Technische Universität München
Fakultät für Informatik

Lehrstuhl für Sicherheit in der Informatik

Implicit Remote Attestation of Microkernel-based Embedded Systems

Steffen Wagner

Dissertation

Technische Universität München
Lehrstuhl für Sicherheit in der Informatik
an der Fakultät für Informatik

Implicit Remote Attestation of Microkernel-based Embedded Systems

Steffen Wagner

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:	Univ.-Prof. Dr.-Ing. Jörg Ott
Prüfer der Dissertation:	1. Univ.-Prof. Dr. Claudia Eckert
	2. Univ.-Prof. Dr. Uwe Baumgarten

Die Dissertation wurde am 08.11.2017 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 21.03.2018 angenommen.

Abstract

As embedded systems become ubiquitous and consistently gain new hardware and software features, their complexity and code base continue to grow. Consequently, the attack surface and hence the probability of successful attacks potentially increases, which can not only put users at risk, but also corporation, such as mobile network operators. For instance, distributed denial-of-service (DDoS) attacks from compromised mobile devices are a major threat to mobile networks. That is why we propose a mechanism based on a Trusted Platform Module (TPM), which enables mobile devices to prove that their baseband stack is still trustworthy, while allowing the network to enforce a certain version of the baseband at network connect. As a result, our approach represents an effective method to block devices with a compromised baseband stack and reduce DDoS attacks against mobile networks using remote attestation. In contrast to traditional remote attestation as specified by the Trusted Computing Group (TCG) however, our *implicit attestation mechanism* enables lightweight attestation protocols, since it relies on efficient symmetric cryptographic operations and hash-based message authentication codes instead of asymmetric cryptography, especially digital signatures, and extensive measurement logs. As a result, our implicit attestation mechanism is particularly suitable for resource-constraint embedded devices.

To further improve security, we propose a system architecture based on microkernels, which are not only significantly smaller, but also less complex than monolithic kernels like Linux. While most microkernel-based systems implement non-essential services as user-space tasks and strictly separate those tasks during runtime, they often rely on a static configuration to ensure safety and security. Although that benefits our implicit attestation mechanism, it does not necessarily imply trustworthiness. That is why we combine our microkernel-based system architecture with a TPM and propose an integrity verification mechanism for microkernel execution environments, which calculates integrity measurements before loading (remote) binaries. As a result, our approach is the first to adopt the main ideas of the Integrity Measurement Architecture (IMA), which has been proposed for Linux-based systems, to a microkernel. In comparison, however, it significantly reduces the trusted computing base (TCB) and allows for a strict separation of the integrity verification component from any rich operating system, such as GNU/Linux or Android, running in parallel.

Abstract

We then enhance our system architecture with a multi-context hardware security module (HSM) based on a TPM-inspired design, which enables integrity verification, anomaly detection, and efficient lightweight attestation of multiple separated tasks. Our attestation mechanism, which we formally verified using *ProVerif*, implicitly proves the integrity of multiple tasks, efficiently communicates the result to a remote verifier, and enables a secure update and recovery protocol without the need for digital signatures. Finally, we enhance our implicit attestation mechanism with TPM 2.0 policies, which can be specified by the verifier as well as the prover and are verified and enforced by a TPM instead of the host operating system. As a result, both parties have to cooperate for a successful attestation, which implicitly creates verifiable proof of the prover's trustworthiness using mainly symmetric operations instead of expensive asymmetric cryptography.

Kurzfassung

Während eingebettete Systeme nahezu allgegenwärtig sind und ständig neue Hardware- und Softwarefeatures gewinnen, wächst ihre Komplexität und Code-Basis weiter an. Infolgedessen steigt die Angriffsfläche und damit die Wahrscheinlichkeit erfolgreicher Angriffe im Allgemeinen an, was nicht nur Anwender gefährden kann, sondern auch Unternehmen, wie z. B. Mobilfunknetzbetreiber. So stellen zum Beispiel „Distributed Denial-of-Service (DDoS)"-Angriffe von kompromittierten mobilen Geräten eine große Bedrohung für mobile Netzwerke dar. Das ist der Grund, warum wir einen Mechanismus auf der Grundlage eines Trusted Platform Module (TPM) präsentieren, der es einerseits mobilen Geräten ermöglicht zu beweisen, dass ihre Baseband-Software immer noch vertrauenswürdig ist, und der es andererseits dem Netzwerk erlaubt, für das mobile Endgerät beim Aufbau der Verbindung zum Mobilfunknetz eine bestimmten Baseband-Version zu erzwingen. Unser Ansatz ist somit eine effektive Methode, durch Attestierung Geräte mit kompromittierter Baseband-Software zu blockieren und dadurch DDoS-Angriffe gegen Mobilfunknetze zu reduzieren. Im Gegensatz zur traditionellen Attestierung, wie sie von der Trusted Computing Group (TCG) spezifiziert wird, ermöglicht unser *impliziter Attestierungsmechanismus* jedoch leichtgewichtige Attestierungsprotokolle, da er auf symmetrischen Verschlüsselungsoperationen und hashbasierten Nachrichtenauthentifizierungscodes anstelle von asymmetrischer Kryptographie, insbesondere digitalen Signaturen, und umfangreichen Logdaten beruht. Daher eignet sich unser impliziter Attestierungsmechanismus besonders für ressourcenschwache eingebettete Systeme.

Um die Sicherheit weiter zu verbessern, präsentieren wir eine Systemarchitektur auf der Basis eines Mikrokernels, der nicht nur deutlich kleiner, sondern auch weniger komplex ist als ein monolithischer Kernel wie z. B. Linux. Während die meisten Mikrokernel-basierten Systeme alle nicht-essentielle Dienste als *user-space* Tasks implementieren und diese während der Laufzeit strikt trennen, verlassen sie sich zumeist auf eine statische Konfiguration, um Sicherheit zu gewährleisten. Auch wenn dies unseren impliziten Attestierungsmechanismus begünstigt, bedeutet dies nicht unbedingt Vertrauenswürdigkeit. Aus diesem Grund kombinieren wir unsere Mikrokernel-basierte Systemarchitektur mit einem TPM und präsentieren einen Mechanismus zur Verifikation der Integrität, der in der Mikrokernel-gestützten Ausführungsumgebungen implementiert ist und vor dem Laden von (ausführbaren) Dateien Integritätsmesswerte z. B. für eine Attestierung berechnet.

Kurzfassung

Damit ist unser Ansatz einer der ersten, der die Grundgedanken der Integrity Measurement Architecture (IMA), die für Linux-basierte Systeme entwickelt wurde, auf einem Mikrokernel umsetzt. Im Vergleich reduziert unser Ansatz jedoch deutlich die trusted computing base (TCB) und ermöglicht eine strikte Trennung der Softwarekomponenten für die Integritätsverifikation (z. B. von Betriebssystemen wie GNU/Linux oder Android, die gegebenenfalls parallel in einem eigenen Teil des Systems laufend).

Anschließend erweitern wir unsere Mikrokernel-basierte Systemarchitektur um ein Hardware-Sicherheitsmodul (HSM) mit mehreren kryptographischen Kontexten, das vom Design eines Trusted Platform Module (TPM) inspiriert ist und die Integritätsüberprüfung, Anomalieerkennung und effiziente leichtgewichtige Attestation mehrerer getrennter Mikrokernel-Tasks ermöglicht. Unser Attestierungsmechanismus, dessen wesentliche Sicherheitseigenschaften wir mit *ProVerif* formal verifiziert haben, ermöglicht die Integrität mehrerer separate Tasks implizit zu beweisen, das Ergebnis auf einfache Weise an einen Prüfer (*Verifier*) zu kommunizieren und ein sicheres Update- und Recovery-Protokoll ohne digitale Signaturen umzusetzen. Abschließend setzen wir unseren impliziten Attestierungsmechanismus auch mit TPM 2.0 Policies um, die sowohl vom *Verifier* als auch vom Geprüften (*Prover*) spezifiziert werden können und von einem TPM anstelle des Host-Betriebssystems verifiziert und durchgesetzt werden. Dies führt dazu, dass beide Parteien für eine erfolgreiche Attestierung, bei der der Nachweis der Vertrauenswürdigkeit des geprüften Systems implizit durch (vorwiegend) symmetrische Operationen anstelle von aufwendiger asymmetrischer Kryptographie generiert wird, kooperieren müssen.

Acknowledgements

In retrospect, the time I have spent as a doctoral candidate at *Fraunhofer AISEC* and the *Technische Universität München* has been very exciting, sometimes challenging, but ultimately highly educating and immensely rewarding on a personal level—mainly because of the people I met along the way. That is why I would like to take the opportunity to thank the people both in my academic as well as my personal life, who have been supporting my research and the path I chose.

First and foremost, I would like to express my deepest gratitude to Prof. Dr. Claudia Eckert for her excellent supervision in a very personal, relaxed, and cooperative atmosphere as well as her professional guidance, especially in the final phase of my doctorate. Even in difficult times, I could always count on her continued support, for which I am immensely grateful.

I am also very grateful to Prof. Dr.-Ing. Georg Sigl for his advice, support, and for the discussions during my years of research. Further, I would like to thank Prof. Dr. Uwe Baumgarten for being the second reviewer and examiner of this thesis.

I would also like express my gratitude to my colleagues at Fraunhofer AISEC who have inspired and supported my research. In particular, I would like to personally thank Dr. Frederic Stumpf, Dr. Christoph Krauß, Dr. Michael Weiß, as well as all other current and former members of the department *Secure Operating Systems*, including those colleagues in the predecessor departments *Embedded Security and Trusted OS* and *Embedded Software Security*.

In addition, I would like to personally thank my colleagues at the *Chair for IT Security* in the *Department of Informatics* at the Technische Universität München. My special thanks go to Dr. Thomas Kittel as well as Sergej Proskurin, who has supported parts of the research conducted for this thesis as a student research assistant for several years.

Last but not least, I would like to sincerely thank my parents, my sister, and the rest of my family for their unconditional love, constant encouragement, and continuous support in my academic and my personal life. Most certainly, this thesis would not have been possible without them. I dedicate this thesis to them.

Acknowledgements

Contents

List of Figures

List of Listings

List of Listings

List of Tables

List of Tables

1

Introduction

For decades, most computers used to be rather large, in the early days even room-sized[1] machines, which provided only a very limited set of features and required highly skilled operators to execute even the most basic programs by today's standard. It took years of research in miniaturization, performance increases, and advanced software engineering to develop modern computer systems as we know them, which have an incredible number of software and hardware features. As a result, modern systems enable regular users, professionals, companies, and governments to run software ranging from casual apps to highly complex programs. Without this research and development, small user-customizable cellular phones, the Internet of Things (IoT), airplanes with fly-by-wire technology, self-driving cars, and industrial control systems with a large array of tiny sensors connected and managed via the Internet would not have been possible.

Although there are still mainframe systems, super-computers, and other special-purpose systems, e.g., for large data centers, which even look like their predecessors, the majority of today's modern computer systems are small, *embedded systems*[2], which perform a dedicated function [Gan03]. Hence, it is not surprising that these embedded systems have found their way into our daily lives as they change the way we view computers. Interestingly, however, most people do not even realize how much they use, trust, and sometimes depend on such small, almost 'invisible' systems, whether it be while they take a plane or drive a modern car[3].

1 Examples include the Zuse Z3 (1941), the Harvard Mark I (1944), or the ENIAC (1946) [Com15].

2 The BMBF estimated in 2006 that more than 90 percent of existing computer systems are embedded systems [Bun06]. Gartner forecasts 8.4 billion connected devices will be in use in 2017 (up 31 percent from 2016) [Gar17]

3 A recently manufactured high-end cars can have more than 70 electronic control units (ECUs) [Wol07].

In contrast to general-purpose systems, such as personal computers (PCs), embedded systems are often defined as "a combination of computer hardware and software, and perhaps additional mechanical or other parts, designed to perform a dedicated function" [Gan03]. Those information processing systems are, in turn, embedded into a larger system or device [cf. Bar06; Mar17; Osh13], e.g., as an ECU in a car. Other examples of embedded systems include cell phones and similar mobile devices, heart monitors, engine controllers, traffic lights, thermostats, pacemakers, or blood gas monitors [Gan03], but also include those larger systems, such as cars, trains, planes, as well as telecommunication and fabrication equipment [Mar17].

As specialized computer systems, which are often integrated as a part of a larger system and "designed to form computational engines that will perform specific tasks" [Osh13], most embedded systems are constraint in their application. While desktop systems are designed to perform general-purpose functions, embedded systems are "built to control a function or a range of functions" [Hea02]. Hence, they are not supposed to be programmed by the end user in the same way a PC is, i.e., the user may be able to make choices concerning functionality, but cannot easily change the functionality by adding or replacing software. In contrast, the software on a PC, which may function, for example, as word processor or games machine, can be changed easily [cf. Hea02]. As a result, the design of embedded systems is usually characterized by the intended application, which often also determines factors like hardware features, power consumption and cost.

Similar to general-purpose computers, the main components of embedded systems are the *processor*, *memory*, and *peripherals* as well the *software*. A *processor* for an embedded system is usually designed and later specifically selected for the intended function of the system in order to optimize, for example, power consumption and costs. Consequently, most embedded systems do not necessarily have the most recent and powerful hardware features that are available for general-purpose device, such as PC workstations or laptop computers. In recent years, however, features like hardware-protected execution environments have been made available to more and more low-cost systems, which enables more secure system designs.

In addition, the size of available *memory* has increased over time. A few years ago, a large portion of embedded systems were only able to store the software they were designed to run. Nowadays, even simple embedded systems, such as wireless sensor nodes or IoT devices, are able to collect sensor data and store those values until they are transferred to a backend system.

Besides the processor and memory, embedded systems are typically equipped with various *peripherals*. Characteristically, most embedded systems have input peripherals, such as sensors, but also a number of output peripherals, which can include binary, serial, and analog outputs [cf. Hea02]. Furthermore, embedded systems might be equipped with LED outputs and a display.

The *software* of a typical embedded systems includes initialization code and configurations, a boot loader, an operating system (OS) and/or a runtime environment, applications as well as error handling, as well as optional debug and maintenance code [cf. Hea02].

As those embedded computer systems become part of our daily lives and provide more and more features, both in hardware and software—a trend that is likely to continue—, they become immensely complex and often require a large code base. Unfortunately, it is a well-known fact that with growing complexity and code size, the probability of bugs and errors significantly increases[1]. As a consequence of the increasing complexity and growing code base, most systems are and will be vulnerable to unexpected failure and malicious attacks, which exploit bugs in existing functionalities. This troubling fact is especially dramatic and fatal for monolithic systems, such as Linux systems, where a multitude of functionalities is combined in a large kernel running in privileged mode and a malicious modification of a privileged component, such as a kernel module, can compromise or crash the entire system. However, there are alternative system architectures, e.g., based on a microkernel, which use separation mechanisms to overcome this issue (or at least limit the effects).

A microkernel requires a fraction of the code of a monolithic kernel and, hence, relies on less privileged code, which reduces the complexity and the attack surface, especially of privileged parts. Since microkernel-based systems compartmentalize functionality and implement all non-essential system services in user space, they enable a better recovery after partial system failure or an attack. That way, a microkernel is perfectly suitable, for example, to act as a small core for a virtual machine monitor (which runs in user space) and enable the virtualization of another OS, such as a Linux-based OS like Debian or Android. By virtualizing the rich OS—for example, using paravirtualization or hardware-assisted virtualization—, a microkernel-based system can severely limit the effects of an attack, e.g., against the monolithic kernel of the rich OS, on the overall system.

Unfortunately, most microkernel-based systems do not support existing implementations for calculating software integrity measurements, such as the Integrity Measurement Architecture (IMA) for Linux, which is not compatible with microkernels. In fact, integrity verification[2] and remote attestation mechanisms for microkernel-based systems do not exist in the way they do for Linux. Although the basic concepts can be applied to a system based on a microkernel, the limitations and restrictions of a microkernel and a common embedded system architecture are quite significant. For example, without secure storage, e.g., provided by a hardware security module (HSM), such as a TPM, the integrity measurements can be easily modified, even if the system is based on a microkernel. Unfortunately, most systems—especially embedded systems—are not equipped with a security-enhanced chip(set) or an HSM. And even with an HSM or a TPM, security protocols designed to securely communicate the integrity measurements to a remote verifier are often avoided. The reason is that these *remote attestation* protocols are often considered (too) complex, usually require expensive asymmetric cryptographic operations[3], and are hence inefficient, which might make them unsuitable for resource-constraint embedded or IoT systems.

1 In fact, that is one reason why some systems execute critical software components on separate hardware resources.
2 In this context, verification means cryptographic validation, not a verification with formal methods.
3 This statement compares asymmetric with symmetric cryptography.

1.1 Problem Statement

As the number of embedded systems, which control safety- and security-critical processes and operations, grows and their complexity and code base increase, it becomes crucial that there exist cryptographic methods to verify their integrity and trustworthiness. One approach, which, however, is only one possible solution, is based on Trusted Computing (TC), specifically *authenticated boot* and *remote attestation* in combination with a hardware-based secure element (SE), the TPM.

For a remote attestation as specified by the Trusted Computing Group (TCG), hash values of software components, so-called *integrity measurements*, which are stored in special-purpose registers inside the TPM, are signed with an asymmetric key generated and protected by the TPM. The integrity measurements are calculated during authenticated boot where the current component in the boot chain hashes the next one before executing it and then they are sent to the remote verifier together with a measurement log containing additional details about individual measurements. By comparing those integrity measurements with trusted reference values, aggregating them, and verifying the digital signature, the attestor can determine the trustworthiness of a remote system. However, it is not always desirable, necessary, or possible to create digital signatures and store the full details about all integrity measurements in extensive logs for a subsequent comparison with trusted reference values, especially with respect to resource-constraint devices. As a consequence, we explore *implicit attestation mechanisms*, which do not rely on expensive asymmetric cryptographic, but use symmetric cryptographic operations to prove the trustworthiness of a system without the need for extensive measurement logs. Since symmetric cryptography is more efficient compared to asymmetric cryptographic operations, we expect that this lightweight approach can be integrated more easily into existing security protocols, e.g., for authentication, secure access, and code updates.

Furthermore, authenticated boot—as the name suggests—only measures the boot chain and does not prevent the execution of (potentially malicious) binaries after the boot process has completed. To overcome this limitation, IMA for Linux-based systems calculates integrity measurements during runtime whenever a binary is loaded. However, since systems with monolithic kernels, such as Linux, are difficult to evaluate using IMA and remote attestation because of their complexity, we explore the use of microkernel-based systems architectures. Microkernels are smaller in terms of code size, hence less complex, and by implementing all non-essential system components in separate user space tasks, microkernel-based systems have smaller trusted computing base (TCB). By reducing the TCB [cf. Def85; Lam91; Rus81], the integrity measurements are more expressive and the attestation is not only simplified but also more focused on the relevant system components. Unfortunately, IMA focuses on Linux-based systems, which means it is not compatible with a microkernel, and does not prevent loading of remote binaries from an unknown source. For that reason, we will propose similar measurement architecture for microkernel-based systems enabling a secure loading mechanism, which makes sure that the integrity of our system is not compromised after a successful remote attestation and the execution of a remote binary provided by the verifier.

Lastly, since modern computer systems—including embedded systems—increasingly support hardware-assisted virtualization, TPM-based remote attestation faces new challenges, because current TPMs only provide limited native support for virtualization. One major limitation concerns the number of available cryptographic contexts. Due to hardware constraints, TPMs usually have, for example, only one set of registers to store integrity measurements. In a system with virtualization, however, it would be desirable to have a set of these registers for each virtual machine (VM) in order to separately store integrity measurements for different VMs. As a result, systems featuring virtualization currently implement virtual TPMs in software in order to provide each VM with its own TPM, although the concept could be realized within dedicated hardware security modules. For that reason, we explore how multiple cryptographic contexts can improve remote attestation protocols and in what way our mechanism benefits from such a multi-context HSM. In addition, we discuss how a Trusted Execution Environment (TEE) and cryptographic policies can further assist in improving TPM-based remote attestation in a system featuring virtualization.

In summary, we focus on research in attestation mechanisms and protocols for embedded systems, which are suitable for resource-constraint devices and are based on a secure microkernel-based system architecture that make use of hardware-based isolation and takes advantage of a simplified and reduced TCB. In addition, the attestation mechanisms are aimed to be efficient, lightweight, and suitable for systems with virtualization. Furthermore, we explore how our attestation mechanism can be integrated into existing protocols, e.g., for authentication or secure code updates, and how TPM policies can make our implicit attestation beneficial for both the verifier and the prover.

1.2 Contributions

The main contributions of this thesis are outlined in the following paragraphs:

Implicit Attestation Mechanism based on Efficient Symmetric Cryptography

While traditional remote attestation as specified by the TCG relies on expensive asymmetric cryptography, we provide an attestation mechanism that is based on efficient symmetric operations, in particular hash functions and message authentication codes, particularly *hash-based message authentication code (HMAC)*. At the same time, our work introduces the notion of *implicit attestation*, which communicates the cryptographic proof to infer knowledge about the trustworthiness of the prover's system without requiring digitally signed integrity measurements and a corresponding measurement log with the entire history of the measurement process. Instead, our approach provides a method to implicitly generate knowledge about the trustworthiness of the prover's system and enables the remote verifier to infer security properties of the system. As a result, our attestation mechanism does not require digitally signed integrity measurements, but relies on symmetric cryptography to establish knowledge about trustworthiness, which makes the approach more efficient, lightweight, and hence suitable for resource-constraint embedded systems.

Microkernel-based System Architecture with a TPM

As a basis for our attestation mechanism, we propose a detailed system architecture with a TPM, which is based on a microkernel that only has a fraction of the code size of regular monolithic kernel like Linux and thus significantly reduces the complexity and the trusted computing base. In addition, microkernel-based systems implement all non-essential system services, e.g., drivers, as user-space tasks, strictly separate those tasks, and provide only a small number of system calls.

