Martin, Eds., vol. 5471 of *Lecture Notes in Computer Science*, Springer, pp. 101–119.

[216] Poritz, J., Schunter, M., Van Herreweghen, E., and Waidner, M. Property Attestation – Scalable and Privacy-friendly Security Assessment of Peer Computers. Tech. Rep. RZ 3548, IBM Research, May 2004.

[217] Prabhu, P., Akel, A., Grupp, L. M., Yu, W.-K. S., Suh, G. E., Kan, E., and Swanson, S. Extracting Device Fingerprints from Flash Memory by Exploiting Physical Variations. In *4th International Conference on Trust and Trustworthy Computing (TRUST 2011)* (2011), J. M. McCune, B. Balacheff, A. Perrig, A.-R. Sadeghi, A. Sasse, and

Y. Beres, Eds., vol. 6740 of *Lecture Notes in Computer Science*, Springer, pp. 188–201.

[218] Preneel, B., Govaerts, R., and Vandewalle, J. Hash Functions Based on Block Ciphers: A Synthetic Approach. In *Advances in Cryptology – CRYPTO 1993* (1994), D. R. Stinson, Ed., vol. 773 of *Lecture Notes in Computer Science*, Springer, pp. 368–378.

[219] Quisquater, J.-J., and Samyde, D. ElectroMagnetic Analysis (EMA): Measures and Counter-Measures for Smart Cards. In *International Conference on Research in Smart Cards (E-smart 2001)* (2001), I. Attali and T. P. Jensen, Eds., vol. 2140 of *Lecture Notes in Computer Science*, Springer, pp. 200–210.

[220] Raoux, S., Burr, G. W., Breitwisch, M. J., Rettner, C. T., Chen, Y.-C., Shelby, R. M., Salinga, M., Krebs, D., Chen, S.-H., Lung, H.-L., and Lam, C. H. Phase-change random access memory: A scalable technology. *IBM Journal of Research and Development 52*, 4-5 (2008), 465–480.

[221] Rogaway, P., Bellare, M., Black, J., and Krovetz, T. OCB: A Block-Cipher Mode of Operation for Efficient Authenticated Encryption. In *8th ACM Conference on Computer and Communications Security (CCS 2001)* (2001), M. K. Reiter and P. Samarati, Eds., ACM, pp. 196–205.

[222] Rogaway, P., and Steinberger, J. P. Constructing Cryptographic Hash Functions from Fixed-Key Blockciphers. In *Advances in Cryptology – CRYPTO 2008* (2008), D. Wagner, Ed., vol. 5157 of *Lecture Notes in Computer Science*, Springer, pp. 433–450.

[223] Sadeghi, A.-R., Selhorst, M., Stüble, C., Wachsmann, C., and Winandy, M. TCG inside?: A Note on TPM Specification Compliance. In *1st ACM Workshop on Scalable Trusted Computing (STC 2006)* (2006), A. Juels, G. Tsudik, S. Xu, and M. Yung, Eds., ACM, pp. 47–56.

[224] Sadeghi, A.-R., and Stüble, C. Property-based Attestation for Computing Platforms: Caring about properties, not mechanisms. In *New Security Paradigms Workshop 2004* (2004), C. Hempelmann and V. Raskin, Eds., ACM, pp. 67–77.

[225] Sadeghi, A.-R., Stüble, C., and Pohlmann, N. European Multilateral Secure Computing Base – Open Trusted Computing for You and Me. *Datenschutz und Datensicherheit (DUD) 9* (Sept. 2004), 548–554.

[226] Sailer, R., Zhang, X., Jaeger, T., and van Doorn, L. Design and Implementation of a TCG-based Integrity Measurement Architecture. In *13th USENIX Security Symposium* (2004), USENIX, pp. 223–238.

[227] Sarmenta, L. F. G., van Dijk, M., O'Donnell, C. W., Rhodes, J., and Devadas, S. Virtual Monotonic Counters and Count-Limited Objects using a TPM without a Trusted OS. In *1st ACM Workshop on Scalable Trusted Computing (STC 2006)* (2006), A. Juels, G. Tsudik, S. Xu, and M. Yung, Eds., ACM, pp. 27–42.

[228] Schellekens, D., Tuyls, P., and Preneel, B. Embedded Trusted Computing with Authenticated Non-Volatile Memory. In *1st International Conference on Trusted Computing and Trust in Information Technologies (Trust 2008)* (2008), K.-M. Koch, P. Lipp, and A.-R. Sadeghi, Eds., vol. 4968 of *Lecture Notes in Computer Science*, Springer, pp. 60–74.

[229] Schellekens, D., Wyseur, B., and Preneel, B. Remote Attestation on Legacy Operating Systems With Trusted Platform Modules. In *1st International Workshop on Run Time Enforcement for Mobile and Distributed Systems (REM 2007)* (2008), F. Massacci and F. Piessens, Eds., vol. 197 of *Electronic Notes in Theoretical Computer Science*, Elsevier, pp. 59–72.

[230] Schellekens, D., Wyseur, B., and Preneel, B. Remote attestation on legacy operating systems with trusted platform modules. *Science of Computer Programming 74*, 1-2 (2008), 13–22.

[231] Schmidt, A. U., Kuntze, N., and Kasper, M. On the deployment of Mobile Trusted Modules. In *2008 IEEE Wireless Communications and Networking Conference (WCNC 2008)* (2008), IEEE, pp. 3169–3174.

[232] Schulz, S., Sadeghi, A.-R., and Wachsmann, C. Short Paper: Lightweight Remote Attestation Using Physical Functions. In *4th ACM Conference on Wireless Network Security (WISEC 2011)* (2011), D. Gollmann, D. Westhoff, G. Tsudik, and N. Asokan, Eds., ACM, pp. 109–114.

[233] Schulz, S., Wachsmann, C., and Sadeghi, A.-R. Lightweight Remote Attestation using Physical Functions. Tech. Rep. TR-2011-06-01, Technische Universität Darmstadt, July 2011.

[234] Selhorst, M., Stüble, C., Feldmann, F., and Gnaida, U. Towards a Trusted Mobile Desktop. In *3rd International Conference on Trust and Trustworthy Computing (TRUST 2010)* (2010), A. Acquisti, S. W. Smith, and A.-R. Sadeghi, Eds., vol. 6101 of *Lecture Notes in Computer Science*, Springer, pp. 78–94.

[235] Seshadri, A. *A Software Primitive for Externally-verifiable Untampered Execution and its Applications to Securing Computing Systems.* PhD thesis, Carnegie Mellon University, 2009.

[236] Seshadri, A., Luk, M., Perrig, A., van Doorn, L., and Khosla, P. K. Externally Verifiable Code Execution. *Commununications of the ACM 49*, 9 (2006), 45–49.

[237] Seshadri, A., Luk, M., Shi, E., Perrig, A., van Doorn, L., and Khosla, P. K. Pioneer: Verifying Code Integrity and Enforcing Untampered Code Execution on Legacy Systems. In *20th ACM Symposium on Operating Systems Principles (SOSP 2005)* (2005), A. Herbert and K. P. Birman, Eds., ACM, pp. 1–16.

[238] Seshadri, A., Perrig, A., van Doorn, L., and Khosla, P. K. SWATT: SoftWare-based ATTestation for Embedded Devices. In *2004 IEEE Symposium on Security and Privacy (S&P 2004)* (2004), IEEE, pp. 272–282.

[239] Shamir, A., and van Someren, N. Playing 'Hide and Seek' with Stored Keys. In *3rd International Conference on Financial Cryptography (FC 1999)* (1999), M. K. Franklin, Ed., vol. 1648 of *Lecture Notes in Computer Science*, Springer, pp. 118–124.

[240] Shankar, U., Chew, M., and Tygar, J. D. Side Effects Are Not Sufficient to Authenticate Software. In *13th USENIX Security Symposium* (2004), USENIX, pp. 89–102.

[241] Shi, E., Perrig, A., and van Doorn, L. BIND: A Fine-Grained Attestation Service for Secure Distributed Systems. In *2005 IEEE Symposium on Security and Privacy (S&P 2005)* (2005), IEEE, pp. 154–168.

[242] Simmons, G. J. Identification of Data, Devices, Documents and Individuals. In *25th IEEE International Carnahan Conference on Security Technology – ICCST 1991* (1991), IEEE, pp. 197–218.

[243] Simpson, E., and Schaumont, P. Offline Hardware/Software Authentication for Reconfigurable Platforms. In *8th International Workshop on Cryptographic Hardware and Embedded Systems (CHES 2006)* (2006), L. Goubin and M. Matsui, Eds., vol. 4249 of *Lecture Notes in Computer Science*, Springer, pp. 311–323.