Combination of Attestation with Authentication, Secure Loading, and Update Protocols

Since remote attestation is not a stand-alone, autotelic[1] security procedure or protocol, but rather assists in making a reasonable decision about the trustworthiness of a remote system, our research also focuses on the integration of attestation into existing protocols. In this thesis, we will therefore explore and show how implicit attestation can be used in combination with authentication protocols. In addition, our joint work with Weiss et al. [Wei14] demonstrates how implicit and local attestation can enable sophisticated security-sensitive operations, such as secure loading of remote binaries. We also show how attestation can assist in securely updating existing code on a remote system.

Lightweight Attestation for Multiple Cryptographic Contexts

Although the TCG specifies methods to attest virtualized systems, which include deep attestation of the underlying hypervisor and the virtual machines, traditional remote attestation suffers from the limitations of the TPM, more precisely the TPM 1.2, which only provides one cryptographic context. In one of our main contributions, we present a system architecture with a multi-context HSM with a TPM-like architecture and firmware design, which enables a remote attestation of multiple tasks with distinct cryptographic contexts. This part of the thesis, which focuses on both hardware and software aspects of a remote attestation for multiple contexts, highlights the necessary features in a future hardware security module, which should be designed to support virtualization natively.

Policy-based Implicit Attestation with Mutual Benefits

With increasing interest, availability, and adoption of the TPM 2.0, the successor to the TPM 1.2, a particular contribution of this work consequently focuses on the new TPM features, especially TPM 2.0 policies. Based on the prior research conducted for this thesis, we propose and implement an "extension" of the implicit attestation mechanism to utilize and incorporate TPM 2.0 policies. As a result of this work, we provide an attestation mechanism for microkernel-based embedded systems, which takes advantage of hardware-assisted virtualization, a hardware TEE, and a TPM 2.0. By using TPM 2.0 policies, we can ensure that these policies are enforced by the TPM and do not have to rely on the operating system of the host. Furthermore, the attestation mechanism enables the verifier as well as the prover to specify policies that need to be satisfied by the other party.

1 The word "autotelic" comes from the Greek word αὐτοτελής (autotelēs), which combines αὐτός (autos, "self") and τέλος (telos, "goal"), and basically means that something has "a purpose in and not apart from itself" [Mer17].

Publications

Parts of the contributions mentioned above are published in the following scientific, peer-reviewed articles, which also include additional contributions that are not covered in detail in this thesis:

[Wag12a] Wagner, Steffen, Christoph Krauss, and Claudia Eckert: "T-CUP: A TPM-Based Code Update Protocol Enabling Attestations for Sensor Networks." *Security and Privacy in Communication Networks: 7th International ICST Conference, SecureComm 2011, London, UK, September 7-9, 2011, Revised Selected Papers*. Ed. by Rajarajan, Muttukrishnan, Fred Piper, Haining Wang, and George Kesidis. Vol. 96. LNICST. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012: pp. 511–521

[Wag12b] Wagner, Steffen, Sascha Wessel, and Frederic Stumpf: "Attestation of Mobile Baseband Stacks." *Network and System Security: 6th International Conference, NSS 2012, Wuyishan, Fujian, China, November 21-23, 2012. Proceedings*. Ed. by Xu, Li, Elisa Bertino, and Yi Mu. Vol. 7645. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012: pp. 29–43

[Wei14] Weiss, Michael, Steffen Wagner, Roland Hellman, and Sascha Wessel: "Integrity Verification and Secure Loading of Remote Binaries for Microkernel-Based Runtime Environments." *2014 IEEE 13th International Conference on Trust, Security and Privacy in Computing and Communications*. Sept. 2014: pp. 544–551

[Wag15] Wagner, Steffen, Christoph Krauss, and Claudia Eckert: "Lightweight Attestation and Secure Code Update for Multiple Separated Microkernel Tasks." *Information Security: 16th International Conference, ISC 2013, Dallas, Texas, November 13-15, 2013, Proceedings*. Ed. by Desmedt, Yvo. Vol. 7807. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2015: pp. 20–36

[Wag16a] Wagner, Steffen and Claudia Eckert: "Policy-Based Implicit Attestation for Microkernel-Based Virtualized Systems." *Information Security: 19th International Conference, ISC 2016, Honolulu, HI, USA, September 3-6, 2016. Proceedings*. Ed. by Bishop, Matt and Anderson C A Nascimento. Cham: Springer International Publishing, 2016: pp. 305–322

In addition, we have published and open sourced a script to extract the code of a fully functional TPM 2.0 simulator, which is included in the public PDF version of the TCG TPM 2.0 Library Specification [Tru14; Tru16], as part of this work. The script, which hopefully enables other researchers to start using a TPM and develop TPM-based security protocols more easily, can be found on GitHub:

[Wag16b] Wagner, Steffen, Sergej Proskurin, and Tamas Bakos: *TPM 2.0 Simulator Extraction Script*. `https://github.com/stwagnr/tpm2simulator`. Jan. 2016

1.3 Outline

In Chapter 2, we provide an overview of basic TC concepts and relevant aspects of secure system architectures, which will be the basis for our integrity verification and remote attestation protocols. Specifically, we discuss the security capabilities and features of a TPM as well as related concepts like authenticated boot, remote attestation, and trusted network connect. Furthermore, we describe concepts and technologies, such as separation, isolation, virtualization, and security extensions, which enable the implementation of secure system architectures as a basis for remote attestation.

In Chapter 3, we discuss related work on TC-based integrity measurement and verification concepts as well as existing attestation protocols, such as traditional remote attestation as specified by the TCG. In particular, we contrast existing remote attestation protocols, which usually send integrity measurements signed by the TPM to a remote verifier, to our implicit attestation technique. In contrast to most existing attestation mechanisms, which often rely on expensive asymmetric cryptography, our lightweight implicit attestation uses symmetric operations to create verifiable proof of the system's trustworthiness, which make it suitable for resource-constraint devices.

In Chapter 4, we present four different attestation scenarios, which focus on specific aspects of secure access to trusted (remote) resources, e.g., files on a corporate network or sensor data produced by an industrial control system. In addition, we specify an attacker model for the scenarios, our remote attestation, and the derived protocols, which include secure loading and code updates. In the following chapter, we use the model in our security discussions and evaluations.

In Chapter 5, we describe our comprehensive system architecture, which is comprised of various protection domains and serves as the basis for our implicit attestation mechanism. More precisely, we present a flexible system architecture with an application processor and a baseband processor, a hardware security module, and multiple separate execution environments for software components with different criticality. As we focus on specific parts and different aspects of our system architecture during the course of this thesis, we continually improve our architecture by extending and describing the relevant protection domains and execution environments, such as the TEE, in more detail.

In Chapter 6, we start by exploring the integrity verification and remote attestation of systems with a baseband stack, such as mobile phones, since these software stacks are usually executed on dedicated baseband processors, which often act as a system master, especially in mobile devices. Although mobile phones already take advantage of a smart card to securely authenticate towards the backend systems of the network operator, we propose the use of a TPM to establish and communicate the integrity and trustworthiness of the baseband software stack to a remote verifier. This mechanism allows the network operator to evaluate and verify the trustworthiness of a critical part of a mobile system before granting access to resources within the network. In particular, we present an attestation protocol for the baseband stack, which uses efficient symmetric operations and integrates into existing protocols used to authenticate mobile devices towards a network operator.

In Chapter 7, we shift our focus to the main application processor and describe our approach to adapt and extend the well-known Integrity Measurement Architecture for Linux to a microkernel, which is one key element of our secure system architecture. Since microkernels are generally very small in terms of code size and significantly less complex compared to monolithic kernels like Linux, integrity measurement and attestation services only rely on a very small and robust TCB. Furthermore, our approach for adapting and extending IMA enables a remote party to securely load external binaries into an isolated execution environment, which is separated by the microkernel from a rich operating system that is virtualized and executed on the same hardware. Finally, this chapter also presents details about our prototype implementation and discusses our evaluation, especially with respect to security, code size, and performance.

Based on the previous results, Chapter 8 extends our microkernel-based system architecture with additional security features, such as the possibility to collect events generated by an anomaly detection component and stored in a custom multi-context HSM with a TPM-like firmware design. By enabling the collection of integrity measurements as well as runtime anomaly detection events in a multi-context HSM, we propose and describe an approach for an implicit attestation of multiple separated microkernel tasks with distinct cryptographic contexts that integrates anomaly detection. The proposed attestation protocol thereby extends and advances the notion of implicit attestation, which does not require to digitally sign PCR values, but implicitly proves the trustworthiness to a remote verifier through efficient symmetric cryptographic operations, which is expected to be particularly important for system featuring virtualization.

In Chapter 9, we build on the results of the research discussed in the previous chapters and present a policy-based implicit attestation mechanism for microkernel-based systems with a hardware TEE. Similar to the previous chapter, where we integrated anomaly detection events, this attestation mechanism shows a method to integrate policies into the attestation protocol by taking advantage of TPM 2.0 authorization policies. As a result, this variation of implicit remote attestation focuses on security technologies, such as a hardware TEE and also hardware-assisted virtualization, and shows the benefits of combining those hardware-based separation mechanisms with security protocols like TPM-based remote attestation.

In Chapter 10, we conclude this thesis by illustrating and summarizing how the contributions presented above result in an efficient, lightweight remote attestation suitable for embedded systems. In addition, we draw conclusions based on the research conducted during the course of this thesis, in particular related to hardware-based remote attestation. Finally, this chapter also highlights potential directions for future research in the area of integrity verification and hardware-based remote attestation.

Introduction

2

Background

In this chapter, we provide relevant background information that serves as a comprehensive basis for the following chapters and the research conducted in this work. In particular, we give an overview of the ideas and concepts behind Trusted Computing and explain how TC aims to enforce that a system consistently behaves as expected and, hence, can be considered trusted. We also introduce complementary concepts as well as software- and hardware-based technologies designed to develop trustworthy and secure system architectures. Since we take advantage of those existing concepts, e.g., microkernel-based system designs, and utilize them as a basis for our own contributions, we not only describe their specific characteristics, but also highlight their advantages and benefits, especially in relation to our contributions.

This chapter is structured as follows. Section 2.1 presents the ideas and goals of TC, describes the TPM as one of the main cornerstones of TC, and briefly contrasts the differences between the TPM 1.2 and the TPM 2.0. Based on the overview, this section subsequently describes fundamental TC concepts, such as *authenticated boot* and *remote attestation*, which are relevant for our research. Following the introduction and overview of TC, Section 2.2 presents additional concepts and available technologies to develop a secure system architecture for embedded devices. More precisely, it describes theoretical concepts, such as *isolation* and *separation*, as well as existing technologies like *ARM Virtualization* and *Security Extensions* [ARM12; ARM09; ARM10].

2.1 Trusted Computing

The term Trusted Computing is based on the more general notion of *trusted systems*, which describes and assess systems that—to a specified extent—can be relied upon to behave in a specified way and enforce a specified set of (security) rules and policies. In this context, TC defines specific requirements for a system and specifies the hardware, software, and protocols to enable the system to behave according to specified security policies. More specific, the TC components ensure that the system will consistently "behave in the expected manner" [cf. Mit05; Tru16, *Root of Trust*], which is the basic definition for *trust* in Trusted Computing.

To enforce consistent system behavior, the Trusted Computing Group, a consortium of companies, research institutions, and governments, which was formed in 2003 as a successor to the Trusted Computing Platform Alliance (TCPA), collaboratively specifies the TC technology comprising of both hardware and software components, particularly the Trusted Platform Module. The TPM, which is usually a non-programmable hardware-based security module, but can be a software implementation in a secure runtime environment, is the cornerstone of this specification effort. Together with software, such as the TCG Software Stack (TSS), and protocols like authenticated boot and remote attestation, the TPM can enforce a specified behavior and ensure that the system acts according to its policies.

2.1.1 Motivation and Goals

As the importance of computer security has been increasing since the mid-1990s and the number of vulnerabilities that have been discovered and are exploited grows daily, TC aims to provide mechanisms to improve the security of personal, off-the-shelf as well as corporate, server-oriented computer platforms. The reason is that the hardware and software of regular platforms (without TC technology) usually provide insufficient protections, because they are usually highly complex and often lack effective isolation mechanisms. As a result, their system software is usually susceptible to attacks, such as malicious modification of boot components. Consequently, TC aims to improve the security of computing platforms, for example, by reducing their TCB and isolating critical functions. In addition, the goals of TC include compatibility with existing commodity systems, both hardware and software, the ability to reuse existing components, such as the OS, and an open architecture (at least regarding the standards and specifications). These goals aim to enable a user or a remote party to reason about the trustworthiness of a system that is enhanced with TC technology.

To achieve the goals of TC, the TCG proposes to solve some of the problems with an addition security component, the TPM, which acts as a trust anchor and helps to create a trusted platform. The main idea of the TCG's approach is to establish trust by relinquishing control to the TPM, i.e., by transferring control to TC hardware and software. Since those TC components are outside of the control of the regular system (including any malicious code) and only allow specified operations, the user as well as any remote party can expect a specified behavior, i.e., trust the system.

Obviously, this type of technology, which forces the owner of a system to relinquishing control (at least to a certain degree), can raise suspicion. Since TC can potentially be used to implement digital rights management (DRM), restrict the freedom of the owner's choice, e.g., regarding the software on the device, or violate privacy, the applications of TC technology can be controversial. As a concept, however, TC provides security benefits and, hence, can enable the design of a more secure system. As such, we use the TC technology, especially the TPM, as a basis for our integrity verification and remote attestation mechanisms, which help to assess and verify our system.

2.1.2 Trusted Platform Module

Usually, a Trusted Platform Module is described as a microcontroller with additional security features similar to a smart card, such as a subscriber identity module (SIM). However, a TPM is actually a specification, more precisely an international standard[1], for a security component, which acts as a hardware-based root of trust (or *trust anchor*) and has standardized features and interfaces. In fact, while a TPM 1.2 generally is a secure crypto-processor in a dedicated hardware component, such as a secure microcontroller, the specification of a TPM 2.0 explicitly allows for software-based implementations, in particular *firmware TPMs*, which use security features of the host's system, such as a TEE, to ensure their security guarantees.

In addition to dedicated TPMs and firmware implementations, there are also *integrated TPMs*, which are part of a hardware chip that also provides functions not necessary related to security, as well as *software TPMs*, which—as the name suggests—implement a TPM purely in software. While integrated TPMs are resistant to software bugs, but not tamper-resistant like dedicated TPMs, software and firmware-based TPMs are inherently less secure. Since software-based TPM implementations usually cannot be hardened against sophisticated physical and side channel attacks, hence lack tamper resistance, the majority of TPMs is predominantly hardware-based and built into PCs and servers during production and hardware assembly.

As dedicated security modules, which are part of the platform's hardware architecture, these hardware TPMs provides the highest level of security and can securely store and protect sensitive information, such as cryptographic keys, even against most physical attacks. Since TPMs are logically and physically linked to a platform (not to a person like smart cards), their main function is to secure the device by providing secure key storage, authentication services, and integrity measurements to the system. In particular, by collecting measurements of the system's integrity, TPMs can be used to implement security protocols, e.g., remote attestation, which will be described in Section 2.1.3 following an overview of a TPM's architecture, keys, and security-specific features.

1 The TPM 1.2 Main Specification (Revision 103) has been approved by JTC 1, a joint technical committee of the International Organization for Standardization (ISO) and the International Electrotechnical Commission (IEC), as international standard ISO/IEC 11889:2009 [ISO09] and the TPM 2.0 Library Specification (Revision 01.16) as ISO/IEC 11889:2015 [ISO15].

2.1.2.1 Architecture of a TPM

The basic architecture of a hardware TPM 1.2 (Figure 2.1) or TPM 2.0 (Figure 2.2), respectively, can be divided into two parts: one comprised of standard microcontroller components (gray) and one consisting of TPM-specific components (white). However, although the gray components are building blocks of a standard microcontroller, such as the CPU (central processing unit), input/output (I/O), random access memory (RAM), read-only memory (ROM), and EEPROM (electrically erasable programmable read-only memory), a hardware TPM as a security controller has special protection and anti-tamper mechanisms, e.g., internal encryption, power sensors, and shielding, which protect against most attacks including physical tampering, such as probing.

On the right, Figure 2.1 shows the main TPM-1.2-specific low-level hardware components, such as the cryptographic engines for *RSA (Rivest-Shamir-Adleman cryptosystem)* [C1] and the *Secure Hash Algorithm 1 (SHA-1)* [C5] as well as the random number generator (RNG) [C4] which is, for example, used by the key generation component [C2]. The TPM 1.2 components also include an HMAC engine [C3], which is for internal use only, and an integrated AES module, which is part of the cryptographic engine(s) and also not available externally. Additional components are a power detection module [C6], an opt-in component [C7], and an execution engine [C8] that handles commands. Furthermore, TPMs provide non-volatile (NV) memory [C9], e.g., for certain keys (cf. Sections 2.1.2.2 and 2.1.2.3), and volatile memory [C10] including PCRs, which can be used to collect and record integrity measurements and create a platform configuration in form of a continuous hash chain (see Section 2.1.2.4 and the following).

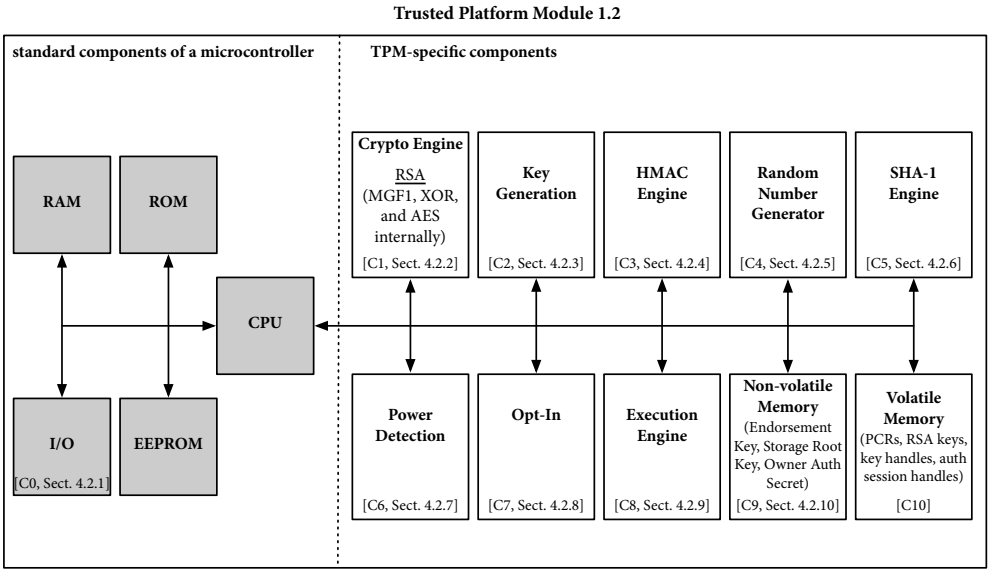

Figure 2.1: Architecture of a TPM 1.2 (based on and with references to [Tru11, Part 1, Figure 4:a])

In comparison, a hardware TPM 2.0 is also based on a secure microcontroller as shown on the left of Figure 2.2. However, in contrast to a TPM 1.2, the TPM components have changed significantly, for example, because the TPM 2.0 supports *algorithm agility*. The main idea of algorithm agility (or *crypto agility*) is to include the capability into the TPM specification to support a set of algorithms rather than only one cryptographic system, e.g., RSA (and SHA-1), like the TPM 1.2 has mandated. As a result, the architecture of a TPM 2.0 comprises, for example, (multiple) hash engines, which might provide SHA-1, but also support *Secure Hash Algorithm 2 (SHA-2)* with different sizes for the digest or any other approved hash algorithm. As a consequence, some TPMs have generic PCRs to support different algorithms, while others have multiple *PCR banks*, where each bank implements its own algorithm. Similar to the hash engines, a TPM 2.0 provides asymmetric engines, e.g., for RSA and elliptic curve cryptography (ECC), and symmetric engines, e.g., for AES, which can even be available externally. The engines are directly connected to the key generation component which, in turn, utilizes the RNG that provides the necessary entropy for cryptographic key.

While the rest of the components shown on the right of Figure 2.2 resemble their counterparts in a TPM 1.2, there are also some new modules, such as the authorization and management components. The authorization components is responsible for new TPM 2.0 authorization mechanisms, which are no longer limited to weak authentication methods (cf. Section 2.1.2.5, *Enhanced Authorization*). The management module handles new features, such as *field upgrades* and the new TPM 2.0 *control domains*, where the TCG distinguish between *platform*, *owner*, and *privacy administrator* controls and separates TPM objects, such as cryptographic keys.