[244] Skorobogatov, S. P., and Anderson, R. J. Optical Fault Induction Attacks. In *4th International Workshop on Cryptographic Hardware and Embedded Systems (CHES 2002)* (2003), B. S. Kaliski Jr., Çetin

Kaya Koç, and C. Paar, Eds., vol. 2523 of *Lecture Notes in Computer Science*, Springer, pp. 2–12.

[245] Smerdon, M., and Durant, G. Taking Device DNA Technology to the Next Level. *XCell Journal 63* (2008), 49–52. http://www.xilinx.com/publications/xcellonline/xcell_63/xc_pdf/p49_52_63-helion.pdf.

[246] Smith, S. W. Outbound Authentication for Programmable Secure Coprocessors. In *7th European Symposium on Research in Computer Security (ESORICS 2002)* (2002), D. Gollmann, G. Karjoth, and M. Waidner, Eds., vol. 2502 of *Lecture Notes in Computer Science*, Springer, pp. 72–89.

[247] Smith, S. W. Outbound authentication for programmable secure coprocessors. *International Journal of Information Security 3*, 1 (2004), 28–41.

[248] Smith, S. W., Palmer, E. R., and Weingart, S. Using a High-Performance, Programmable Secure Coprocessor. In *2nd International Conference on Financial Cryptography (FC 1998)* (1998), R. Hirschfeld, Ed., vol. 1465 of *Lecture Notes in Computer Science*, Springer, pp. 73–89.

[249] Smith, S. W., and Weingart, S. Building a High-Performance, Programmable Secure Coprocessor. *Computer Networks 31*, 8 (1999), 831–860.

[250] Sparks, E. R. A Security Assessment of Trusted Platform Modules. Tech. Rep. TR2007-597, Dartmouth College, Computer Science, June 2007.

[251] Stam, M. Blockcipher-Based Hashing Revisited. In *16th International Workshop on Fast Software Encryption (FSE 2009)* (2009), O. Dunkelman, Ed., vol. 5665 of *Lecture Notes in Computer Science*, Springer, pp. 67–83.

[252] Stamer, H., and Strasser, M. A Software-Based Trusted Platform Module Emulator. In *1st International Conference on Trusted Computing and Trust in Information Technologies (Trust 2008)* (2008), P. Lipp, A.-R. Sadeghi, and K.-M. Koch, Eds., vol. 4968 of *Lecture Notes in Computer Science*, Springer, pp. 33–47.

[253] Standaert, F.-X., Macé, F., Peeters, E., and Quisquater, J.-J. Updates on the Security of FPGAs Against Power Analysis Attacks. In *2nd International Workshop on Applied Reconfigurable Computing (ARC 2006)* (2006), K. Bertels, J. ao M. P. Cardoso, and S. Vassiliadis, Eds., vol. 3985 of *Lecture Notes in Computer Science*, Springer, pp. 335–346.

[254] Standaert, F.-X., Örs, S. B., and Preneel, B. Power Analysis of an FPGA: Implementation of Rijndael: Is Pipelining a DPA Countermeasure? In *6th International Workshop on Cryptographic Hardware and Embedded Systems (CHES 2004)* (2004), M. Joye and J.-J. Quisquater, Eds., vol. 3156 of *Lecture Notes in Computer Science*, Springer, pp. 30–44.

[255] Standaert, F.-X., Örs, S. B., Quisquater, J.-J., and Preneel, B. Power Analysis Attacks Against FPGA Implementations of the DES. In *14th International Conference on Field Programmable Logic and Applications (FPL 2004)* (2004), J. Becker, M. Platzner, and S. Vernalde, Eds., vol. 3203 of *Lecture Notes in Computer Science*, Springer, pp. 84–94.

[256] Standaert, F.-X., Peeters, E., Rouvroy, G., and Quisquater, J.-J. An Overview of Power Analysis Attacks Against Field Programmable Gate Arrays. *Proceedings of the IEEE 94*, 2 (2006), 383–394.

[257] Standaert, F.-X., Rouvroy, G., and Quisquater, J.-J. FPGA Implementations of the DES and Triple-DES Masked Against Power Analysis Attacks. In *16th International Conference on Field Programmable Logic and Applications (FPL 2006)* (2006), IEEE, pp. 1–4.