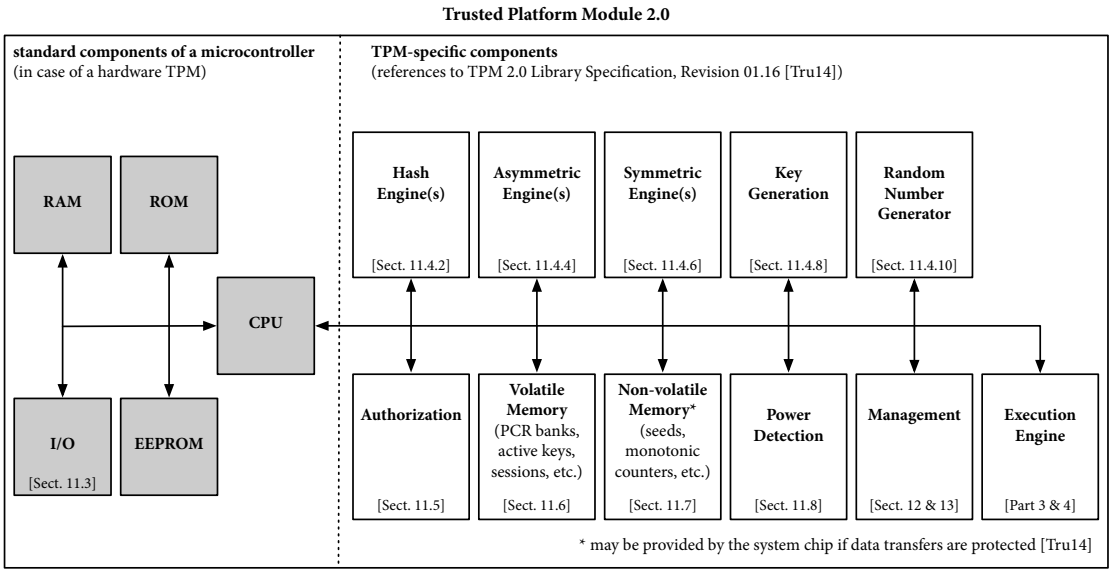

Figure 2.2: Architecture of a TPM 2.0 (based on and with references to [Tru16, Part 1, Figure 2])

2.1.2.2 Roots of Trust

To ensure that a platform behaves in an expected manner, TC requires an initial basis for trust, also referred to as *Roots of Trust*, which must be relied upon without further verification and validation, because modifications or misbehavior cannot be detected and remedied. In its specifications, the TCG postulates three Roots of Trust:

1. Root of Trust for Measurement (RTM)
2. Root of Trust for Reporting (RTR)
3. Root of Trust for Storage (RTS)

Each Root of Trust ensures that a particular TC primitive (integrity measurements, remote attestation, and secure storage for cryptographic keys) is based on a component, which can be relied upon. It is hence not surprising that the TPM implements most parts of the three Roots of Trust, RTM, RTR and RTS, and only partially relies on the host system for some aspects of the RTM.

Root of Trust for Measurement

The RTM is the basis for the collection of integrity measurements and the subsequent establishment of a platform configuration, which can be used and reported in a remote attestation (cf. Section 2.1.3). The measurement process starts with the Core Root of Trust for Measurement (CRTM), which is usually a piece of software that is executed immediately after the system starts booting. Although the CRTM is supposed to be immutable, it is usually part of the Basic Input/Output System (BIOS) or Unified Extensible Firmware Interface (UEFI) on x86-based systems[1]. After the CRTM has measured the BIOS, it stores the integrity measurements inside the PCRs of the TPM. Thus, the TPM implements the other half of the RTM, i.e., secure storage for integrity measurements.

Root of Trust for Reporting

The RTR, which is located in a shielded location of the TPM, is the root of trust for reporting the current state of the platform, i.e., the current integrity measurements stored inside the PCRs. To link the reported platform configuration and its integrity measurements to a particular system, each TPM has a so-called Endorsement Key (EK), which basically implements the RTR. In a TPM 1.2, the EK is a 2048-bit RSA key pair, which is usually generated during production, stored inside the non-volatile memory of the TPM, and originally could not be changed during the TPM's lifetime[2]. However, because of privacy concerns (and since the EK is not intended to directly encrypt or sign arbitrary data), it is not possible to report the current platform configuration by signing PCR values with the EK. Instead, attestation identity keys, which are described in Section 2.1.2.3, are used.

1 Despite the negative implications, the TCG allows this type of implementation as long as the update process is secure.
2 In the latest version of the specification [Tru11], a revokable EK may be created using `TPM_CreateRevocableEK`, which can then be deleted using `TPM_RevokeTrust`.

In contrast, a TPM 2.0 allows a so-called *privacy administrator*, which is often identical with the owner, to create a virtually unlimited number of EKs. Those EKs, which are based on a TPM-specific seed value (cf. Section 2.1.2.3, *Endorsement Primary Seed*), solve the privacy issue and remove the need for attestation identity keys. As a result, the RTRs of a TPM 2.0 is effectively a large random value, which is stored in a protected location inside the TPM and never leaves the TPM.

In comparison to a TPM 1.2, the RTR of a TPM 2.0 is not only the root of trust for reporting evidence of a platform configuration stored in PCRs. The TPM 2.0 can also use the RTR to certify audit logs and key properties (cf. [Tru16, Part 1, Section 9.4.3]). Although we do not use audit logs in this work, the possibility of certifying key properties might complement our protocols.

Root of Trust for Storage

The RTS, which is also stored in a shielded location of the TPM, is the root of trust for protecting keys that are stored on the host's system in encrypted form. Since the TPM can be trusted to prevent inappropriate access to its shielded memory, it can act as an RTS [cf. Tru16, Part 1, Section 9.4.2]. As such, the RTS is the anchor inside the TPM, which enables protected storage of encrypted keys and data outside the TPM and solves the problem of its limited internal storage capacities.

In a TPM 1.2, the RTS is realized as a 2048-bit RSA key pair, which is referred to as Storage Root Key (SRK). The SRK is generated when the owner of the TPM is created via `TPM_TakeOwnership`. As a root of trust, the private portion of the SRK never leaves the TPM, otherwise the whole key hierarchy would be compromised. The SRK is deleted (and the other keys invalidated), when the TPM is cleared using `TPM_Clear`.

In comparison, a TPM 2.0 allows the owner to create a virtually unlimited number of "SRKs", which are referred to as *Storage Primary Keys* and are based on a TPM-specific seed value (cf. Section 2.1.2.3, *Storage Primary Keys* and *Storage Primary Seed*). As a result, the RTS of a TPM 2.0 is effectively a seed value, which never leaves the TPM and is changed after each `TPM2_Clear`.

2.1.2.3 Key Types and Constraints

One of the main functions of a TPM is the secure generation and handling of different keys, which have a certain purpose and, hence, are used in specific cryptographic operations provided the TPM. Those cryptographic keys not only differ in their values and parameters, they also have specific types and constraints, which are specified by the TCG.

The type of a key usually defines, but also limits a key's usage. For example, an asymmetric TPM 1.2 signature key cannot be used as an HMAC or encryption key. In a TPM 2.0, types and constraints are defined by attributes, which include the intended use (e.g., signing or encryption), the cryptographic type (e.g., symmetric or asymmetric) and algorithm, such as AES, ECC, or RSA, as well as restrictions on use and key management [Art15, Key Types and Attributes, p. 125].

Furthermore, most TPM 1.2 keys can be *migratable* or *non-migratable*, which is a constraint on key management and usage. Migratable keys can be transferred from one TPM to a different TPM, e.g., for backup purposes, while non-migratable keys are restricted to a specific TPM, which means the private portion of the key never leaves that particular TPM. In a TPM 2.0, migratable keys are called *duplicable*, which emphasizes that a duplicable key can be copied to a different TPM and exist in two or more TPMs at the same time. Consequently, TPM 2.0 *Duplication*, which is described in detail in Section 2.1.2.5, is the process of securely migrating a duplicable key to a different TPM enabling, for example, key backups. We use duplication to securely migrate a key with specific attributes to a different TPM and, thereby, establish a shared cryptographic basis for our attestation protocol (see Chapter 9).

TPM 1.2 Key Types

The TPM 1.2 Main Specification [Tru11] defines the following key types (not including *authorization change* and *legacy keys*, which are not relevant for our work):

Storage Keys: A storage key is an RSA key pair with a minimum key length of at least 2048 bit. Storage keys can only be used locally by the TPM, may be protected with a password, and optionally cryptographically bound to certain PCR values. This type of key is used to create a key hierarchy starting with the SRK. Hence, storage keys mainly encrypt other TPM keys, such as binding or signature keys, which can then be securely stored on the host system. Storing keys externally is necessary because of the limited storage capacity of the TPM.

Binding Keys: A binding key is an asymmetric RSA key pair, which can be used by any platform or user to encrypt data that can only be decrypted by a TPM with the private portion of that particular binding key.

Signature Keys: Since the TPM 1.2 usually only supports RSA, signatures keys are asymmetric RSA key pairs that must have a specified length, usually 1024 or 2048 bit. As the name suggests, this type of key can be used to sign arbitrary data provided to the TPM by the host system. Signatures keys can be either migratable or non-migratable.

Identity Keys: For security reasons and due to privacy concerns, the EK can not be used directly to calculate signatures [cf. Tru11, Section 11.4]. Instead, an Attestation Identity Key (AIK) provides an alias for the EK, i.e., acts as a pseudonym, and can be used to sign certain TPM data structures. As such, AIKs are special-purpose non-migratable signature keys.

In addition to those keys, the TPM 1.2 dynamically calculates (session) secrets, which are used during the authorization protocols between the host system and the TPM. These TPM 1.2 protocols are called Object-Specific Authorization Protocol (OSAP) and Object Independent Authorization Protocol (OIAP). They basically create secure sessions for authorizations, which ensure authenticity, confidentiality, and integrity of the data transferred between the host and the TPM.

TPM 2.0 Key Types

The TPM 2.0 Library Specification [Tru16] defines the following three main types of keys:

Primary Keys: All Primary Keys are mainly derived from one of the three TPM Primary Seeds (PS): the Platform Primary Seed (PPS), the Storage Primary Seed (SPS), or the Endorsement Primary Seed (EPS). A Primary Key is thereby always associated with corresponding hierarchy of the Primary Seed, i.e., the Platform Hierarchy (PH), Storage Hierarchy (SH), or the Endorsement Hierarchy (EH). In addition, a Primary Key never leaves the TPM.

Ordinary Keys: In comparison to Primary Keys, Ordinary Keys are regular keys in one of the three TPM 2.0 hierarchies. They are created underneath a Primary Key (or another Ordinary Key), which acts as the parent key. Ordinary Keys are seeded with entropy from the TPM's RNG and can be duplicable, hence, migrated to a different TPM.

Derived Keys: While Ordinary Keys use entropy from the RNG, Derived Keys are generated using the key derivation function *KDFa* and inputs like the *sensitive* value of the Derivation Parent.

Based on those main key types, the configuration of key attributes determines the properties of a particular key. In addition to key's use and overall type (e.g., asymmetric RSA key for signing with a key length of 2048 bits), the attributes also define the restrictions on duplication and key usage [cf. Art15, Key Types and Attributes, p. 125]. For example, the main duplication attributes are:

`fixedTPM`: If this attribute is set to `TRUE`, a key cannot be duplicated (at all), because it is always restricted to this particular TPM. Even a duplication within the same TPM is not possible.

`fixedParent`: A key with this attribute set to `TRUE` is restricted to a particular parent key, i.e., this key cannot be duplicated to a different parent key. However, migration is indirectly possible if the parent key is duplicable. In this case, both keys form a *duplication group*.

Finally, the TPM 2.0 specifies two pre-defined variations of the attributes, which result in two particular key types, which are similar to TPM 1.2 AIKs and Storage Keys:

Restricted Signing Keys: As the name suggests, this key is basically a signing key, which is however restricted to only sign TPM (attestation) structures. Those structures include PCR quotes, audit logs, certified key properties, and the TPM time [cf. Art15, p. 128]. In order to make sure that the structures have not been created externally, the TPM always adds and verifies the existence of a magic 4-byte value, `TPM_GENERATED`. If the input value for the signatures starts with this value, the TPM does not produce a signature. That way, a remote verifier can be sure that the TPM created the signed value internally.

Restricted Decryption Keys: This type of key is essentially a storage key, which only decrypts certain data structures that have a specified format. Those keys are mainly used to decrypt child objects, such as externally stored keys, or to activate a credential [cf. Art15, p. 129].

2.1.2.4 Platform Configuration Registers

As shown in the architecture (cf. Figures 2.1 and 2.2), a TPM provides secure volatile memory, which includes special-purpose memory referred to as Platform Configuration Registers (PCRs). PCRs are designed to record cryptographic measurements of the current configuration and software state of the platform. As such, PCRs are automatically set to zeros once the host platform and TPM are powered up and it is ensured that they cannot be cleared without resetting the whole system[1]. However, since TPMs have limited storage capacities and, hence, cannot provide an unlimited number of PCRs to store the integrity measurement of each software component separately, they usually only implement 24 PCRs (since TPM 1.2, before that 16).

To overcome this limitation and enable a TPM to record a virtually unlimited number of integrity measurements, the specification defines an update function, which *extends* the current value in a PCR with a new measurement using concatenation and a cryptographic hash function, i.e.,

```
PCR[i] = Hash( PCR[i] || integrity measurement ) .
```

That way, a PCR not only stores one integrity value, but records a complete chain of measurements. As a consequence of using a hash function, however, the individual measurements in a PCR can no longer be distinguished and need to be stored separately on the host system. Fortunately, since the extend function is implemented inside the TPM and the PCRs are part of the shielded locations, the most recent PCR values (and thereby also the complete chain of all measurements) are protected. That means if an attacker modifies the externally stored individual measurements, any remote verifier is able to check and detect the attack, because the TPM always signs the internally stored PCR values and never externally provided data that looks like PCR values.

Since the TPM calculates the PCR values using an internal hash function, the digests are always 160-bit SHA-1 hash values in case of a TPM 1.2, because SHA-1 is the only available hash algorithm. That means a TPM 1.2 implements only one set of PCRs with a fixed size and the measurement log with the individual integrity values can use a simple format. On the other hand, as SHA-1 is more and more deemed insecure, an upgrade to a new hash function usually requires a hardware change.

For the TPM 2.0, in contrast, it is possible to select one of the implemented hash algorithms. Since the TPM 2.0 supports the concept of algorithm agility, there is usually more the one hash algorithm available. As mentioned above, that means a TPM 2.0 has to provide flexible PCRs: The TPM can either implement a generic set of PCRs with the size of the maximum length of all implemented hash digests or provide separate PCR banks, which are activated depending on the selected hash algorithm. In both cases, the TPM 2.0 transparently provides a set of 24 PCRs, which are suitable for the selected hash algorithm.

1 Actually, the TPM 1.2 as well as the TPM 2.0 specifications reserve PCR 16 as a debug PCR, which can be used to test software and is the only PCR that can be reset without a power cycle of the host system.

2.1.2.5 TPM 2.0 Features

Since the TPM 2.0 significantly improves and extends the features of its predecessor, the TPM 1.2, this section gives a brief overview of the relevant TPM 2.0 features, which we will use in our work. These features include *Authorization Sessions*, which enable three types of authorization methods, *Enhanced Authorization*, which improves the authorization mechanisms and implements policies, *Duplication* of keys, which has already been introduced in Section 2.1.2.3, as well as *NV Indices*, which extend the TPM 1.2 non-volatile storage interface, e.g., by providing new NV types and integrating authorization policies for internal TPM memory.

Authorization Sessions

According to the TPM 2.0 Library Specification, a session is a collection of TPM state that changes after each use of that session [Tru16]. As such, *session* are a generic concept, which can be used for *authorizations*, *audits*, and *encryption*. Since we primarily use authorization sessions in our attestation protocol, we briefly describe this type of session (and refer to the specification for the other session types).

As the name suggests, authorization sessions are a specific type of session, which can be used to authorize actions related to TPM entities, such as cryptographic keys. Sessions are created using the command `TPM2_StartAuthSession`, which generates a new session handle inside the TPM that can be used to reference that session.

To authorize actions via sessions, the TPM provides three types of authorization sessions:

Password "Sessions": A password session is actually a one-time authorization (not actually a session), which uses a plaintext password to authorize an action and, hence, does not maintain a state. Since no session context is created by the TPM, this type of authorization does not require the use of `TPM2_StartAuthSession`.

HMAC Sessions: As a more secure method to use password-based authorization, this type of session calculates an HMAC, which is mainly based on the authentication value of the TPM entity. This value is referred to as `AuthValue`, set at the creation of the TPM object, and only one of the inputs to the HMAC calculation used for a more secure authorization. Other inputs are random numbers (`noncesTPM` and `nonceCaller`), which protect against replay attacks.

Policy Sessions: Built on top of HMAC sessions, policy sessions extend the methods of authorizing actions related to a specific TPM entity. The name policy session comes from the fact that this type of authorization enhances HMAC-based authorizations with cryptographic policies (see *Enhanced Authorization*). While HMAC sessions are mainly based on the `AuthValue`, policies can include, for example, passwords, TPM state information (e.g., PCR values), command sequences, or content in non-volatile memory. Those authorization elements can even be combined using `TPM2_PolicyOR` in order to create complex policies.

Enhanced Authorization

With Enhanced Authorization (EA), the TPM 2.0 improves and extends the previously very limited authorization mechanism and unifies the method for authorizing TPM operations and objects. While TPM 1.2 authorizations are mainly based on passwords (authorization secrets), EA provides a novel authorization mechanism based on TPM 2.0 policies, which can authorize the use of any TPM entity, such as cryptographic keys, NV memory, or TPM functions. As such, EA is a TPM capability that allows entity-creators (or administrators) to require specific tests or operations to be performed before an action can be authorized and completed (based on [Tru16, Part 1, Section 19.7.1]). The specific policy for an entity is stored in a value referred to as `authPolicy`, which is short for authorization policy.

A TPM 2.0 policy is represented as a single cryptographic hash value, which is constructed in a similar way PCR values are calculated: The initial policy hash created by `TPM2_StartAuthSession` is always all zeros. Because of algorithm agility, the size of the hash digest depends on the selected hash algorithm. To construct a particular policy, the initial policy hash is "extended" in a specified way, which describes the test conditions and operations that have to be performed before the new entity can be used in a specific way as indicated in the policy.

For example, a policy for a new key could require particular PCR values and a specific content in a certain NV location. To construct such as policy, one has to start a new policy session, which creates the initial policy hash (and a session handle), and extend the two policy conditions onto the initial policy hash as specified in the TPM 2.0 Library Specification resulting in a hash chain, which is the policy. By referencing the session handle, the key can be created and later used if the policy can be re-created, i.e., the current conditions match the specified policy conditions.

As mentioned in the previous section, EA not only extents the method of authorization from a password to a sequence of authorization steps, but also enables the use of Boolean operators `OR` via `TPM2_PolicyOR` to combine multiple alternative authorization paths to a policy tree. If a leaf of the policy tree can be reached, the TPM entity can be used as specified by that policy.

Key Duplication

Duplication is the process of migrating cryptographic keys from one TPM to a different TPM or within key hierarchies of the same TPM. As described in Section 2.1.2.3, a key can be migrated if the key has both attributes, `fixedTPM` and `fixedParent`, set to `FALSE`. In case the attribute `fixedParent` is set to `TRUE`, the key cannot be migrated directly, but moves with its parent if that key is duplicable. If `fixedTPM` is set to `TRUE`, a migration (even within the same TPM) is not possible at all. In addition, duplicable keys need to be associated with a policy, which must at least contain the command code for `TPM2_Duplicate`. Such a policy can be created using `TPM2_PolicyCommandCode`, which enters a special authorization role for duplication (`DUP`).

NV Indices

As described in Section 2.1.2.1, the TPM includes non-volatile memory, which is required to store, for example, authorization values, proofs and secrets, seeds, as well as state information (e.g., counter or clock values). The NV memory can also be used to make TPM entities, such as cryptographic keys, persistent, hey are readily available.

In addition to those data structures defined by the TPM 2.0 Library Specification, the platform or a user can also store unspecified data in dynamic NV locations, so-called *NV Indices*. These NV indices are created using `TPM2_NV_DefineSpace` and have a certain size, (user-defined) handle, authorization value, and one of the following (data) types, which are defined in the specification:

Ordinary: Ordinary NV indices are similar to TPM 1.2 NV spaces. They can be used to store arbitrary data using `TPM2_NV_Write` and only limit the size of data that can be written to the index.

Bit field: NV bit fields also contain 64 bits that are initialized to all zeros. As defined by the specification, bits in a NV bit field can be set using `TPM2_NV_SetBits`, but not cleared.

Counter: NV counters store 64-bit values, which are initialized to the maximum value that any counter ever had on the same TPM. According to the specification, NV counters can only be incremented (using `TPM2_NV_Increment`).

Extend: NV extend indices behave like PCRs and are associated with a particular hash function, which determines the size and, in some respect, the content of the NV space. Initially, the index is set to zero and is extended with new values using `TPM2_NV_Extend`.

In contrast to the TPM 1.2, which also provides user-defined NV memory locations, NV indices can only be read after they have been initialized with a write operation. Otherwise, the TPM prevents reading the content of the index and returns an error message, which ultimately results in failed read operation. That way, it is no possible to inadvertently use an uninitialized NV index.

Furthermore, access to user-defined NV spaces can not only be protected by a *secret*, which is used in an HMAC-based authorization, *localities* (a level indicating the current mode of operation), *PCR values*, or *physical presence*. TPM 2.0 NV indices can alternatively be associated with a policy, which must be satisfied to be able to read or write data. While the corresponding policy hash is initially part of the public information, which is used to create the NV index with `TPM2_NV_DefineSpace`, the policy hash is later stored as `authPolicy` in protected memory.

In addition, TPM entities can reference an NV index in their policy using `TPM2_PolicyNV`, which requires a certain value in the NV memory location to satisfy the policy of that TPM entity. Depending on the type of NV index, the specification defines several operations, such as a comparison of equality or a bit check, which can be specified in the policy. We use policies associated with NV policies in our remote attestation protocol described in Chapter 9.

2.1.3 Trusted Computing Concepts and Protocols

This section gives a brief overview of relevant TC concepts, in particular, *authenticated boot* and *remote attestation*. While authenticated boot is a method for collecting integrity measurements, attestation is a way to report those integrity measurements to a remote party, which can, in turn, cryptographically verify those measurements to evaluate the platform's trustworthiness.