[258] Standaert, F.-X., van Oldeneel tot Oldenzeel, L., Samyde, D., and Quisquater, J.-J. Power Analysis of FPGAs: How Practical is the Attack? In *13th International Conference on Field Programmable Logic and Applications (FPL 2003)* (2003), P. Y. K. Cheung, G. A. Constantinides, and J. T. de Sousa, Eds., vol. 2778 of *Lecture Notes in Computer Science*, Springer, pp. 701–711.

[259] Steffen, A. The Linux Integrity Measurement Architecture and TPM-Based Network Endpoint Assessment. In *Linux Security Summit 2012* (2012).

[260] Steinberg, U., and Kauer, B. NOVA: A Microhypervisor-Based Secure Virtualization Architecture. In *5th European Conference on Computer Systems (EuroSys 2010)* (2010), C. Morin and G. Muller, Eds., ACM, pp. 209–222.

[261] Strackx, R., and Piessens, F. Fides: Selectively Hardening Software Application Components against Kernel-level or Process-level Malware. In *19th ACM Conference on Computer and Communications Security (CCS 2012)* (2012), T. Yu, G. Danezis, and V. D. Gligor, Eds., ACM, pp. 2–13.

[262] Strackx, R., Piessens, F., and Preneel, B. Efficient Isolation of Trusted Subsystems in Embedded Systems. In *6h International Conference on Security and Privacy in Communication Networks (SecureComm 2010)*

(2010), S. Jajodia and J. Zhou, Eds., vol. 50 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, Springer, pp. 344–361.

[263] Su, Y., Holleman, J., and Otis, B. P. A Digital 1.6 pJ/bit Chip Identification Circuit Using Process Variations. In *2007 IEEE International Solid-State Circuits Conference (ISSCC 2007)* (2007), IEEE, pp. 15–17.

[264] Suh, G. E. *AEGIS: A Single-Chip Secure Processor.* PhD thesis, Massachusetts Institute of Technology, Sept. 2005.

[265] Suh, G. E., Clarke, D. E., Gassend, B., van Dijk, M., and Devadas, S. AEGIS: Architecture for Tamper-Evident and Tamper-Resistant Processing. In *17th Annual International Conference on Supercomputing (ICS 2003)* (2003), U. Banerjee, K. Gallivan, and A. González, Eds., ACM, pp. 160–171.

[266] Suh, G. E., Clarke, D. E., Gassend, B., van Dijk, M., and Devadas, S. Efficient Memory Integrity Verification and Encryption for Secure Processors. In *36th Annual International Symposium on Microarchitecture (MICRO 2003)* (2003), ACM/IEEE, pp. 339–350.

[267] Suh, G. E., and Devadas, S. Physical Unclonable Functions for Device Authentication and Secret Key Generation. In *44th Design Automation Conference (DAC 2007)* (2007), IEEE, pp. 9–14.

[268] Suh, G. E., O'Donnell, C. W., Sachdev, I., and Devadas, S. Design and Implementation of the AEGIS Single-Chip Secure Processor Using Physical Random Functions. In *32st International Symposium on Computer Architecture (ISCA 2005)* (2005), IEEE, pp. 25–36.

[269] Suzuki, D., and Shimizu, K. The Glitch PUF: A New Delay-PUF Architecture Exploiting Glitch Shapes. In *12th International Workshop on Cryptographic Hardware and Embedded Systems (CHES 2010)* (2010), S. Mangard and F.-X. Standaert, Eds., vol. 6225 of *Lecture Notes in Computer Science*, Springer, pp. 366–382.

[270] Tan, G., Chen, Y., and Jakubowski, M. H. Delayed and Controlled Failures in Tamper-Resistant Systems. In *8th International Workshop on Information Hiding (IH 2006)* (2007), J. Camenisch, C. S. Collberg, N. F. Johnson, and P. Sallee, Eds., vol. 4437 of *Lecture Notes in Computer Science*, Springer, pp. 216–231.

[271] Tarnovsky, C. Deconstructing a 'Secure' Processor. In *Black Hat DC 2010* (2010).

[272] Telikepalli, A. Is Your FPGA Design Secure? *XCell Journal 47* (2003). http://www.xilinx.com/publications/xcellonline/xcell_47/xc_pdf/xc_secure47.pdf.

[273] Tereshkin, A., and Wojtczuk, R. Introducing Ring -3 Rootkits. In *Black Hat USA 2009* (2009).

[274] Tiri, K., and Verbauwhede, I. A Logic Level Design Methodology for a Secure DPA Resistant ASIC or FPGA Implementation. In *2004 Design, Automation and Test in Europe Conference and Exposition (DATE 2004)* (2004), IEEE, pp. 246–251.

[275] Tiri, K., and Verbauwhede, I. Synthesis of Secure FPGA Implementations. In *13th International Workshop on Logic and Synthesis (IWLS 2004)* (2004), pp. 224–231.

[276] Tolk, K. M. Reflective Particle Technology for Identification of Critical Components. In *33rd Institute of Nuclear Materials Management Annual Meeting* (1992), pp. 648–652.

[277] Torrance, R., and James, D. The State-of-the-Art in IC Reverse Engineering. In *11th International Workshop on Cryptographic Hardware and Embedded Systems (CHES 2009)* (2009), C. Clavier and K. Gaj, Eds., vol. 5747 of *Lecture Notes in Computer Science*, Springer, pp. 363–381.

[278] Trimberger, S. Trusted design in FPGAs. In *44th Design Automation Conference (DAC 2007)* (2007), ACM, pp. 5–8.

[279] Trusted Computing Group. PC Client Specific TPM Interface Specification (TIS), May 2011. http://www.trustedcomputinggroup.org/developers/pc_client/specifications.

[280] Trusted Computing Group. Storage Architecture Core Specification, Nov. 2011. http://www.trustedcomputinggroup.org/developers/storage/specifications.

[281] Trusted Computing Group. TPM Main Part 1 Design Principles Specification Version 1.2, Mar. 2011. http://www.trustedcomputinggroup.org/resources/tpm_main_specification.

[282] Trusted Computing Group. TPM Main Part 2 TPM Structures Specification Version 1.2, Mar. 2011. http://www.trustedcomputinggroup.org/resources/tpm_main_specification.

[283] Trusted Computing Group. TPM Main Part 3 Commands Specification Version 1.2, Mar. 2011. http://www.trustedcomputinggroup.org/resources/tpm_main_specification.

[284] Trusted Computing Group. PC Client Specific Implementation Specification For Conventional BIOS, Feb. 2012. http://www.trustedcomputinggroup.org/developers/pc_client/specifications.

[285] Trusted Computing Platform Alliance. Main Specification Version 1.1b, Feb. 2002. http://www.trustedcomputinggroup.org/resources/tpm_main_specification.

[286] Tseng, C. W. Lock Your Designs with the Virtex-4 Security Solution. *XCell Journal 52* (2005). http://www.xilinx.com/publications/xcellonline/xcell_52/xc_pdf/xc_v4security52.pdf.

[287] Tuyls, P., Schrijen, G. J., Škorić, B., van Geloven, J., Verhaegh, N., and Wolters, R. Read-Proof Hardware from Protective Coatings. In *8th International Workshop on Cryptographic Hardware and Embedded Systems (CHES 2006)* (2006), L. Goubin and M. Matsui, Eds., vol. 4249 of *Lecture Notes in Computer Science*, Springer, pp. 369–383.

[288] Tuyls, P., Škorić, B., and Kevenaar, T., Eds. *Security with Noisy Data: On Private Biometrics, Secure Key Storage and Anti-Counterfeiting.* Springer, 2007.

[289] Tuyls, P., Škorić, B., Stallinga, S., Akkermans, A. H. M., and Ophey, W. Information-Theoretic Security Analysis of Physical Uncloneable Functions. In *10th International Conference on Financial Cryptography and Data Security (FC 2005)* (2005), A. S. Patrick and M. Yung, Eds., vol. 3570 of *Lecture Notes in Computer Science*, Springer, pp. 141–155.

[290] van der Leest, V., Schrijen, G.-J., Handschuh, H., and Tuyls, P. Hardware Intrinsic Security from D flip-flops. In *5th ACM Workshop on Scalable Trusted Computing (STC 2010)* (2010), S. Xu, N. Asokan, and A.-R. Sadeghi, Eds., ACM, pp. 53–62.