2.1.3.1 Authenticated Boot (and the Integrity Measurement Architecture)

If a platform supports TC and is equipped with a TPM, it can measure each boot component starting from the CRTM and store the integrity measurement in the TPM's platform configuration registers before executing the next component in the boot chain. This process is called *authenticated boot* (or sometimes *measured boot*). In contrast to secure boot, which verifies cryptographic signatures for each boot component before executing the component, authenticated boot merely measures software, such as the BIOS, and its configuration to establish a so-called platform configuration. Since the measurement process starts with the CRTM, which is one half of the RTM, the chain of measurements is rooted in a trusted component. By measuring the next component in the boot chain before executing it, the TPM (the second half of the RTM) continues that chain. As a result, authenticated boot creates a transitive chain of trust, which is rooted in the RTM and contains the integrity measurements of all boot components including their configuration as well as the OS.

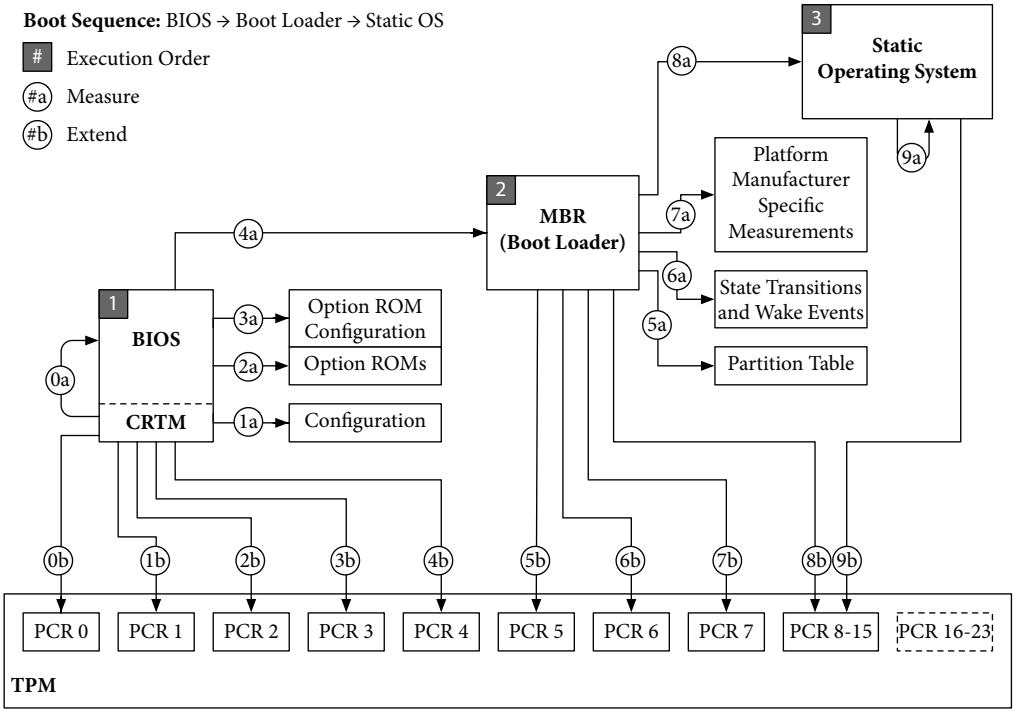

Figure 2.3: TCG Authenticated Boot for BIOS (PCR assignment based on [Art15, p. 152] and [Bul13])

As depicted in Figures 2.3 and 2.4, which show the authenticated boot process for a BIOS- and a UEFI-based system, the procedure starts with the CRTM measuring the BIOS/UEFI (step 0a). However, since the CRTM is usually part of the BIOS/UEFI firmware, this measurement includes the CRTM as well. The BIOS/UEFI, in turn, measures their configurations, additional components, such as Option ROMs or Extensible Firmware Interface (EFI) Drivers, as well as the Master Boot Record (MBR) or UEFI OS loader (steps 1a-4a). The results are extended into PCRs 0-4 (steps 0b-4b) and control is handed over to the MBR (boot loader) or the UEFI OS loader.

In steps 5-8, the boot loader in the BIOS-based system measures the partition table, state transition and wake events, the platform-specific components, and the static operating system (steps 5a-9a). Similarly, the UEFI OS loader measures the EFI variables and Globally Unique Identifier (GUID) partition table, state transition and wake events, secure boot keys and variables, as well as the static operating system. The resulting integrity measurements are extended into PCRs 5-9 as indicated in steps 5b-8b. For the final step 9, the boot loader hands over control to the kernel of the static operating system, which might measure OS specific components, such as the boot sector and boot block of the root file system. Those integrity measurements are usually extended into PCRs 10-15.

As a result of the authenticated boot process, the PCR values cryptographically represent the state of the platform. Since each boot component is measured before its is executed, an attacker cannot simply modify a boot component to compromise the measurement process. Also, such an attack can be easily detected by comparing the contents of the PCRs with trusted reference values.

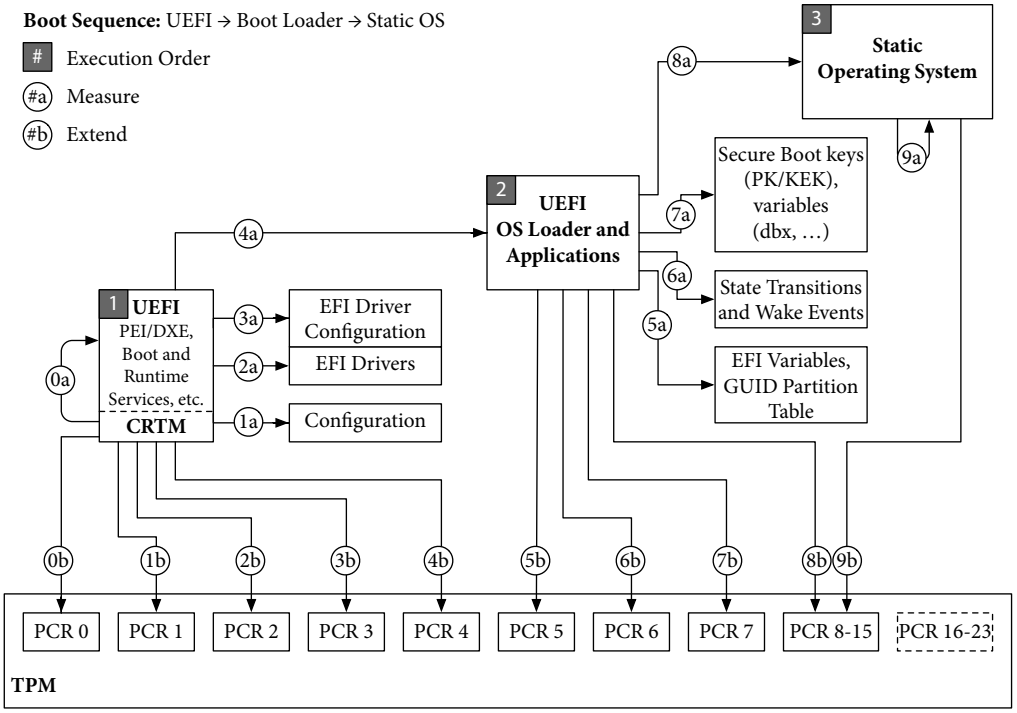

Figure 2.4: TCG Authenticated Boot for UEFI (PCR assignment based on [Art15, p. 152] and [Bul13])

Unfortunately, since authenticated boot only measures the boot components, the cryptographic measurements of the platform merely represent the state of the system when the boot process has completed. After some time, the integrity values stored in the TPM no longer describe the actual platform state, since malicious binaries might have been executed and have been able to compromise the system in the meantime. As a consequence, the integrity measurements inevitably suffer a loss of trustworthiness, when the system solely relies on authenticated boot.

To overcome this limitation, the Integrity Measurement Architecture [Sai04] has been proposed for Linux-based systems, where integrity values are calculated during run-time whenever a new binary is loaded. Particularly, the IMA subsystem is responsible for calculating the hashes of files, i.e., application binaries, regular files, libraries, and others, before they are loaded (and executed). As shown in Figure 2.5 (step 10), the resulting integrity measurements are stored in PCR 10, which is primarily used for IMA and initially contains a measurement value that is an aggregate of all preceding boot measurements linking the authenticated boot measurements to IMA. In addition to collecting run-time measurements, IMA also maintains a measurement list, which contains the individual integrity values and enables a (remote) verification of the platform state and its trustworthiness. As a result, IMA continues the chain of measurements started with the CRTM and established by an authenticated boot process. Most significantly, it provides a more recent state of the platform, because it contains measurements for all relevant files of a Linux-based system, particularly binaries, configuration files, and scripts.

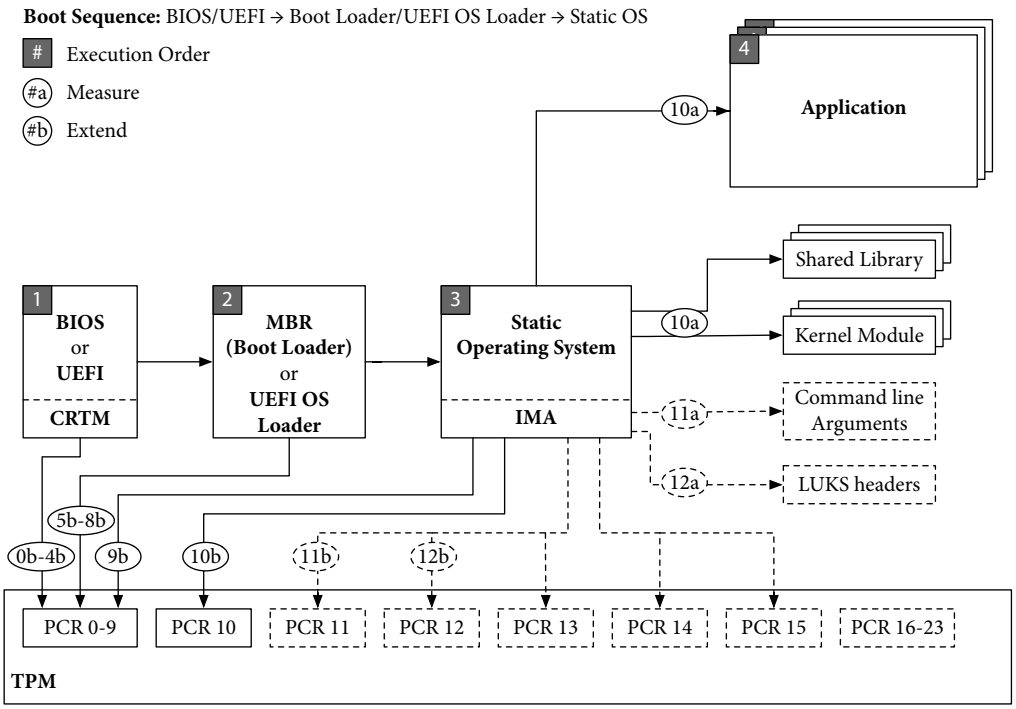

Figure 2.5: Integrity Measurement Architecture for Linux-based Systems

To be able to securely measure the relevant files, IMA is proposed as a so-called Linux Integrity Module (LIM) on top of Linux Security Modules (LSMs), which instrument certain kernel functions equipped with security hooks. Those hooks enable the extension of existing kernel functions by calling additional (privileged) code before or after security-critical parts of the original kernel function. More precisely, IMA mainly utilizes the security hooks `security_bprm_check` (which invokes `ima_bprm_check(bprm)`) and `security_mmap_file` (which invokes `ima_file_mmap`) to measure the contents of a file, whenever the Linux kernel maps that file to memory.

For example, when the Linux program loader maps a statically linked binary or a binary with no library dependencies into memory, that file is measured and the integrity value extended into the TPM before it is executed using the system call `execve`. Similarly, IMA measures binaries with shared libraries, which are loaded by the dynamic linker/loader `ld.so` or `ld-linux.so.{1,2}`, respectively. In this case, the loader uses `mmap` to map those libraries into memory which, in turn, invokes the security hook `security_mmap_file` and executes `ima_file_mmap`, which triggers the measurement process.

If the Linux kernel does not provide a security hook in a mapping or loader function, IMA requires a modification to the existing code. For example, in case of Linux Kernel Modules (LKMs), the system call `sys_init_module` is used to inform the kernel about a new kernel module, which has been loaded into user-space memory and needs to be copied into kernel memory and relocated. However, since there was no suitable security hook available in kernel 2.6, the authors of IMA added a measure call into the `load_module` routine that was called by the `init_module` system call, when the LKM resided in kernel memory and before it was relocated [cf. Sai04, Section 5.1 (Inserting Measurement Point), Kernel Modules]. That way, binaries like kernel modules can be measured, too, even if there is not pre-defined security hook.

As a result, the IMA concept and implementation for Linux enforces the measurement of integrity values for (relevant) files like binaries, including dynamically linked programs, before executing those binaries. In addition, regular files like configurations or scripts can be hashed and extended into the TPM, which provides a more recent cryptographic representation of the platform state. As such, IMA is a useful extension for authenticated boot and a strong basis for further verification mechanisms, such as *IMA-appraisal*, which enforces local validation of measurements, or the *Extended Verification Module (EVM)*, which can detect offline modifications and prevent execution.

Unfortunately, IMA is only available for Linux-based system and, without the additional security extensions like *appraisal mode*, does not prevent the execution of (potentially malicious) binaries. That is why we will adopt the IMA concepts to our microkernel-based system architecture and implement a secure loader, which enforces authenticity and integrity verification of binaries before executing them. Furthermore, our system will provide a mechanism, usually referred to as remote attestation, which allows a remote party to verify the platform's trustworthiness. The concept of a remote attestation as specified by the TCG is described in the following section.

2.1.3.2 Remote Attestation

Since authenticated boot and IMA cannot fully prevent the execution of potentially malicious binaries, an attacker might be able to compromise a system component, such as the boot loader or an OS component, during boot or run-time. That means the malicious component is measured, but nonetheless executed, which (unknowingly) puts the system in a untrustworthy state. In fact, this compromised state is difficult to detect, because the system has no way to reliably verify its own integrity measurements without a separated execution environment that can be trusted even after some components have been compromised.

As a consequence, the TCG has specified a protocol referred to as *remote attestation*, which is designed to enable a remote verifier to evaluate the trustworthiness of a platform equipped with a TPM. The main idea of remote attestation is to create cryptographic evidence, which supports a platform's claim about its current state and its general behavior, hence its trustworthiness. This cryptographic evidence usually has the form of a digital signature, which is calculated by the TPM, and since the TPM cannot be easily tampered with by an attacker or the platform to report arbitrary integrity measurements, it can be trusted to only produce and sign legitimate integrity values. Hence, remote attestation and the TPM acting as a trusted component enable a remote verifier to reason about the trustworthiness of a TPM-equipped platform.

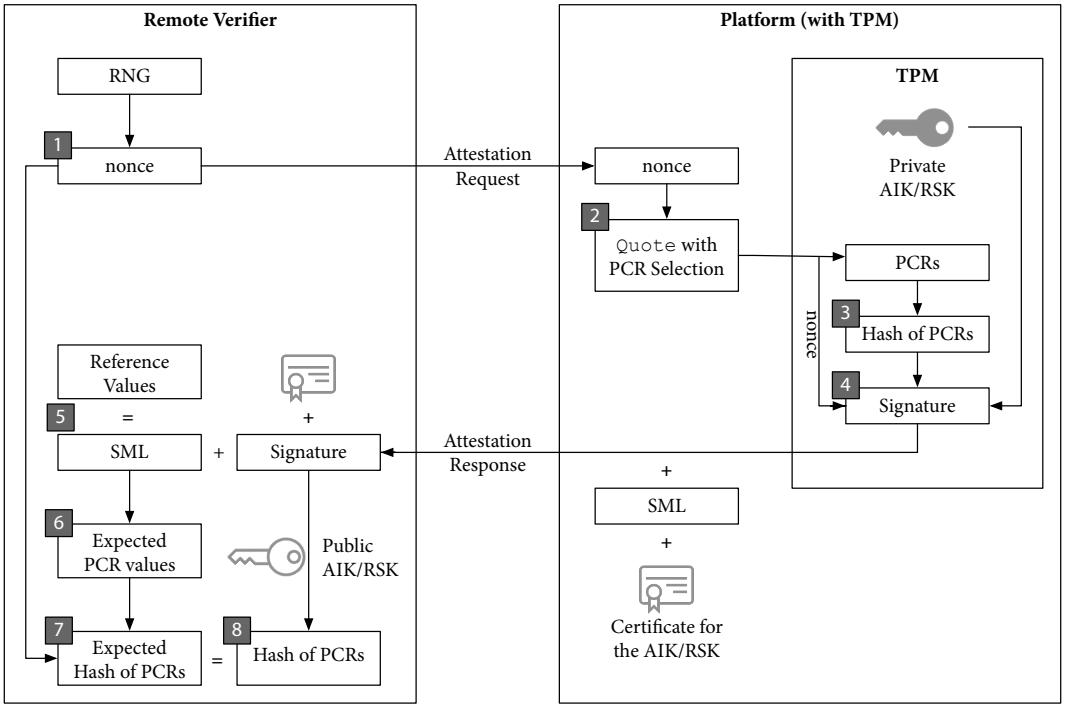

Figure 2.6: Remote Attestation as specified by the TCG (simplified)

More precisely, for a remote attestation as specified by the TCG, which is depicted in Figure 2.6, the TPM signs a selection of PCRs representing the current platform state and sends it to a remote party for verification. As shown on the left of Figure 2.6, the remote verifier usually starts by generating a nonce using its RNG in order to ensure freshness (step 1). The nonce is sent to the platform equipped with the TPM as part of an attestation request, which can also contain a selection of PCRs that must be included in the attestation.

When the platform receives the attestation request, the TSS issues an `TPM[2]_Quote` command including the nonce as shown in step 2 on the right of Figure 2.6. The TPM, in turn, calculates a hash based on the selected PCRs, which is referred as *composite hash* (step 3). This hash is part of an attestation data structure (*quote*), which is digitally signed by the TPM with a non-migratable signing key in step 4. If the platform is equipped with a TPM 1.2, this key is an AIK; if a TPM 2.0 is used, the key is a restricted signing key. The resulting signature is then sent to the remote verifier together with the so-called stored measurement log (SML) and the certificate for the signing key.

After the verifier has received the attestation response, the remote verifier compares the integrity measurements listed in the SML with a set of trusted or well-known references values (step 5). Based on the validated measurements of the SML, the verifier can then calculate the expected PCR values in step 6. The expected signature hash is then generated by hashing the nonce and the expected PCR values as specified by the TCG (step 7).

Furthermore, to check the digital signature of the TPM *quote*, the remote party validates the certificate and decrypts the signature using the public key from the certificate, which should not only result in the same hash as calculated by the TPM in step 3, but also as freshly created in step 7. If the hash values match, the remote party knows that the content of the quote is fresh, because it includes the random nonce (cf. step 1), and that the measurement log has not been tampered with. In addition, if the remote party can match all measurements to trusted and/or well-known references values, it is ensured or at least very likely that no unknown binaries has been executed before the TPM has created the quote[1]. Hence, the remote verifier can make an informed decision whether the platform is (still) trustworthy.

In this thesis, we call this protocol specified by the TCG *explicit remote attestation*, because the complete measurement log is transferred to the remote verifier and the entries are evaluated individually by the remote party. In contrast, we will present a remote attestation mechanism, which does not require digital signatures and the transmission of a log, but relies on symmetric cryptography. We refer to this type of lightweight attestation as *implicit remote attestation*. While explicit attestation is designed to PC and server platforms, our attestation is targeted at resource-constraint embedded system.

1 In practice, the so-called time of check to time of use (TOCTOU) problem describes the fact that there is usually a time gap between the calculation of the quote by the TPM and its verification by the remote party.

2.1.3.3 Trusted Network Connect

With remote attestation, the TCG specifies a protocol that enables a system to reason about the trustworthiness of a remote platform. However, the specification of a remote attestation does not define where or how the decision is made and what the consequences of a particular decision are. That is why the TCG *Trusted Network Communication* work group additionally specifies a complex abstract architecture which enables a Trusted Network Connect (TNC) that ensures endpoint compliance with integrity policies at and after network connect using remote attention.

As shown in Figure 2.7, the TNC architecture consists of five roles: the access requestor (AR), the policy enforcement point (PEP), the policy decision point (PDP), the metadata access point (MAP), and the MAP clients (MAPCs). The AR is an arbitrary system equipped with a TPM, which executes an Integrity Measurement Collector (IMC) that utilizes the TPM though a platform trust service (PTS), a TNC client (TNCC), and a network access requestor (NAR). By using the NAR component, the system requests access to the network, which is granted or denied by the PDP's network access authority (NAR) and enforced by the PEP. The PDP's decision is based on the results of the Integrity Measurement Verifier (IMV) in the integrity measurement layer, which is provided with integrity measurements by AR's TNCC sending the measurements to the TNC server (TNCS). Based on the decision, the MAP enables access to MAPCs, such as sensors or controllers.

As a result, TNC specifies abstract components and interfaces, which enable the implementation of a TCG remote attestation and the subsequent enforcement of network access policies, and ensures that a system requesting access behaves in a certain way. In this thesis, we adopt the idea of TNC and propose a lightweight approach for secure network access to mobile networks in Chapter 6.