[291] Van Le, T., and Desmedt, Y. Cryptanalysis of UCLA Watermarking Schemes for Intellectual Property Protection. In *5th International Workshop on Information Hiding (IH 2002)* (2002), F. A. P. Petitcolas, Ed., vol. 2578 of *Lecture Notes in Computer Science*, Springer, pp. 213–225.

[292] van Oorschot, P. C., Somayaji, A., and Wurster, G. Hardware-Assisted Circumvention of Self-Hashing Software Tamper Resistance. *IEEE Transactions on Dependable and Secure Computing 2*, 2 (2005), 82–92.

[293] Verdult, R., Garcia, F. D., and Balasch, J. Gone in 360 Seconds: Hijacking with Hitag2. In *21st USENIX Security Symposium* (2012), USENIX, pp. 237–252.

[294] Škorić, B., Tuyls, P., and Ophey, W. Robust Key Extraction from Physical Unclonable Functions. In *3rd International Conference on Applied Cryptography and Network Security (ACNS 2005)* (2005), J. Ioannidis, A. D. Keromytis, and M. Yung, Eds., vol. 3531 of *Lecture Notes in Computer Science*, Springer, pp. 407–422.

[295] Wang, Y., Yu, W.-K. S., Wu, S., Malysa, G., Suh, G. E., and Kan, E. Flash Memory for Ubiquitous Hardware Security Functions: True Random Number Generation and Device Fingerprints. In *2012 IEEE Symposium on Security and Privacy (S&P 2012)* (2012), IEEE, pp. 33–47.

[296] Wilson, P., Frey, A., Mihm, T., Kershaw, D., and Alves, T. Implementing Embedded Security on Dual-Virtual-CPU Systems. *IEEE Design and Test of Computers 24*, 6 (2007), 582–591.

[297] Winter, J. Trusted Computing Building Blocks for Embedded Linux-based ARM TrustZone Platforms. In *3rd ACM Workshop on Scalable Trusted Computing (STC 2008)* (2008), S. Xu, C. Nita-Rotaru, and J.-P. Seifert, Eds., ACM, pp. 21–30.

[298] Winter, J. Eavesdropping Trusted Platform Module Communication. Tech. rep., Institute for Applied Information Processing and Communications, Technische Universität Graz, July 2009.

[299] Winter, J., and Dietrich, K. A hijacker's guide to communication interfaces of the trusted platform module. *Computers and Mathematics with Applications* (2012). to appear.

[300] Winter, J., and Dietrich, K. A Hijacker's Guide to the LPC Bus. In *8th European Workshop on Public Key Infrastructures, Services and Applications (EuroPKI 2011)* (2012), S. Petkova-Nikova, A. Pashalidis, and G. Pernul, Eds., vol. 7163 of *Lecture Notes in Computer Science*, Springer, pp. 176–193.

[301] Wojtczuk, R., and Rutkowska, J. Attacking Intel® Trusted Execution Technology. In *Black Hat DC 2009* (2009).

[302] Wojtczuk, R., and Rutkowska, J. Attacking SMM Memory via Intel® CPU Cache Poisoning, Mar. 2009.

[303] Wojtczuk, R., Rutkowska, J., and Tereshkin, A. Another Way to Circumvent Intel® Trusted Execution Technology, Dec. 2009.

[304] Wojtczuk, R., and Tereshkin, A. Attacking Intel® BIOS. In *Black Hat USA 2009* (2009).

[305] Wollinger, T. J., Guajardo, J., and Paar, C. Security on FPGAs: State-of-the-art implementations and attacks. *ACM Transactions on Embedded Computing Systems 3*, 3 (2004), 534–574.

[306] Wong, H.-S. P., Raoux, S., Kim, S., Liang, J., Reifenberg, J. P., Rajendran, B., Asheghi, M., and Goodson, K. E. Phase Change Memory. *Proceedings of the IEEE 98*, 12 (2010), 2201–2227.

[307] Wroblewski, G. *General Method of Program Code Obfuscation.* PhD thesis, Wroclaw University of Technology, 2002.

[308] Wurster, G., van Oorschot, P. C., and Somayaji, A. A Generic Attack on Checksumming-Based Software Tamper Resistance. In *2005 IEEE Symposium on Security and Privacy (S&P 2005)* (2005), IEEE, pp. 127–138.

[309] Wyseur, B. *White-Box Cryptography.* PhD thesis, Katholieke Universiteit Leuven, 2009.