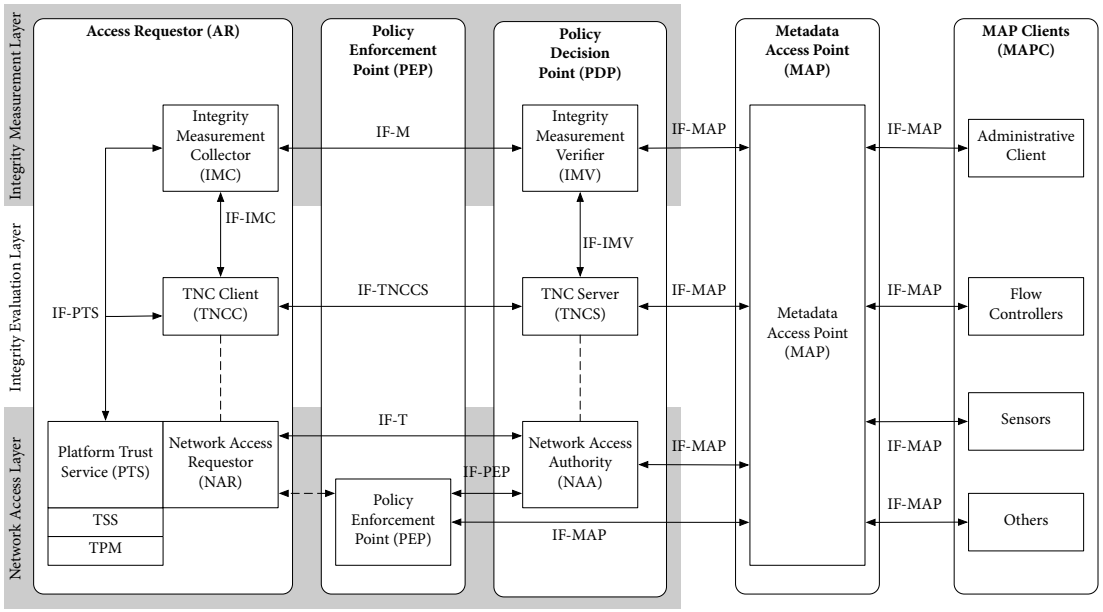

Figure 2.7: TNC Architecture (based on [Tru12, Figure 2])

2.2 Concepts and Technologies for Secure System Architectures

In addition to TC, this section presents relevant concepts and technologies, which enable the design and implementation of system architectures with a strong focus on security. The main concepts, which we utilize for our secure system architecture, are *isolation* and the *principle of least privilege*.

Isolation requires that the system prevents components from interacting with each other, unless those components are explicitly allowed to communicate. As a consequence, isolation can be archived through a combination of *separation*, a *reduced number of interfaces*, and a privileged system core (ideally with *small TCB*), which can control communication. Possible variants include:

Hardware Separation: Dedicated software components are implemented for separate hardware resources. For instance, a system like a mobile phone with special-purpose software, such as a baseband stack, might run this component on a dedicated (baseband) processor. Alternatively, other systems might use a separate Field Programmable Gate Array (FPGA) or a secure co-processor to execute security-critical software.

Software-driven Separation: In this case, software is divided into components and assigned to compartments, which use specific parts of the same hardware. For instance, a microkernel-based system, such as *Fiasco.OC* with *L4Re*, can create user-space compartments in software, which are (to some degree) independent of the underlying hardware.

Virtualization: As a combination of hardware and software separation, virtualization can execute software (even a complete OS) within a VM. To achieve isolation, the hypervisor dynamically divides/assigns hardware resources and focuses on keeping the boundaries of VMs secure.

Trusted Execution Environment: TEE-based separation divides software into security-critical and non-critical parts and statically assigns critical components to secure hardware resources while executing non-critical components in a non-secure environment on the same hardware.

The second main concept, the *principle of least privilege*, states that the system must only grant those rights that are necessary to fulfill a specific task. To achieve this objective, we target and develop a secure system architecture based on a microkernel, which implements all non-essential system components in user-space tasks and strictly separates those tasks. Since the microkernel has a very small TCB and, in general, controls all communication channels, e.g., using capabilities, it can not only isolate the microkernel tasks, but also assign fine-grained access rights.

In the rest of the section, we will discuss separation mechanisms and privileges in more detail. Specifically, we focus on hardware separation (Section 2.2.1) as well as software-based separation though microkernels (Section 2.2.2), which serves as an example for the principle of least privilege. Furthermore, we give a brief overview of virtualization and trusted execution environments in Sections 2.2.3 and 2.2.4. As part of the overview, we describe the virtualization and security technology on ARM systems, which we use to implement our secure embedded system architecture.

2.2.1 Hardware Separation

Similar to hardware redundancy, which can ensure robustness and availability, especially in safety-critical systems, dedicated hardware resources can provide a high level of security through isolation. By providing distinct hardware components, such as dedicated (co-)processors, and using those hardware resources to separate critical or sensitive components from the rest of the system, security can be significantly increased.

Practically, there are a number of possible ways to make use of hardware separation. For example, a system can consist of multiple processors or integrate a programmable hardware component, such as an FPGA, which can be used to implement security functions or components in hardware. Other systems include a dedicated programmable security co-processor, such as an HSM, or a special-purpose security module like a SIM. An example of a non-programmable security co-processor is a TPM. In the following section, we give an overview of relevant approaches on hardware separation.

2.2.1.1 Multi-Processor Systems

Computer systems, in general, harness and utilize more than one processor for two main reasons: to increase overall computational performance or to enable certain functionalities, which could not be realized otherwise. While the first reason applies, for example, to large computing centers, which often use symmetric multi-processor systems for performance reasons, the second aspect is more important for resource-constraint devices. For example, embedded systems and mobile devices, such as smartphones, often rely on (asymmetric) multi-processor architectures, which might provide a dedicated processor for their baseband stacks or energy-efficient co-processors for data acquisition and processing[1].

However, system with multiple separate processors not only benefit from an increase in system performance and a reduced energy consumption in case of asymmetric multi-processor systems. By isolating security-critical or real-time components, such as the baseband stack, using dedicated hardware resources, the ramifications of a successful attack on the main system can also be limited. For example, if the OS of a mobile phone runs on the application processor and is compromised by an attacker, software on the baseband processor is not necessarily affected.

As multi-processor architectures are common in embedded systems, especially mobile devices, we develop a remote attestation mechanism, which is suitable for this specific type of architecture. In particular, our attestation mechanism takes into consideration that multiple processors alone cannot realize a secure system architecture, which is why those system usually have an additional hardware-based security module, such as a SIM and/or TPM. Consequently, we describe this approach—the use of an (independent) hardware security module—in the following section.

1 For instance, Apple mobile devices, such as iPhone or iPad, use an Apple M7/M8 coprocessor based on an ARM Cortex-M3 for low-energy collection and processing of data, e.g., produced by motion sensors.

2.2.1.2 Dedicated Security Co-Processors

In contrast to separate general-purpose processors, which can be used to isolate arbitrary software components with different criticality, but are usually unable to protect against physical tampering, a dedicated security co-processor primarily implements and isolates security-related functions. Consequently, those secure hardware components provide elaborate protection mechanisms, such as shielding, which protect against sophisticated, in many cases even invasive physical attacks. For instance, most HSMs provide tamper-resistant key generation and secure hardware-based cryptographic accelerators. That way, an HSM can assist the host system in performing, for instance, computationally intensive cryptographic operations in real-time while protecting keys and secrets against extraction and misuse.

As security co-processors, which can protect against a range of physical and remote attacks, HSMs can be either realized as peripheral devices, such as extension modules or security chip cards, or integrated into a microcontroller. In both cases, those HSMs generally run a secure firmware, which enables the execution of HSM applications. For instance, SIM cards used in mobile networks are external security chip cards, which usually provides a Java Card platform and execute a smart card operating system, such as Java Card OpenPlatform (JCOP), that allows the operator as well as the user to run applications referred to as applets. Network operator, in particular, distribute SIM cards as trust anchors for mobile phones in order to implement a secure authentication process based in keys stored on the SIM. In this case, the SIM card acts as a hardware-based secure key store and trusted hardware component for a device, which otherwise must be considered untrusted.

In case of a TPM as a non-programmable security co-processor, the application executed by the smart card firmware is the TPM application logic and command interface, which is defined in the TPM specifications. As such, a TPM is basically a security chip card with a specified software interface, which is attached to a platform and implements the TPM specification. Since the TPM does not provide the ability to execute arbitrary code, remote parties can establish trust in a platform equipped with a TPM, because it becomes much harder to compromise the TPM. As a consequence, a TPM can act as a trust anchor, which isolates specific security functions and provides those functions as a service to the host system.

By separating security-critical functions from the rest of the system and isolating those functions in a dedicated security co-processors with a specified interface, the system architecture can significantly increase security. As a result, dedicated hardware-based security modules are used in mission-critical systems that have high security requirements as well as in regular devices, such as mobile phones, which require a hardware trust anchor, e.g., for authentication purposes. Hence, our system architecture will take advantage of a hardware-based security module, more precisely a TPM or HSM, respectively, in order to secure cryptographic keys and enable a secure remote attestation protocol.

2.2.2 Software-driven Separation

In this section, we briefly discuss relevant hardware-enforced, but software-driven separation concepts and architecture designs, particularly *process isolation* and *system compartmentalization*. Based on the discussions of these concepts, we introduce *microkernel-based systems* and their characteristics, such as the small TCB and the implementation of the principle of least privilege, which includes a high-level comparison with operating systems relying on monolithic kernels, such as Linux.

2.2.2.1 Process Isolation

The concept of process isolation requires the system to separate each process and protect an individual process from indiscriminate and unrestricted (write) access by other processes. Although the idea has been introduced decades ago, the concept of process isolation is so fundamental that it still applies today and might be more relevant than ever.

One way to realize process isolation is the use of *memory segmentation*, which divides the system's main memory and/or programs into segments or sections. Typically, different segments are created for different software components or for different types of program memory, e.g., code and data segments. A second concept for process isolation is based on *per-process virtual address spaces*, where one process address space is different from another virtual address space, hence preventing one process from writing onto other processes. Virtual memory is usually combined with paging, a memory management scheme, where the system stores and retrieves pages of data to and from secondary storage for use in main memory. This technique uses (per-process) page tables, which define the mapping used for virtual to physical address translation. To effectively implement this type of process isolation, most modern operating systems take advantage of special hardware components, such as a Memory Management Unit (MMU). An MMU is responsible for protecting memory segments and for translating a location, i.e., a segment and offset, into a physical memory address. Similarly, the MMU translates virtual memory addresses into physical addresses, handles page mappings, and checks that the mapping is valid and permitted.

In addition, modern operating systems often provide kernel and system features for advanced process isolation, which are directly based on the privileged role of the kernel, which controls the resources of the entire system. For example, the Linux kernel supports mechanisms for enforcing security policies, e.g., via Security-Enhanced Linux (SELinux), which implements mandatory access control. A complementary approach is the concept of *namespaces*, which separate resources and assign different names to objects, such as file system and network resources, process IDs, inter-process communication (IPC), or user and group IDs. Based on these software-based isolation techniques, the Linux kernel is able to support *(software) containers*, which allow for executing Linux-based user-space instances (cf. *LXC* [LXC08], *Docker* [Doc13]).

2.2.2.2 System Compartmentalization

In comparison to process isolation mechanisms, the concept of system compartmentalization not only isolates unprivileged user-space processes, but also proposes to de-privilege kernel components, such as memory management, file system implementations, and drivers, as depicted in Figure 2.8. Ultimately, the main goal is to increase security by reducing the complexity of the privileged code, which intuitively results in a decreased probability of a fatal flaw compromising the whole system. One way to achieve this objective is to move all non-essential system components into user space and, thus, reduce the amount of privileged code in the kernel core component.

To implement such a compartmentalized system, however, it is not only crucial that the system components in the various compartments stay isolated, the kernel also needs to implement a fast communication mechanism. The reason is that system components typically run in kernel mode and, hence, can communicate directly and without delays. With system compartmentalization, those de-privileged components run in user mode and are required to utilize the kernel for privileged operations and may need the kernel's support to establish a communication channel with other compartments. Hence, without efficient IPC, the system performance significantly degrades.

Assuming that the system provides a strict separation of system components, which are implemented as unprivileged components in user-space compartments, and efficient IPC communication, a compartmentalized system can be a strong basis for a secure system architecture. For that reason, we not only explore system architecture with dedicated hardware, but also research microkernel-based, compartmentalized systems and propose remote attestation protocols for this particular type of system design.

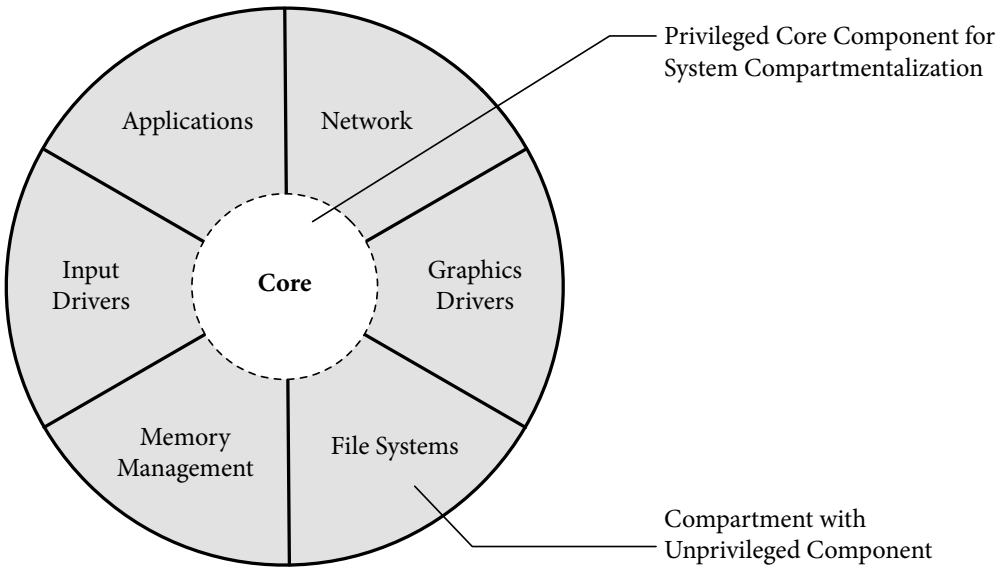

Figure 2.8: System Compartmentalization (based on [Fes06])

2.2.2.3 Microkernel-based Systems

As described in the previous section, the concept of compartmentalization proposes to separate and de-privilege kernel components, execute these components in user-space compartments, and strictly isolate those software compartments using mechanisms provided by a small, robust core. Consequently, modern system designs have adopted this concept for a secure system architecture and, furthermore, aim to minimize the complexity and size of the core, resulting in a *microkernel*. A microkernel is a *near-minimum* amount of privileged code that is as small as practically possible and still provides the necessary mechanisms to implement a complete OS. As shown on the right of Figure 2.9, these mechanisms include address space and thread management, scheduling, and IPC.

In contrast to a monolithic kernel, which is shown on the left of Figure 2.9, a microkernel implements all non-essential system components as user-space tasks, whereas a monolithic system combines various services that often require a large amount of privileged code in a complex kernel. As a general rule,

> "*[a] concept is tolerated inside the microkernel only if moving it outside the kernel, i.e., permitting competing implementations, would prevent the implementation of the system's required functionality.*" [Lie95]

For example, a monolithic kernel, such as Linux, handles memory management in kernel space, while a microkernel, such as Fiasco.OC, implements memory management *servers* in user space. Similarly, device drivers and services like networking are provided by servers running in user space. As a result, a microkernel-based system architecture can significantly decrease the probability that a fatal flaw in a component, such as a device driver, can compromise the whole system. As long

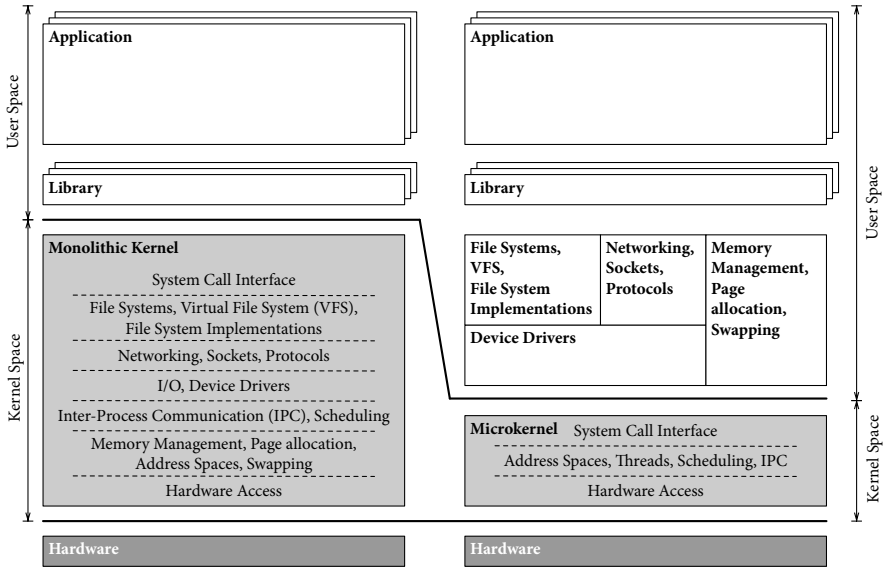

Figure 2.9: Monolithic System Design vs. Microkernel-based System Architecture (based on [Wei17])

as the system compartment is isolated, a device driver in user space can be restored and restarted without rebooting the complete system. Thus, a compromised component in one compartment usually cannot affect a component in a different compartment. More important, an attack on a user-space component, in general, has no impact on the core and, hence, no fatal ramifications.

From a security perspective, compartmentalized microkernel-based systems also naturally apply and enforce the principle of least privilege. Since system components, such as memory servers, which are usually referred to as *pagers*, only require the right to assign and manage specific memory, the system only grants those privileges. In a monolithic kernel, all kernel components can access all kernel data structures, hence there is only a small degree of separation and each component effectively has more privileges than required for its specific task.

Furthermore, modern microkernels like the 3rd generation microkernel Fiasco.OC and others, such as *seL4* or *Nova*, are characterized by an Application Programming Interface (API) with a focus on security, where access to resources is controlled by *capabilities*. In contrast to monolithic systems, which usually implement hierarchical protection domains (also referred to as protection rings), capabilities are system-wide unique object identities or tokens, which can be transferred, but not easily forged. Similar to a key, a capability is required to access a specific object or resource, which is locked or linked to that capability. Consequently, applications or services in capability-based systems directly share capabilities with each other according to the principle of least privilege.

Applications of Microkernel-based Systems

Based on their resilience against attacks and flaws, their reduced complexity and code size, as well as their strict organizational structure and implementation of the principle of least privilege, microkernels are perfectly suitable for safety- and security-critical applications. That means microkernel-based systems are not only suited for embedded systems, but also provide a robust way to implement, for example, a resilient hypervisor and virtualize other operating systems. Since the TCB of a microkernel-based hypervisor would be very small as the virtual machine monitor is executed in user space, hence requires only a small amount of privileged code for the host-VM-transition, we explore this particular application and design our attestation protocols to make use of this system architecture.

However, before we describe the concept and main ideas of virtualization in more detail, we will provide a short overview of the two microkernel-based systems, which we will use in our work: Fiasco.OC (with L4Re) and *Genode*. Both microkernel-based systems are open source and, hence, perfectly suited to research new applications for remote attestation mechanisms, which are designed for embedded systems with virtualization capabilities, where the microkernel acts as a very small hypervisor.

Fiasco.OC

The microkernel Fiasco.OC is a very small and robust 3rd generation real-time kernel supporting preemption and hard priorities, which implements a unified universal mechanism for naming, authorization, and communication control referred to as object-capability (OC) model [TUD11b]. As a multi-tasking, multi-address-space kernel, Fiasco.OC can create, isolate, and execute multiple microkernel tasks simultaneously. In addition, Fiasco.OC supports para- and hardware-assisted virtualization, which enables the system to run other operating systems, e.g., based on Linux.

Primarily, however, Fiasco.OC is designed to run the L4 Runtime Environment (L4Re), which is a user-space infrastructure that includes basic services, such as program loading and memory management [TUD11a]. As shown in Figure 2.10, a minimal L4Re-based system consists of three components: the Fiasco.OC microkernel, the *root pager (Sigma0)*, as well as the *root task (Moe)*. The root pager *Sigma0* is started as first user-space component, initially owns all system resources, and is usually only required to resolve page faults for the *Moe* root task. *Moe*, in turn, provides the essential services to normal user applications, such as an initial program loader, a region-map service for virtual memory management, and a memory (data space) allocator [TUD11a]. Most L4Re-based system, in addition, include the *init task (Ned)* and the *input/output server (Io)*.

We use those components in our work to implement our integrity verification, remote attestation, and secure loading mechanism described in Chapter 7. More precisely, we extend the system with additional tasks and device drivers for a TPM to measure and report a integrity values stored in the TPM's PCRs. Furthermore, we adapt the root task *Moe* to securely verify remote binaries, initiate the measurement process, and load the verified remote binaries into a new address space, i.e., create and execute the binary as a microkernel task.

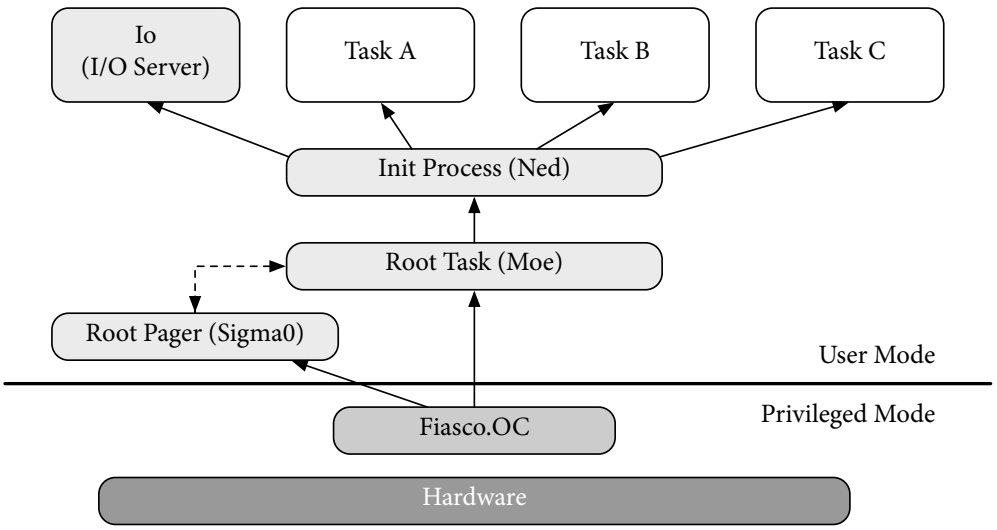

Figure 2.10: Fiasco.OC and L4Re Software Architecture

Genode

The *Genode OS* framework aims to provide a generic, microkernel-agnostic user-level infrastructure, which can be executed, for example, on top of an existing third-party microkernel, such as Fiasco.OC. As shown in Figure 2.11(a), Genode therefore provides user-space components similar to L4Re, which include a *core* (specific to the microkernel API), an *init* process, and a set of device drivers. As a result, Genode enables the development of microkernel-independent services and client applications by providing generic microkernel and OS interfaces.

In addition to the user-level components, Genode also provides a bare-metal implementation of its *core* component, which normally runs in user space on top of a third-party kernel like Fiasco.OC. As shown in Figure 2.11(b), this bare-metal core is executed directly on the hardware and eliminates the need for a third-party kernel, which further reduces the complexity and size of the TCB. Similar to the Fiasco.OC microkernel, the base-metal Genode core implements only required system functionalities and utilizes capability-based authorization mechanisms, e.g., to protect access to kernel objects and control communication channels.

Furthermore, like Fiasco.OC and L4Re, the Genode OS supports para- and hardware-assisted virtualization and can be executed inside the ARM TrustZone in order to implement a trusted OS inside a hardware TEE. Since Genode can be used in both cases as a trusted software component or operating system, which contains a near-minimum amount of privileged code, and we explore this approach in our work, the concepts behind virtualization and a TEE are described in the following Sections 2.2.3 and 2.2.4.

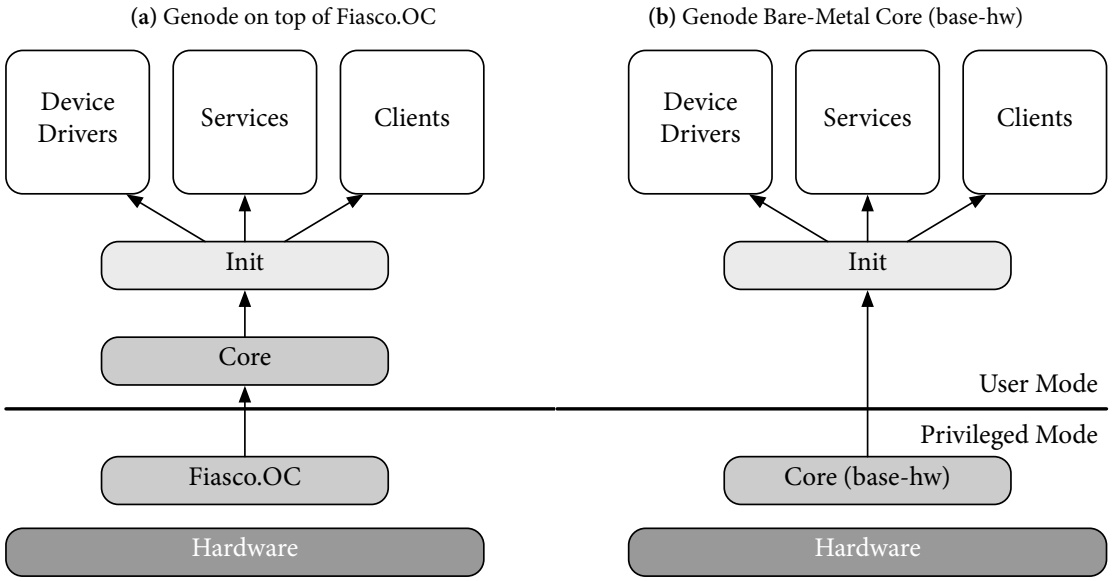

Figure 2.11: System Architectures based on Genode OS Framework

Advantages and Disadvantages of Microkernel-based Systems

Although microkernel-based systems might be considered more robust compared to monolithic systems, because they require less privileged code and implement all non-essential operating system services in user space, performance is a concern and can suffer if IPC is not implemented efficiently. Research [Lie93] has shown that "[IPC] has to be fast and effective, otherwise programmers will not use remote procedure calls (RPC), multithreading and multitasking adequately". Hence, IPC performance is crucial, especially for microkernel-based systems, because of the higher number of system calls as well as context switches compared to monolithic operating systems.

However, as a consequence of reducing privileged code und implementing OS services as user space tasks, microkernel-based systems are usually also considered more resilient against faults. The reason is that if one service is compromised or crashes, other services are generally not affected, since they are isolated by the microkernel and continue functioning as they use separate resources. Since the microkernel is very small and much less complex compared to monolithic kernels, such as Linux, the chance of successfully compromising a microkernel is, in general, significantly lower, which is why microkernel-based systems are often used in safety- and security-critical applications, such as airplanes.

By strictly following the principle of least privilege, i.e., implementing all non-essential tasks (even critical ones) in user space, isolating those tasks, and restrictively granting access rights, microkernel-based systems aim to reduce the risk of complete system compromise or failure. However, implementing critical tasks in unprivileged user space might also open up the possibility that those tasks are more vulnerable to attack. Hence, it is crucial that the microkernel strictly separates and protects tasks by correctly handling permissions, managing kernel objects, and controlling communication channels. Fortunately, compared to most complex monolithic kernels, the implementation of these vital tasks and functions of a microkernel can be verified more easily, because the code base of microkernels is inherently smaller by design and significantly less complex.

Finally, one major drawback of microkernel-based systems is the fact that most implementations are research projects and highly experimental for the most part [cf. Gen17; TUD11b]. There only exist a small number of commercial products, such as *SYSGO*'s *PikeOS* [SYS91] or *SiMKo 3* [Tel13] developed by *Deutsche Telekom (T-Labs)*, *Trust2Core*, TU Berlin, TU Dresden, and *Kernkonzept*. Although some modern operating systems, such as Apple's *iOS* [App07], make use of a few features adapted from microkernels (e.g., the message passing capability or memory protection mechanisms), *iOS* employs a hybrid kernel (*XNU* [App96]) and, therefore, does not have the same design goals, such as minimality [Lie95]. One of the main reasons not to use a microkernel-based OS usually is that performance was being a concern, especially in commercial products, such as mobile devices. However, as *PikeOS* or *SiMKo 3* demonstrate, microkernel-based systems can be the right solution if the requirements, such as high fault tolerance and robustness, mandate a very resilient OS, e.g., in government-issued communication devices, avionic systems, or critical infrastructure components.

2.2.3 Virtualization

In the 1960s, mainframe systems first started to use a basic form of virtualization as a method for locally dividing system resources, such as CPU time or memory. One main objective was to effectively enable different applications to run simultaneously without interference by separating resources and isolating applications. In the following decades, research and development broadened the concept of virtualization, which today includes, but is not limited to memory/storage, network, software, and hardware virtualization techniques.

For instance, most modern computer systems provide software and hardware mechanisms to implement *virtual memory*, which creates the impression of a (large) contiguous memory region, although the underlying physical memory is usually fragmented (and much smaller). As a result, each application has its individual, exclusive virtual address space, while the system is able to manage and restrict access to physical memory addresses, which enables, for example, process isolation and compartmentalization.

Another variation is *OS-level (software) virtualization*, which enables a kernel of a host operating system to execute multiple virtualized user-space environments often referred to as containers. While running on a shared kernel, individual (software) containers are isolated by kernel-based separation mechanisms, which can be rooted in physical hardware components, such as an MMU. As a consequence of running directly on a shared kernel, OS-level virtualization usually imposes little to no overhead, but cannot easily execute a guest operating system different from the host OS, because is is not designed to virtualize a different guest kernel.

In comparison to software virtualization, the concept of hardware virtualization usually refers to the emulation of a physical computer system by a *hypervisor* or *virtual machine monitor (VMM)*, which creates, manages, and isolates *system virtual machines*[1]. Depending on the type of emulation, hardware virtualization concepts can be divided into *paravirtualization* and *full virtualization*. While paravirtualization simulates a virtual hardware environment and provides an API to the guest OS, hence requires modifications to the guest, full virtualization emulates actual hardware with the help of a hypervisor/VMM such that a guest OS can be executed without modifications. However, since the emulation of actual hardware results in a significant performance impact and a complex hypervisor, modern CPUs usually implement *hardware-assisted virtualization* capabilities, such as Intel Virtualization Technology (VT) [Nei06] or ARM's Virtualization Extensions [ARM12; ARM10], which enable efficient full virtualization with a relatively simple hypervisor.

In the next sections, we discuss *paravirtualization* and *full virtualization* in more detail, as we explore and propose attestation mechanisms for virtualized systems. We also provide an overview of *hardware-assisted virtualization*, since it can significantly reduce the complexity of the *hypervisor*.

1 Compared to system virtual machines, which provide a virtual representation for a real machine, *process virtual machines* like the Java Virtual Machine (JVM) are designed to run programs in a platform-independent environment.

2.2.3.1 Paravirtualization

The term *paravirtualization* refers to a concept and technique, which simulates a virtual hardware platform and, thus, enables the execution of virtualized guest operating systems. However, since paravirtualization does not fully emulate actual hardware, but instead simulates virtual hardware with an interface that is different from the underlying physical hardware, guests need to be modified. The reason for the modification and the interface, which is usually referred to as the para-API, is to increase performance and reduce the complexity of the VMM by relocating the execution of certain operations, which are difficult to virtualize, from the virtual to the host/physical domain.

One example of a paravirtualized system is *L4Linux* [Här97], which is executed on Fiasco.OC. *L4Linux* is a modified Linux kernel, which has been ported to run as a service on L4 microkernels. As a result, the *L4Linux* kernel can be execute next to regular microkernel tasks and, simultaneously, acts as a root pager and task for ordinary Linux processes, which enables the execution of a rich Linux-based operating system on top of a microkernel. In our work, we take advantage of this virtualized system architecture in order realize our remote attestation and secure loading protocols. In our proposed system architecture, the paravirtualized Linux acts as a rich OS, which provides, for example, networking capabilities in a complex and untrusted environment.

2.2.3.2 Full Virtualization

In contrast to paravirtualization, the concept of *full virtualization* emulates actual hardware and provides a virtual representation of a physical computer system in form of a system virtual machine. To emulate actual hardware and isolate virtual machines, full virtualization often utilizes binary translation, a technique that allows to automatically detect and replace (or trap and emulate) unsafe or privileged instructions, e.g., I/O operations, which alter the state of other VMs or the hardware. Besides binary translation and I/O emulation, full virtualization usually also requires the shadowing of certain data structures used by the processor(s) in order to fully emulate the hardware. As a result, full virtualization can create a virtual machine environment, which is a complete emulation of the underlying physical hardware. In addition, since the guest is not required to use a host interface like a para-API, modifications of the guest OS are not necessary.

Unfortunately, full virtualization often means a considerable overhead, which usually results in a significant decrease in performance over systems executed natively. In addition, since the virtual machine monitor needs to fully simulate the actual hardware, virtual machine monitors for full virtualization are usually relatively large, complex, and privileged software components. To overcome these issues and because some hardware platforms like IA-32 (Intel Architecture, 32-bit) have not always provided support for full virtualization, modern CPUs implement hardware virtualization extensions. Those extensions enable efficient hardware-assisted full virtualization and much simpler hypervisors. As a result, the TCB of virtualized systems can be decreased significantly.

2.2.3.3 Hardware-assisted Virtualization

Although system virtualization is possible without support by the underlying physical hardware, software-based virtualization generally results in a performance degradation compared to systems executed natively and requires a rather complex hypervisor. As a result, more and more systems with advanced CPUs implement hardware-based capabilities, which enable efficient full virtualization. In general, these virtualization capabilities enhance the hardware with an additional privilege level or CPU mode for the virtual machine monitor. As a consequence of the hardware extensions, privileged instructions can be trapped and handled more efficiently by a much simpler hypervisor, which requires significantly less privileged code and, hence, reduces the TCB.

Two of the most widely used hardware virtualization extensions are *Intel VT* [Nei06] and ARM's *Virtualization Extensions* [ARM12; ARM10], which add an additional privilege level or mode for the hypervisor to the CPU, implement system registers specifically designed for system virtualization, and provide support for memory, I/O, and device virtualization. Since our work concentrates on embedded systems, which are predominantly equipped with ARM-based system on chips (SoCs), we focus on ARM virtualization extensions, which are described in the following paragraph.

ARM Virtualization Extensions

As shown in Figure 2.12, the ARM Virtualization Extensions implement a CPU mode for the hypervisor in a privilege level 2, which is referred to as HYP mode. Furthermore, the hardware extensions provide additional system registers designed for virtualization purposes and extend hardware components, such as the MMU, timers, or the ARM Generic Interrupt Controller (GIC), with virtualization support. That way, the hypervisor can efficiently virtualize multiple guest operating systems, which are executed in their own (protection) domains as depicted in Figure 2.12. In particular, since the VMM is located in a higher privilege level (PL2), sensitive instructions that change the state of another VM or the hardware can be trapped and handled transparently.

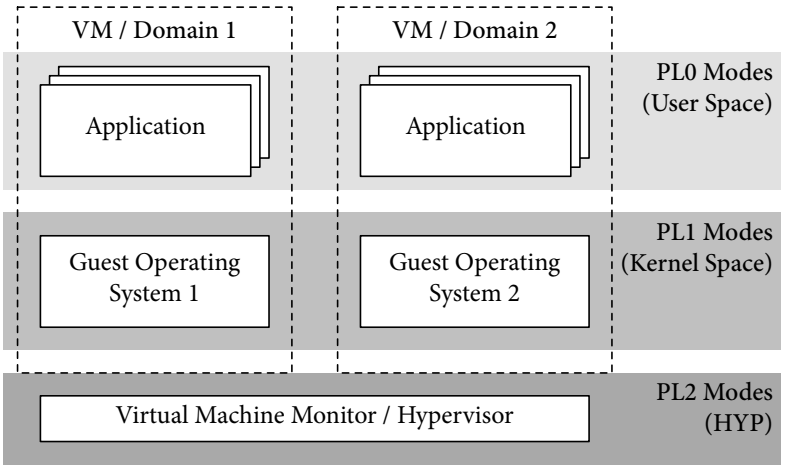

Figure 2.12: ARM Virtualization Extensions: HYP Mode and Protection Domains

To enable transparent hardware-assisted full virtualization, the ARM Virtualization Extensions provide virtualization-specific system registers, which are accessible in HYP mode only, e.g., the Virtualization Translation Control Register (VTCR), the Hyp Configuration Register (HCR), or Virtualization Translation Table Base Register (VTTBR), which, for example, stores an 8-bit field called `VMID`, The `VMID` allows the hypervisor to track the current VM in a similar way the OS tracks processes and their memory pages using an Address Space Identifier (ASID) from the Translation Table Base Register (TTBR) as tags for the entries in the translation lookaside buffer (TLB).

The main purpose of the VTTBR, VTCR, and HCR, however, is the setup and configuration of the second-stage address translation (SLAT), also referred to as *nested paging* or *Stage-2 page-tables*, which enables a guest OS in a virtual machine to set up its own pages tables in a Stage-2 MMU. As shown in Figure 2.13, the guest OS can subsequently translate guest virtual to guest physical addresses (stage 1) using the Stage-1 MMU and its own page tables without impacting the VMM and its translation from guest physical to host physical addresses (stage 2). The hypervisor, in turn, can not only setup translations from guest/immediate to host physical addresses, but also trap memory access by a guest OS, which enables, for example, emulation of memory-mapped devices.

As a result, the hardware assists in translating addresses over multiple stages and enable the VMM to effectively create distinct protection domains for virtual machines, which are isolated by hardware-based separation mechanisms, and configure memory traps. In addition, the complexity of the hypervisor can be reduced significantly, because the management of guest virtual and physical address translations is mostly handled by the MMUs. As a result, hardware-assisted virtualization enables the design of a near-minimum hypervisor, which is the basis for our secure system architecture and discussed in the following paragraph.

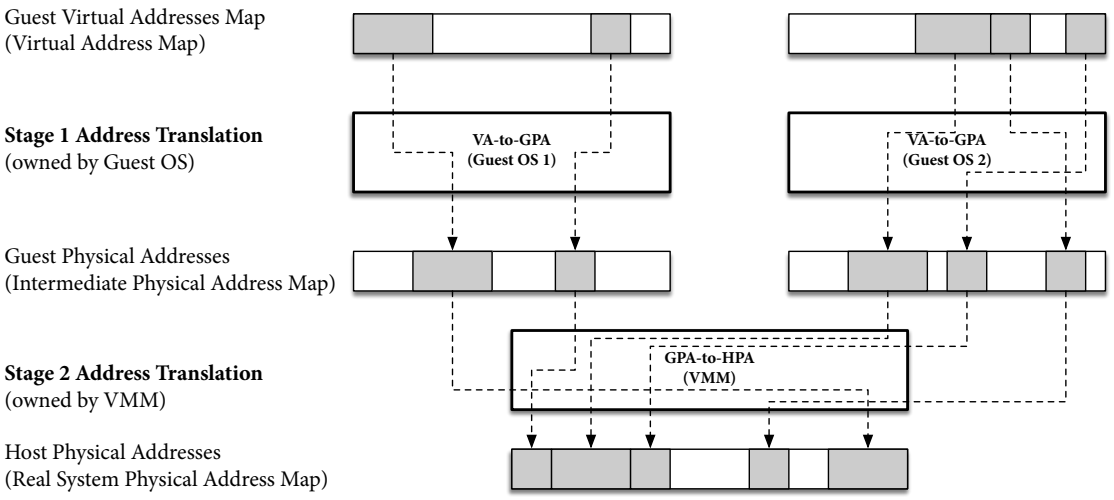

Figure 2.13: ARM Virtualization Extensions: 2-Stage Memory Address Translation

Near-Minimum Hypervisor

Hypervisors are usually classified as *type-1* or *type-2 hypervisor* based on their implementation and location in the system [Pop74]: type-1 hypervisors, which are often referred to as bare-metal or native hypervisors, run directly on physical hardware (see Figure 2.14(a)), while type-2 hypervisors require a host OS as shown in Figure 2.14(b) and, hence, are also referred to as hosted hypervisors. In practice, however, the distinction between type-1 and type-2 hypervisors is not necessary clear. For example, Kernel-based Virtual Machine (KVM) for Linux and FreeBSD's *bhyve* effectively convert the host OS into a type-1 hypervisor, while practically competing for resources with other applications running on the OS and, hence, behaving like a type-2 hypervisor. Nevertheless, the categories help to analyze the design of hypervisors and assess their characteristics and suitability.

Evidently, for the implementation of a simple, near-minimum hypervisor with a small TCB, type-1 hypervisors are inherently more suitable, since their design and implementation require no host OS. Additionally, type-1 hypervisors are more likely to support embedded systems, because those system usually have limited resources and type-1 hypervisors can run directly on hardware. As a result, the use of a microkernel acting as a very simple core for a compartmentalized system in combination with hardware-assisted virtualization further enables the design of a hypervisor with a near-minimum amount of privileged code as shown in Figure 2.14(c). Hence, we use this design for our hypervisor-based architecture, which we will introduce in Chapter 5 and explore in Chapter 9.

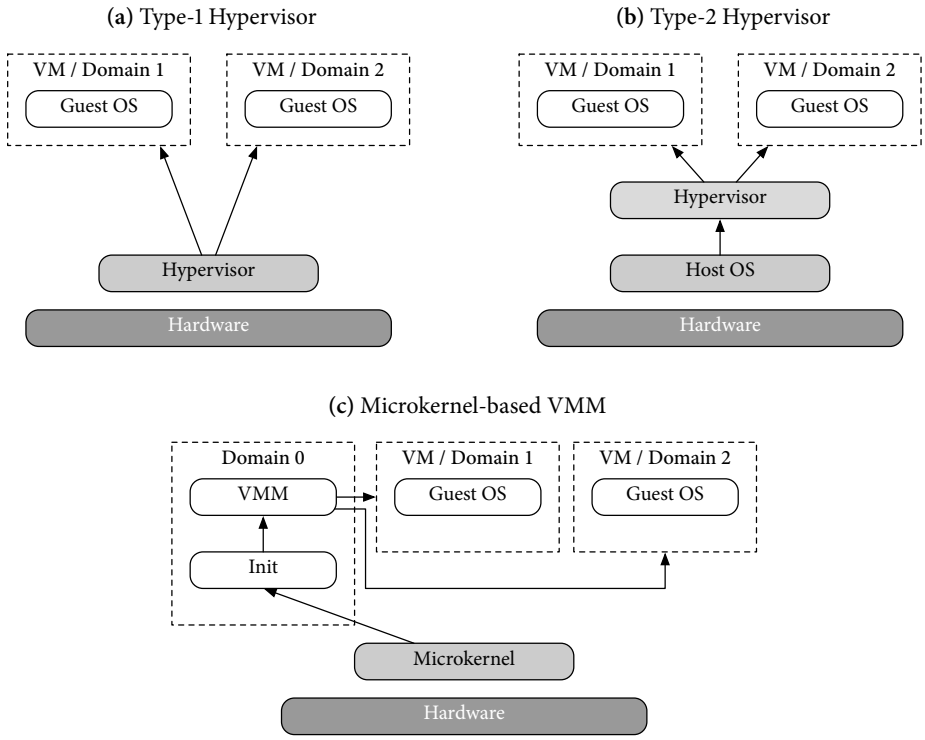

Figure 2.14: Type-1 and Type-2 Hypervisors versus Microkernel-based VMM

2.2.4 Trusted Execution Environment

While virtualization enables the execution of virtual machines with rich operating systems and platforms may even use hardware virtualization capabilities to efficiently run and isolate those VMs, a Trusted Execution Environment (TEE) usually focuses on securing critical software and data, such as trusted applications and their cryptographic keys. Hence, in addition to isolated execution, a TEE may offer secure storage, remote attestation, secure provisioning, and a trusted path [Vas12].