[310] Wyseur, B., Michiels, W., Gorissen, P., and Preneel, B. Cryptanalysis of White-Box DES Implementations with Arbitrary External Encodings. In *14th International Workshop on Selected Areas in Cryptography (SAC 2007)* (2007), C. M. Adams, A. Miri, and M. J. Wiener, Eds., vol. 4876 of *Lecture Notes in Computer Science*, Springer, pp. 264–277.

[311] Xilinx. Spartan-3AN FPGA In-System Flash User Guide, Jan. 2009. http://www.xilinx.com/support/documentation/user_guides/ug333.pdf.

[312] Yee, B. S. *Using Secure Coprocessors.* PhD thesis, Carnegie Mellon University, May 1994.

[313] Yee, B. S., and Tygar, J. D. Secure Coprocessors in Electronic Commerce Applications. In *1st USENIX Workshop on Electronic Commerce (EC 1995)* (1995), pp. 155–170.

[314] Yu, M.-D. M., and Devadas, S. Secure and Robust Error Correction for Physical Unclonable Functions. *IEEE Design & Test of Computers 27*, 1 (2010), 48–65.

[315] Yu, P. Implementation of DPA-Resistant Circuit for FPGA. Master's thesis, Virginia Polytechnic Institute and State University, 2007.

[316] Yu, P., and Schaumont, P. Secure FPGA circuits using controlled placement and routing. In *5th International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS 2007)* (2007), S. Ha, K. Choi, N. D. Dutt, and J. Teich, Eds., ACM, pp. 45–50.

[317] Yuan, L., Pari, P. R., and Qu, G. Soft IP Protection: Watermarking HDL Codes. In *6th International Workshop on Information Hiding (IH 2004)* (2004), J. J. Fridrich, Ed., vol. 3200 of *Lecture Notes in Computer Science*, Springer, pp. 224–238.

[318] Zeineddini, A. S., and Gaj, K. Secure Partial Reconfiguration of FPGAs. In *2005 IEEE International Conference on Field-Programmable Technology (FPT 2005)* (2005), IEEE, pp. 155–162.

[319] Zhang, X., Aciiçmez, O., and Seifert, J.-P. A Trusted Mobile Phone Reference Architecture via Secure Kernel. In *2nd ACM Workshop on Scalable Trusted Computing (STC 2007)* (2007), P. Ning, V. Atluri, S. Xu, and M. Yung, Eds., ACM, pp. 7–14.

[320] Zhang, X., and Gupta, R. Hiding Program Slices for Software Security. In *1st IEEE/ACM International Symposium on Code Generation and Optimization (CGO 2003)* (2003), IEEE, pp. 325–336.

[321] Ziener, D., Assmus, S., and Teich, J. Identifying FPGA IP-Cores Based on Lookup Table Content Analysis. In *16th International Conference on Field Programmable Logic and Applications (FPL 2006)* (2006), IEEE, pp. 1–6.

[322] Ziener, D., and Teich, J. Power Signature Watermarking of IP Cores for FPGAs. *Signal Processing Systems 51*, 1 (2008), 123–136.

# Curriculum Vitae

Dries Schellekens was born on March 24, 1979 in Zoersel, Belgium. He received the Master's degree in Electrical Engineering (Multimedia and Signal Processing) from the KU Leuven, Belgium in July 2002. His master thesis dealt with the security of peer-to-peer protocols. In October 2002, he joined the COSIC (Computer Security and Industrial Cryptography) research group at the Department of Electrical Engineering (ESAT) of KU Leuven as a research assistant.

He has been involved in a number of European and national research projects, including PAMPAS (Pioneering Advanced Mobile Privacy and Security), OpenTC (Open Trusted Computing), RE-TRUST (Remote Entrusting by Run-Time Software Authentication), SoBeNeT (Software Security for Network Applications), SEC SODA (Security of Software for Distributed Applications), OMUS (Optimizing Multimedia Service Delivery), and EVENT (Electronic Vouchers for Events using NFC Technology).

# List of publications

## International Journals

1. G. Ergeerts, D. Schellekens, F. Schrooyen, R. Beyers, K. De Kock, T. Van Herck, “Vision towards an Open Electronic Wallet on NFC Smartphones,” *International Journal of Electronics and Communications, under submission*

2. R. Maes, D. Schellekens, I. Verbauwhede, “A Pay-per-Use Licensing Scheme for Hardware IP Cores in Recent SRAM based FPGAs,” *IEEE Transactions on Information Forensics and Security 7(1)*, pp. 98-108, 2012.