One way to realize a (very limited) TEE is based on a TPM, which not only isolates the execution of cryptographic operations, but also helps to implement secure storage despite its own internal storage constraints. Furthermore, a TPM provides mechanisms for a remote attestation as well as secure provisioning, which includes, for example, the secure generation of cryptographic keys. However, since TPMs are passive security modules, which rely on an RTM provided by the CPU, TPMs alone are unable to create a trusted path without the help of the platform and, hence, cannot provide a full TEE [Bai11; Glo11], which supports, for example, custom trusted applications.

Fortunately, other hardware-based technologies, such as ARM's *Security Extensions* (also known as *TrustZone*) or Intel's *Software Guard Extensions (SGX)*, enable the implementation of a fully isolated TEE, which can accommodate and protect a trusted OS, applications, and sensitive data. Intel SGX, for example, allows user-level code to create private memory regions known as *Enclaves*, which are protected from other code (and vice versa) by hardware-based separation mechanisms. Thus, Intel SGX provides the basis for a CPU-based TEE, which allows for isolated execution of code and helps to create secure storage. Since the technology is tightly integrated with the CPU, Intel SGX also enables secure provisioning of cryptographic keys (stored in CPU-internal memory), the measurement of integrity values, remote attestations, and the creation of a trusted path.

However, since our work focuses on embedded system, which predominately use ARM-based systems, where Intel SGX is not available, we concentrate on the ARM Security Extensions instead. As shown in Figure 2.15, ARM TrustZone provides a privileged CPU mode known as *Monitor Mode* and separates the system into two execution environments—the *Secure* and the *Non-secure World*.

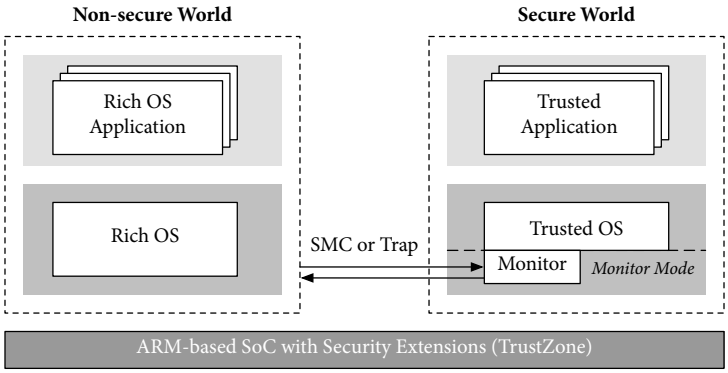

Figure 2.15: ARM Security Extensions: Secure World and Non-secure World

A switch between the Secure World and the Non-Secure World is only possible via Monitor Mode, which can be entered by deliberately issuing a Secure Monitor Call (SMC) or by triggering a trap. As a result, an ARM-based system with TrustZone support can host a rich operating system in the Non-secure World and rely on a trusted OS in the Secure World.

As depicted in Figure 2.16, the ARM Security Extensions are orthogonal to the Virtualization Extensions, which provide hardware-assisted virtualization, e.g., by extending the Non-secure World with HYP mode in PL2 and enable the efficient virtualization of more than one rich OS. ARM Security Extensions provide the Monitor Mode, which is considered more privileged than any privilege level in the Non-secure World. Hence, ARM TrustZone effectively extends the system with an additional secure execution mode. This secure execution mode is not limited to the CPU, but also includes the CPU busses as well as all TrustZone-aware peripherals. In Monitor Mode, the current execution mode can be configured via Secure Configuration Register (SCR), which includes the so-called non-secure (NS) bit. By setting this bit to zero, the CPU, caches, main memory, and peripherals switch to secure execution mode, which means the Non-secure World is no longer able to access secure resources, such as RAM regions or devices, which have been tagged as secure.

Since we focus on ARM-based embedded systems, we use TrustZone in combination with a TPM, which is accessible via Secure World only, to create a TPM-equipped TEE based on a microkernel. Our final system architecture presented in Chapter 9 also uses a near-minimum hypervisor, which is also based on a microkernel and executed in a separate privilege level in the Non-secure World. As a result, our system architecture enables a sophisticated remote attestation mechanism.

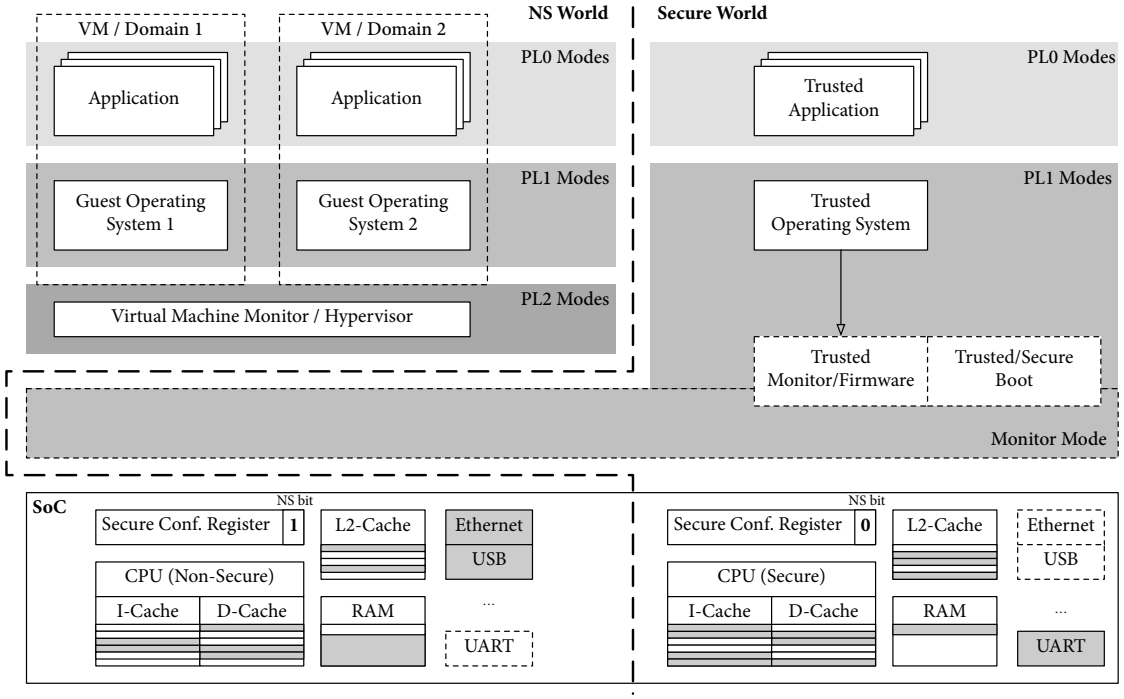

Figure 2.16: ARM Security Extensions: TrustZone-based System Architecture

Background

3

Related Work

This chapter presents related work on *integrity measurement and verification concepts* as well as *remote attestation*, in particular, hardware-based attestation mechanisms, mobile device attestation, and attestation protocols for virtualized systems. Specifically, we discuss existing remote attestation schemes based on TC concepts and contrast their characteristics to our attestation mechanisms, which are described in detail in the following chapters, more precisely Chapters 6 to 9.

As a result, this chapter provides a broad overview of related work regarding integrity verification with a focus on hardware-based remote attestation mechanisms. Furthermore, by including related work on mobile device attestation, this chapter also motives the need for a comprehensive attestation, which comprises all critical software components in a system, such as the mobile baseband stack. In addition, the discussion of related work on integrity verification and remote attestation concepts for virtualized systems aims to contrast these approaches with our own integrity measurement concept, which supports microkernel-based systems, and our attestation mechanism, which uses efficient symmetric cryptography instead of expensive asymmetric operations.

The remainder of this chapter is structured as follows. In Section 3.1, we present related work on integrity measurement concepts based on authenticated boot or UEFI secure boot, such as IMA with its various extensions, which add, for example, local integrity verification capabilities. In Section 3.2, we discuss existing software- and hardware-based attestation schemes, particularly their core mechanisms to create verifiable cryptographic proof for the trustworthiness of a platform. Furthermore, we present additional related work on mobile devices attestation in Section 3.2.1 and discuss attestation mechanisms specifically designed for virtualized systems in Section 3.2.2. Finally, we discuss selected dissertations with related topics in an addendum.

3.1 TC-based Integrity Measurement and Verification Concepts

In the context of Trusted Computing, *authenticated boot* as specified by the TCG (cf. Section 2.1.3.1) was one of the first TPM-based mechanisms for establishing a cryptographic platform configuration, i.e., a set of cryptographic integrity measurements for all components involved in the boot process. Unfortunately, TPM-based authenticated boot does not include all components of the OS and is not intended to enable the measurement of drivers, applications, or files loaded during runtime. Similarly, *UEFI Secure Boot* [UEF12, Sect. 27] may be able to protect the boot process by verifying digital signatures of certain boot components, which prevents loading of unsigned and possibly malicious drivers or OS loaders. Nevertheless, secure boot is also not designed to measure and/or verify OS components or applications, which are loaded after the boot process has been completed.

To solve this problem, Sailer et al. [Sai04] proposed IMA for Linux, which measures every binary that is executed (cf. Section 2.1.3.1) and, by today, includes various extensions, such as IMA-appraisal and IMA's EVM, which enable local integrity verification of binary and configuration files. Jaeger et al. [Jae06], for example, extended IMA for Linux with a mechanism based on SELinux, which additionally verifies information flows at runtime. In an implementation for Samsung smartphones, which is referred to as TrustZone-based Integrity Measurement Architecture (TIMA) and part of Samsung KNOX [Sam13], the IMA components run in the ARM TrustZone, which enables the software in the Secure World to measure and verify the Linux kernel executed in the Non-secure World.

However, since IMA is usually realized as part of the monolithic Linux kernel, its implementation is not directly compatible with microkernel-based operating systems or similar embedded/real-time operating systems, e.g., for baseband processors, which usually do not use Linux. In those cases, integrity measurement values are collected during authenticated or secure boot if at all, which only enables a remote attestation of a static platform configuration (see Chapter 6). As a consequence, we present a concept and implementation of IMA for microkernel-based systems in Chapter 7, where the integrity measurement components are executed as unprivileged user-space servers, while still enabling the system to measure microkernel application binaries before they are loaded. As a result, the microkernel-based system can calculate a set of cryptographic integrity values including run-time measurements, which allows for a remote attestation of a dynamic platform configuration.

Based on the platform configuration, which is composed of integrity measurements, a remote attestation is the cryptographic process for creating verifiable proof that enables a remote verifier to detect modifications to the prover's system, thus allowing the attestor to determine the prover's trustworthiness. As a consequence, it is not surprising that most remote attestation are based on some type of integrity measurements, while their attestation mechanisms and integrity verification process can differ significantly as the following section shows.

Alternative and Complementary Concepts for Integrity Verification

In addition to TC-inspired integrity measurement and verification concepts, there are a number of alternate approaches, such a *virtual machine introspection (VMI)*, *control flow integrity (CFI)*, or various methods for *anomaly detection (AD)* based on machine learning. Those concepts can compliment the idea of cryptographic, hash-based integrity verification, especially during runtime. For microkernel-based systems, those integrity verification techniques likely need to be adapted to address the characteristics of such a particular system architecture, which should be feasible.

Although these alternative concepts for integrity verification, i.e., VMI, CFI, or AD, are not a focus of this thesis, we present a brief overview of the basic ideas and research conducted in these fields as far as relevant. We highlight differences as well as opportunities to combine those concepts with our approach, which might benefit both techniques as they complement each other. In particular, we detail the ideas behind anomaly detection, which is often based on sophisticated machine learning techniques. The reason is that we integrate such a mechanism in Chapter 8 to show how our approach can include AD results in an implicit remote attestation. At the same time, the remote verifier can trust in the AD component if the integrity of the system is verified through remote attestation and the system deemed trustworthy with respect to the load-time integrity measurements.

Virtual Machine Introspection (VMI)

The main goal of VMI is to understand and assess a VM by "externally monitoring the runtime state of a system-level virtual machine" [Pay11], e.g., from the virtual machine monitor or another VM. Although VMI was originally introduced by Garfinkel et al. [Gar03] as "a way to protect a security application from attack by malicious software" [Pay11], the mechanism can also be used in a broader context for intrusion detection, forensic analysis, malware analysis, or kernel debugging [Sch11]. To achieve its objective, VMI components can monitor various aspects of a virtual machine, such as processor registers, memory, network traffic, or hardware-level events. For effective monitoring, however, the so-called *semantic gap* between the view of the VMI tool, e.g., located in the hypervisor, and the observed VM, needs to be bridged, which is not a trivial task to accomplish [Pfo10; Sch11].

As mentioned above, VMI can complement the TC-inspired, hash-based cryptographic integrity verification of a system-level VM. While TC-based collection of integrity measurements combined with a TPM-based remote attestation can report the load-time integrity and trustworthiness of a system, VMI tools can monitor VMs during runtime. Those VMI tools can be implemented either in a dedicated VM, in the hypervisor, or as a user-space task in a microkernel-based system. For example, *Xen* [Len15; Xen17] as a widely used hypervisor, which uses a microkernel-inspired design, supports VMI on Intel and ARM platforms and implements VMI functionality in a dedicated VM. Similarly, in most regular microkernel-based systems, the VMI tools would be implemented as a user-space task within a dedicated protection domain.

Control Flow Integrity (CFI)

Another technique for the verification of software components is referred to as control flow integrity that has been the focus of security researchers for several years [Aba05; Vog14; Wan10; Xia12a; Zha13]. The main goal of CFI is to design effective mechanisms that prevent attackers from redirecting the flow of execution of otherwise benign programs. For that purpose, CFI [Aba05] implements checks to ensure that the control flow of a program remains within its control flow graph. The control flow graph is determined using static analysis, while the protection is usually realized by instrumenting the call sites with runtime checks, which enforce that the function being called is actually in the set of functions identified by the static analysis. The runtime checks can also be implemented as part of the hypervisor [Hor15], which verifies and enforces correct runtime behavior, e.g., using transparent page-based execution tracing. Similarly, such CFI mechanisms could be integrated into the microkernel-based system, more precisely a dedicated security task responsible for checking the control flow of other tasks, either based on function calls or page jumps. This should be feasible in principle, but would certainly require the adaptation of those concepts to the characteristics of a microkernel-based system, such as the principle of minimality (cf. Section 2.2.2.3).

Anomaly Detection (AD)

While VMI monitors the runtime state of virtual machines and CFI aims to prevent attackers from redirecting the flow of execution during runtime, anomaly detection mainly focuses on a reliable classification of behavior and events relevant to system integrity using machine learning techniques. As such, anomaly detection concepts can be used to recognize faults in embedded systems [Max02], manipulation of components for critical infrastructures [Rac12], malware detection [Pfo13], and various other types of attacks, e.g., against the memory of a system or execution flow of software components [Cuz15; Vie17; Yoo17; Yoo15]. Anomaly detection and machine learning can also be utilized to model and monitor system behavior [Xia13a; Xia13b; Xia12b; Xia13e; Xia15]. However, since the system behavior usually depends on external inputs, this approach might require additional protections against attacks that try to compromise the model or the implementation of the machine learning algorithm, hence, proper isolation of the AD component.

Although the research of effective anomaly detection mechanisms is not the focus of this thesis, we take advantage of AD concepts in Chapter 8, where we integrate the result of an anomaly detection component in our implicit attestation protocol. This AD component, which runs as a microkernel task in our system design, monitors other components, i.e., microkernel tasks, classifies certain events, such as system calls, and records anomalies in dedicated registers of an HSM, which are similar to PCRs, but for anomalies. In our attestation protocol, we use those anomaly detection records in combination with the PCR values in order to not only prove cryptographic integrity, but also a classification for the current system behavior. More precisely, we assert that the probability of anomalous behavior is below a certain threshold.

3.2 Remote Attestation

In this section, we discuss existing remote attestation schemes, in particular their main idea for determining the current state of a platform as well as their core mechanism for creating verifiable cryptographic proof and reporting it as a statement about the system's trustworthiness to a verifier. We show that attestation schemes often greatly differ depending on their intension and application, but have the common goal to provide the means to securely determine the trustworthiness of a platform.

While the focus of the section is on hardware-based remote attestation, specifically TPM-based attestation protocols, we start by providing a brief overview of software-based attestation schemes. Although software-based attestation techniques may not provide the same security guarantees, e.g., regarding the protection of integrity measurements or cryptographic keys, those attestation schemes can still be useful in some scenarios, e.g., when the verifier component is actually attached to the platform. In those cases, the attestation is often implemented in form of a challenge-response protocol without the need for integrity measurements or cryptographic keys. Even though such an application of challenge-response protocols only technically constitute a remote attestation, software-based attestation schemes are nonetheless related to the work conducted in this thesis.

Software-based Remote Attestation

In the absence of a hardware security module, such as TPM, which acts as a hardware trust anchor, e.g., to securely store integrity measurements or protect cryptographic keys, software-based remote attestation schemes [Li10; Pre13; Ses04; Sha05; Sri10] usually rely on a challenge-response protocol. The general idea is that the verifier sends a challenge, e.g., a random number, which the prover has to incorporate in its measurement process and, thereby, in the response. The verifier, in turn, calculates the measurement result locally and compares it with the response generated by the prover.

As an example of such a software-based attestation technique, *SWATT* [Ses04] "verif[ies] the memory contents of embedded devices and establish[es] the absence of malicious changes to the memory contents" by hashing a specified memory region. Similarly, *SBAP* [Li10] extends this approach to peripherals and measures the software/firmware as part of a challenge-response protocol. Unfortunately, those protocols assume that the attacker is not able to modify the hardware, in particular the memory, and that the verifies knows the exact hardware specifications, which can be considered rather restrictive.

Based on the concept of a software-based trust anchor, *SobrTrA* [Hor14] proposes and implements a verification mechanism specifically for ARM Cortex application processors. This mechanism, although not designed for attestation purposes, can also be used to verify the trustworthiness of a platform. In comparison to a hardware-based trust anchor, such as a TPM, however, the mechanism requires hardware-specific implementations (e.g., for different Cortex-A-processor), which makes this approach difficult to use in practice.

Hardware-based Remote Attestation

In a *hash-based remote attestation* as specified by the TCG [Tru11; Tru16], the prover's system calculates static load-time integrity measurements for all relevant software components, which are securely stored inside the PCRs of the TPM and can be used to prove the system's integrity to a remote verifier. More precisely, each boot component hashes the next software component during authenticated boot starting from an immutable CRTM as described in Chapter 2, Section 2.1.3.1. After the boot process has been completed, the operating system continues to measure software binaries through integrity verification mechanisms such as IMA. For a hash-based remote attestation, the integrity measurements inside the PCRs are signed by the TPM and sent to the remote verifier as described in Section 2.1.3.2. With the corresponding public key and a so-called SML, the remote party is able to verify the signature and check the entries of the SML against expected measurements provided the prior signature verification was successful.

To address privacy concerns related to the attestation key, the TCG alternatively also adopted a remote attestation primitive called Direct Anonymous Attestation (DAA) [Bri04], which aims to preserve the prover's privacy using zero-knowledge proofs. Since the attestation key for a traditional remote attestation is a cryptographic pseudonym in form of a certificate that needs to be signed by a certificate authority (CA), this *Privacy CA* can easily correlate any attestation key with a certificate that it signed to a specific TPM and, hence, must be trusted. DAA tries to solve this issue by providing a (rather complex) cryptographic scheme based on a zero-knowledge protocol, which enables the prover to convince the verifier that the attestation key is a cryptographic key created by a valid TPM that belongs to the prover without disclosing the identity of that TPM.

However, since both primitives specified by the TCG, traditional remote attestation and DAA, focus on hash-based load-time integrity measurements for software binaries only, other schemes, such as *property-based* [Che06; Küh07; Sad04], *group-based* [Als10], or *logical attestation* [Sir11], have extended and generalized the attestation mechanism. For example, the idea behind property-based attestation is to prove certain security characteristics and qualities rather than to verify the hash-based integrity of certain software components. Similarly, logical attestation is based on attributable, verifiable statements about software properties, which are expressed in a logic. Group-based attestation, in turn, uses Chameleon signatures [Kra98] to enhance privacy and the ability to manage software integrity. As a result, group-based attestation, for example, solves the problem that the same software change over time and old versions need to be deprecated.

Most of these remote attestation protocols, however, mainly rely on quite expensive cryptographic operations—more precisely, digital signatures. Even with a dedicated cryptographic coprocessor, such operations are, in general, mathematically complex, rather inefficient compared to symmetric operations, and thus may not be suitable for a remote attestation of mobile devices and virtualized embedded systems with a microkernel-based software architecture.

3.2.1 Mobile Device Attestation

Existing integrity verification/attestation schemes [Kun06; Mut08; Yin10; Zha09] for mobile devices, such as smartphones, may be able to justify a phone system's integrity and provide cryptographic proof that the device is in a trustworthy state, but they usually only focus on the integrity of the operating system and applications executed on the application processor. As a result, the main idea of these remote attestation protocols is primarily based on traditional TC concepts, such as hash-based remote attestation, which digitally signs integrity measurements stored inside a TPM.

However, since these protocols only consider the software executed on the application processor, e.g., the operating system and applications, they usually ignore the baseband stack running on a dedicated baseband processor. Thus, they cannot be directly used to prove the trustworthiness of the baseband stack, e.g., towards the mobile network. As a consequence, they are not able to prevent, for example, attacks on the mobile network infrastructure in case an attacker has been able to compromise the baseband stack of a larger number of mobile devices. Unfortunately, research shows that such kind of attacks are a realistic scenario, since there already exist a number of demonstrated exploits for vulnerabilities in mobile baseband stacks [Mul11; Tra09].