3. D. Schellekens, B. Wyseur, B. Preneel, “Remote attestation on legacy operating systems with trusted platform modules,” *Science of Computer Programming 74(1-2)*, pp. 13-22, 2008.

## National Journals

4. K. Kursawe, D. Schellekens, “Trusted Platforms,” *Revue HF Tijdschrift 2004(3)*, pp. 46-54, 2004.

## Book Chapters

5. B. Sunar, D. Schellekens, “Random Number Generators for Integrated Circuits and FPGAs,” *Secure Integrated Circuits and Systems, Integrated Circuits and Systems*, I. Verbauwhede, Ed., Springer, pp. 107-124, 2010.

## International Conferences and Workshops

6. K. Kursawe, A. Sadeghi, D. Schellekens, P. Tuyls, B. Škorić, "Reconfigurable Physical Unclonable Functions – Enabling Technology for Tamper-Resistant Storage ," *2nd IEEE International Symposium on Hardware-Oriented Security and Trust - HOST 2009*, IEEE, pp. 22-29, 2009.

7. R. Maes, D. Schellekens, P. Tuyls, I. Verbauwhede, "Analysis and Design of Active IC Metering Schemes," *2nd IEEE International Symposium on Hardware-Oriented Security and Trust - HOST 2009*, IEEE, pp. 74-81, 2009.

8. K. Kursawe, D. Schellekens, "Flexible $\mu$TPMs through Disembedding," *Proceedings of the 4th ACM Symposium on Information, Computer, and Communications Security (ASIACCS 2009)*, ACM, pp. 116-124, 2009.

9. D. Schellekens, P. Tuyls, B. Preneel, "Embedded Trusted Computing with Authenticated Non-Volatile Memory," *1st International Conference on Trusted Computing and Trust in Information Technologies, TRUST 2008, Lecture Notes in Computer Science 4968*, K. Koch, P. Lipp, A. Sadeghi, Eds., Springer-Verlag, pp. 60-74, 2008.

10. D. Schellekens, B. Wyseur, B. Preneel, "Remote Attestation on Legacy Operating Systems With Trusted Platform Modules," *1st International Workshop on Run Time Enforcement for Mobile and Distributed Systems (REM 2007), Electronic Notes in Theoretical Computer Science 197(1)*, F. Massacci , F. Piessens, Eds., Elsevier, pp. 59-72, 2008.

11. T. Eisenbarth, T. Güneysu, C. Paar, A. Sadeghi, D. Schellekens, M. Wolf, "Reconfigurable Trusted Computing in Hardware," *Proceedings of the 2nd ACM Workshop on Scalable Trusted Computing (STC 2007)*, ACM, pp. 15-20, 2007.

12. D. Schellekens, B. Preneel, I. Verbauwhede, "FPGA vendor agnostic True Random Number Generator," *16th International Conference on Field Programmable Logic and Applications (FPL 2006)*, IEEE, pp. 139-144, 2006.

13. D. De Cock, K. Wouters, D. Schellekens, D. Singelée, B. Preneel, "Threat Modelling for Security Tokens in Web Applications," *Proceedings of the IFIP TC6/TC11 International Conference on Communications and Multimedia Security (CMS 2004), IFIP International Federation for Information Processing 175*, D. Chadwick, B. Preneel, Eds., Springer, pp. 183-193, 2005.

# Other Articles

14. N. Mentens, D. Schellekens, W. Heedfeld, P. Timmermans, I. Verbauwhede, "Comparison of two Random Number Generators on an FPGA," *ProRISC workshop*, 3 pages, 2008.

15. J. Cappaert, N. Kisserli, D. Schellekens, B. Preneel, "Self-encrypting Code to Protect against Analysis and Tampering," *1st Benelux Workshop on Information and System Security (WISSec 2006)*, 14 pages, 2006.

16. K. Kursawe, D. Schellekens, B. Preneel, "Analyzing trusted platform communication," *ECRYPT Workshop, CRASH - CRyptographic Advances in Secure Hardware*, 8 pages, 2005.

17. D. De Cock, B. Preneel, D. Schellekens, D. Singelée, K. Wouters, "Threat modelling for security tokens," COSIC internal report, 36 pages, 2004.