That is why we present a remote attestation mechanism for mobile devices with a baseband processor, which creates cryptographic proof that the baseband stack is still trustworthy (see Chapter 6). In addition, since previously proposed attestation protocols mainly rely on expensive asymmetric cryptographic operations, such as signing, which are rather expensive compared to symmetric cryptography, we have designed the attestation mechanism to use symmetric operations.

Regarding the trustworthiness of operating systems executed on the mobile application processor, more recent attestation concepts [Ben11; Mar13; Nau10] focus on providing verifiable proof for the integrity of mobile operating systems, such as Android, because of its increasing popularity. Nauman et al. [Nau10], for instance, propose a remote attestation scheme specifically designed for Android systems. To collect measurements, they extended the ClassLoader of the Dalvik VM, Android's process virtual machine [Smi05, p. 38], which allows to execute Java binaries. For native applications that do not run in the Dalvik VM and the Dalvik binary itself, however, they still rely on IMA in order to collect measurements. In addition, those concepts are unable to prevent loading of unknown binaries, since they only measure, but do not verify the integrity of binaries before they are executed.

Furthermore, those existing remote attestation protocols ignore the baseband stack completely, although they are often specifically designed for mobile phones, which most likely have a dedicated baseband processor with a separate software stack. And since the baseband stack is often even more critical to the mobile device (and the mobile network), it should be imperative to also include it in a remote attestation of a mobile device. One way is presented in Chapter 6 which details our attestation mechanisms and how it fits into the standardized mobile network authentication.

3.2.2 Remote Attestation of Virtualized Systems

For virtualized systems, Berger et al. [Ber06] proposed the concept and implementation of a virtual TPM (vTPM), which provides the facilities to create a virtual version of the hardware TPMs for each VM. As a result, the concept of vTPMs enables a traditional remote attestation as specified by the TCG for individual VMs. Unfortunately, since TPMs, especially the TPM 1.2, only provide limited support for virtualization, some components of the vTPM, such as the virtualized PCRs, have to implemented in software. As a consequence, it is critical to isolate the virtualized TPMs from the rest of the system, e.g., using isolation mechanisms provided by the hypervisor or an additional secure co-processor. Alternatively, PCRs of a hardware TPM can theoretically be shared by adequately multiplexing the process of extending integrity values for different VMs [Vel13].

While the concept of vTPMs focuses on the virtualization of TPMs and relies on a hypervisor or secure co-processor to isolate the software-based TPMs, other virtualization-based proposals primarily utilize the hypervisor to separate the integrity measurement and remote attestation code from a rich operating system kernel. For example, Härtig et al. [Här05] proposed the so-called *Nizza Secure-System Architecture* for microkernel-based systems, which describes how applications can be isolated from a virtualized rich operating system, such as Android, which is also executed on top of the microkernel [Lan11]. However, although the *Nizza* secure-system architecture enables the isolation of critical applications, such as the measurement code, this theoretical concept mainly focuses on the separation mechanisms and the reduction of the TCB while only briefly mentioning a need for a TPM-based remote attestation.

Unfortunately, most existing proposals do not implement a secure system architecture like *Nizza* and are either based on a hosted VMM [Che08], Xen [Sch09], or KVM [Hof13; Sch12] for Intel architectures. Other concepts rely on a TEE utilizing hardware features of an x86 processor [McC08] or a custom hypervisor [McC10], which also requires a modern processor from AMD or Intel. Hence, none of the concepts deal with a TCB-reduced microkernel-based system architecture on an embedded ARM platform as described in Chapter 7, which is equipped with a hardware TPM for securely storing integrity measurements.

For instance, Joshua Schiffman et al. [Sch12] proposed an attestation mechanism for virtualized systems, which includes a local representative (proxy) of the backend system. It is designed to increase the performance and efficiency by implementing a proxy component on the prover's system, which verifies that system locally and, thus, reduces the time between the attestation and the verification of the result. Similar to our local attestation approach proposed in Chapter 7, this reduces the gap between time of measurement and the attestation (cf. TOCTOU problem). However, they do not focus on providing a near-minimal TCB for their proxy as they rely on KVM as their hypervisor utilizing IMA. In addition, this approach still relies on traditional attestation mechanisms based on digital signatures to verify the hypervisor, virtualized device drivers, and the proxy component.

In comparison, our remote attestation mechanism described in Chapter 8, focuses on a lightweight attestation, which relies on symmetric cryptographic operations rather than digital signatures and only requires very small messages to prove the integrity of the hypervisor, which is based on a microkernel, and multiple microkernel tasks. Additionally, our protocol enables secure code updates based on the integrity of certain critical tasks, such as the microkernel or the baseband stack. That allows for the installation of new security-critical tasks and applications, which depend on the trustworthiness of those existing tasks. The trustworthiness, in turn, depends on the cryptographic verification of those tasks, which is evaluated by the remote verifier and follows the definition by the TCG, which requires software to behave as expected.

In contrast to the attestation protocols mentioned above, which are based on characteristics, qualities, and logical properties that are enforced by the operating system, our attestation mechanism presented in Chapter 9 is based on hash-based cryptographic policies that are enforced by a TPM 2.0. This TPM can either be a dedicated hardware TPM or implemented as a software TPM within a TEE. In both cases, the TPM software stack is protected by the ARM TrustZone security mechanism, which separates the components within the *Secure Word* from the software in the *Non-secure World*, e.g., a Linux-based OS.

In addition, due to the use of ARM's hardware-assisted virtualization, our system architecture used in Chapter 9, which employs a second microkernel as near-minimal hypervisor (cf. Chapter 5), allows for the virtualization of multiple rich operating systems in the *Non-secure World*. As a result, the attestation protocol presented in Chapter 9 can, for example, be combined with complimentary systems designs for a rich OS, such as *trust|me* [Hub16]. This Linux-based system architecture, which was originally designed for mobile devices, focuses on OS-level virtualization, which "allows to simultaneously operate multiple user land OS instances on one physical device" [Hub16].

As a result of this comprehensive system architecture, our approach presented in Chapter 9 can protect and isolated various software components based on their trust levels. That enables a policy-based implicit attestation, which does not rely on expensive cryptographic operations, such as digital signatures, to create verifiable proof for the integrity of the prover's system (include the virtualized rich OS). Furthermore, our remote attestation mechanism, which is designed for, but not limited to microkernel-based virtualized systems, enables a verifier to protect the integrity of data, e.g., from a virtualized device, while implicitly verifying the trustworthiness of the prover's system.

Addendum: Delimitation to Related Dissertations

In this supplementary section, we briefly discuss selected dissertations [Vel17; Wei16] dealing with related topics, in particular microkernel-based security architectures, TPM-based remote attestation as well as hardware-based integrity verification and protection for virtualized systems. The reason for the delimitation is that the authors of those dissertations work at the same research institution as the authors of this thesis and part of the research is joint work [Wei14] or certain general topics, e.g., the usage of TPMs for integrity protection, have been researched in parallel.

In the dissertation by Weiss [Wei16], which includes results from a joint publication [Wei14], the author develops *system architectures to improve trust, integrity, and resilience of embedded systems*. Similar to our approach, the author proposes to make use of a microkernel to separate a software-based trusted execution environment from a rich operating system. The author's objective is the design of a secure system architecture that enables secure loading of remote binaries, which is a goal we share. In comparison, however, the author focuses on a purely microkernel-based architecture and explores the security of that software-based separation layer, while we will ultimately propose a system architecture with a hardware-based trusted execution environment with hardware-assisted virtualization instead of paravirtualization. We believe that this system architecture can provide a higher level of security and trust, because the hardware security features are considered to be much harder to attack. On the other hand, the security of the separation layer and the effectiveness of *side channel attacks* as described in [Wei16] continues to be of concern. That is why the results of the dissertation by Weiss are still relevant, even in our hardware-based security architecture.

The dissertation by Velten [Vel17], on the other hand, focuses in parts on the integration of a TPM 1.2 into a virtualized system, where each VM should get access to the TPM. However, since the TPM 1.2 does not support virtualization natively and the VMs are usually provided with software-based virtualized instances of the hardware TPM, one main objective is the possibility to store integrity measurements of multiple VMs in a single PCR of the hardware TPM. In order to achieve this objective, the author proposes a protocol to multiplex and conceal the integrity measurements of VMs when they are extended into the TPM. That way, a remote attestation is based on integrity measurements stored in the TPM and an attestation only reveals the measurements of a particular VM, because the other measurements are concealed. In comparison to the work in this thesis, the approach by Velten extends the concept of traditional remote attestation as specified by the TCG, while we propose a novel mechanism that focuses on symmetric cryptography and on hardware security modules, which better supports virtualization, such as the TPM 2.0. However, hardware-based virtualization support is likely to support only a limited number of VMs, whereas the concept described in [Vel17] allows a virtually unlimited number of VMs with the drawback that the size of measurement logs increases significantly.

4

Attestation Scenarios and Attacker Model

This chapter first defines four scenarios, which aim to provide context for the remote attestation use cases and motivate the work presented in the following chapters. Based on the descriptions of our attestation scenarios in Section 4.1, which include potential threats, we further define our attacker model in Section 4.2, which specifies capabilities and a few reasonable limitations of an adversary.

The four scenarios are *secure mobile network access*, which focuses on mobile devices for accessing mobile network infrastructures, *secure loading* of (remote) applications, *secure code updates* for microkernel-based systems, and a *secure data access*. Although these four attestation scenarios, which are detailed in the following sections, are quite diverse, they unite in a common objective, which is *secure access to critical or sensitive remote (network) resources, applications, and data*.

Furthermore, the scenarios have in common that they are often based on embedded systems, such as a mobile phone, industrial control system, automobile, or airplane, which execute safety- and security-critical components alongside a rich OS, are equipped with a number of peripheral devices like sensors, and have some type of (mobile) internet connectivity. Since a large number of such embedded devices are used in various areas, be it in a private, corporate, industrial, or public/government environment, integrity verification and remote attestation protocols can not only help to protect those devices, but also the networks they connect to against attacks.

As a consequence, our remote attestation mechanisms are designed to provide verifiable proof for the trustworthiness of those systems and protect the network against attacks by an attacker. Furthermore, since we take advantage of a microkernel-based system architecture with a TPM, we can protect keys and make sure the cryptographic proof is based on strong protection mechanisms.

4.1 Attestation Scenarios

As a basis for our attestation and integrity verification/protection protocols, we define four different *attestation scenarios*, which can be combined to a high-level scenario that focuses on *secure access* to critical/sensitive remote (network) resources, applications, and data. Those four scenarios are:

Scenario 1: Secure Mobile Network Access (Chapter 6)
This scenario focuses on the secure authentication and authorization for mobile devices with a baseband stack (e.g., cellular phones) and is described in detail in Section 4.1.1.

Scenario 2: Secure Loading (Chapter 7)
In this scenario, a remote party (e.g., a bank) provides its own application, which must be securely executed inside a protected microkernel-based compartment (see Section 4.1.2).

Scenario 3: Secure Update and Recovery (Chapter 8)
This scenario, which is detailed in Section 4.1.3, focuses on secure code updates and recovery for a microkernel-based OS with multiple operational and cryptographic contexts.

Scenario 4: Secure Data Access (Chapter 9)
In this final scenario, a remote party can access data (e.g., generated by a sensor) on the embedded system and needs to ensure that it can trust its integrity (see Section 4.1.4).

Since secure access to resources is based on the result of a remote attestation in all our scenarios, we define two recurring parties for each attestation scenario, the prover $\mathcal{P}$ and the verifier $\mathcal{V}$. While the prover $\mathcal{P}$ is usually our microkernel-based embedded system, which is derived from the following scenarios and described in Chapter 5, the nature of the verifier $\mathcal{V}$ generally depends on the specific scenario.

4.1.1 Scenario 1: Secure Mobile Network Access

In the first scenario, we focus on secure access to a mobile network with a mobile device featuring a multi-CPU architecture, i.e., with a separate baseband stack executed on a dedicated baseband processor, and an SE issued by the network operator. In this scenario, the prover $\mathcal{P}$ is the mobile device and the verifier $\mathcal{V}$ is a set of verification components within the mobile network.

To access the mobile network, the mobile device has to register and cryptographically sign in, which is mostly handled by the SE that protects the keys and implementations of the cryptographic algorithms. However, while the network operator can trust in the fact that the keys and algorithms are protected by the SE, the software of the baseband processor, the baseband stack, is usually unprotected and can be manipulated. That enables an attacker to not only compromise the mobile device, but also the mobile network. Hence, we propose to equip the mobile device with a TPM and show that an efficient implicit remote attestation mechanism can be integrated into the standardized authentication protocol of 3G and 4G networks, which enables the mobile network to detect compromised baseband stacks and quarantine those mobile devices.

4.1.2 Scenario 2: Secure Loading

Our second scenario describes the case that a remote party, such as a bank, wants to provide its own security-sensitive application, such as an offline banking, to its customers. Unfortunately, most monolithic system would be viewed as too insecure to realize such a scenario, while microkernel-based systems might provide adequate security though separation and strict isolation, but are usually statically configured. In addition, the bank must be sure that the system is not compromised and that the application is securely executed inside a protected compartment on their customers' devices. Hence, the trustworthiness of those devices must be verified in advance, e.g., using a TPM-based remote attestation, which means we have two parties: an embedded device, such as a smartphone, which acts as the prover $\mathcal{P}$, and a remote party, e.g., a bank, which is the verifier $\mathcal{V}$.

To access and use the application provided by the bank, the customer's system first has to provide verifiable cryptographic proof that the system can be trusted. Only if the customer's device can convince the bank that the system is not compromised and that the banking application is executed in an isolated compartment, the bank can be confident that offline transactions generated with the applications are legitimate and not compromised by an attacker.

As a result, we propose that the customer's device uses a microkernel-based system with a small TCB as well as a TPM, which can be used to provide secure storage for application-specific keys and integrity measurements. In addition, if the integrity measurement component is implemented as microkernel task and the banking application is executed in the microkernel execution environment, it is possible to realize secure loading of remote binaries and use cases like secure offline payments.

4.1.3 Scenario 3: Secure Update and Recovery

For our secure update and recovery scenario, the prover $\mathcal{P}$ is an embedded system with different security- or safety-critical applications and at least one virtualized rich OS, such as an airplane or modern car. This system uses a multi-context HSM, which supports virtualization and collects events from an anomaly detection component. $\mathcal{V}$, on the other hand, is a remote verifier, e.g., a manufacturer, which is considered honest, trustworthy, and well-known to the prover.

Without any loss of generality, we assume that the prover is a airplane with a microkernel-based system architecture, which can execute various tasks in isolated execution environments. Typical tasks are the baseband stack for communications with a mobile network, virtualized device drivers for hardware components, native tasks for security-critical applications, such as a secure communication, and regular user applications running on a rich operating system, e.g., a Linux-based Android. Since those tasks have different levels of criticality, they are strictly isolated by the separation mechanisms of the microkernel and the hardware. However, a remote attacker might still be able to compromise tasks, e.g., by fuzzing their interfaces. That is why the prover has to provide verifiable evidence for the integrity of relevant tasks before the verifier grants access to restricted resources, such as emails, confidential documents, or updates.

For example, in this scenario, the airplane might regularly connect to a airline network via virtual private network (VPN), whenever the user needs access to internal resources. To establish a secure connection, the verifier in the airline's network first requires proof for the integrity of security-critical tasks, such as certain device drivers and the VPN client. Based on the attestation result, access to the airline's network is granted or denied. In case access is denied, e.g., because the VPN client was compromised, the verifier should be able to provide a code update based on the integrity of only the most basic security-critical tasks, such as the microkernel and the baseband stack. That way the prover is able to recover.

4.1.4 Scenario 4: Secure Data Access

The secure data access scenario focuses on the protected access to data, e.g., generated by sensors. We assume that the prover's system creates the data records, hence acts as the data producer, while the verifier is interested in (and maybe willing to pay for) those data records or at least aggregated values, thus acts as data consumer. Furthermore, data access control should be realized by *policies*.

In this scenario, the prover $\mathcal{P}$ is an embedded data collection system, such as a monitoring or industrial control system with data access control. Hence, $\mathcal{P}$ is equipped with a TPM 2.0 as it implements policy-based authorization natively. $\mathcal{P}$ is also able to virtualize rich operating systems, such as Linux or Android, though hardware-based virtualization technologies like ARM's Virtualization Extensions, e.g., to provide a user interface. Since we assume the prover also executes safety- or security-critical applications as native microkernel tasks, which must be strictly isolated from the rich operating systems (and sometimes even the VMM), the prover's system additionally provides hardware-enforced separation mechanism like ARM's TrustZone as the basis for a TEE. $\mathcal{V}$, on the other hand, is a verifier that is generally considered honest and trustworthy in the context of our protocol, however, may try to maximize its benefits by accessing as much data as possible. Like the prover, $\mathcal{V}$ is also equipped with a TPM 2.0 to store sensitive information, e.g., keys.

Without loss of generality, we assume that the prover is an *industrial control system* with at least one rich operating system and a *set of sensors* monitoring its state, environmental conditions, and a fixed number of attached components. In our scenario, the verifier is allowed to log into the rich operating system with credentials provided by the prover's administrator and, hence, is able to interact with certain sensors. However, since the rich operating system is virtualized and device access has to go through the hypervisor, which is controlled by the prover, the verifier can only access said sensors in a very controlled and restricted way. As a result, the verifier cannot be sure that the data from a virtualized device has not been modified. For example, the data of a heartbeat sensor might have been modified by the prover to reflect a system working without any interruptions or anomalies, while the system was, in fact, not available for some time. Obviously, we also have to assume that an attacker might try to modify the data when sent to the verifier's system for further evaluation if that is part of the scenario.

4.2 Attacker Model

For our attestation scenarios, we define an attacker model based on the *Dolev-Yao model* [Dol83][1], a formal model (originally designed for public key protocols), which enables to prove cryptographic security properties of interactive, multi-party protocols, such as a remote attestation. In the model, communication parties in the network can exchange messages that consist of formal terms, which are defined by the protocol and can reveal some of the internal structure of the messages to the attacker, but can also hide other parts from the adversary because of cryptographic protections. According to the model, an adversary can overhear, intercept, drop, and generate any message and is only restricted by the constraints of the cryptographic algorithms used to protect those messages.

In our attacker model, as a consequence, an adversary $\mathcal{A}$ can read messages sent between the prover $\mathcal{P}$ and the verifier $\mathcal{V}$ as long as those messages are not encrypted with a scheme that is still considered secure [Eck14, Sect. 1.3.2, Defense against Attacks, *Encryption*]. The attacker can also initiate communication, create new messages, drop messages, and try to replay old ones. An adversary is also able to manipulate data if its integrity is not protected [Eck14, Sect. 1.3.2 and 8.1.4], e.g., by a *message authentication code (MAC)*. As a result, $\mathcal{A}$ can only decrypt an encrypted message or forge a correct MAC for a modified message if the attacker has access to the correct key [Eck14; Ker83, Kerckhoffs' Principle]. In particular, an attacker is not able to invert cryptographic hash functions and break state-of-the-art MACs or encryption schemes like AES.

In addition, an attacker might be able to compromise the rich OS by exploiting its software components. However, we assume that the attacker is limited to the rich execution environment, i.e., assets inside the trusted execution environment are protected by hardware separation mechanisms. Furthermore, hardware attacks are not feasible, as specified for most remote attestation protocols, which only consider a remote attacker. As a consequence, we can assume that removing hardware components, such as the TPM, the SE, or an Secure Digital (SD) memory card, is not possible. In particular, security mechanisms, which are tightly integrated into the chip, like ARM TrustZone, or provided by a TPM or HSM cannot be removed or otherwise compromised by an attacker. That means we have to assume that the implementation of hardware-based security features, e.g., cryptographic engines or security extensions, and any firmware components for security hardware implemented in software is correct.

As a result, our attacker model allows for a very powerful adversary, which can compromise software components executed on our embedded device as along as this software and its integrity is not verified and protected in any way. In fact, the adversary is omnipotent regarding the network and only restricted by the security guarantees provided by the cryptographic methods used to protect the messages sent between the prover and the verifier.

1 We explicitly use this model in Chapter 8 to formally prove the most critical security properties of our attestation protocol, such as key secrecy or correctness of integrity measurements.

4.3 Summary

To provide context and show the practical benefits of an efficient remote attestation, we defined and described four attestation scenarios, such as secure mobile network access, which highlight specific aspects of a more comprehensive scenario regarding the *secure access to trusted resources*. The purpose of the scenarios is to motivate our research in local integrity verification and efficient lightweight attestation suitable for resource constraint embedded devices, such as mobile phones. Primarily, however, the four attestation scenarios describe the general settings and the parties involved in the integrity verification and attestation protocols, in particular the prover and verifier. In all scenarios, we show that the remote attestation protocol can enable the verifier to reason about the trustworthiness of the prover's system and make an informed decision whether to grant or deny the prover access to a trusted resource, such as a mobile network, a software binary, or sensitive data. In this chapter, we also defined a global attacker model, which not only describes the capabilities of an adversary, but also makes reasonable assumptions about the attacker's range of influence and limitations of the adversary, e.g., regarding cryptographic mechanisms and their implementations. Hence, the results of this chapter are a detailed description of the attestation scenarios focusing on the prover and the verifier as well as a characterization of the attacker focusing on capabilities. Those results are the basis for the security discussions, which are conducted in the following chapter and include specific attacks based on the respective scenario.

5

System Architecture

In this chapter, we present our system architecture for embedded systems, which enables an efficient remote attestation, while separating security-critical software using hardware-based isolation mechanisms in form of dedicated security components like a TPM or hardware features of the SoC. These features include hardware-assisted virtualization capabilities and hardware-based separation mechanisms like ARM TrustZone, which facilitate the implementation of a full TEE, especially in combination with a microkernel-based operating system. In our system architecture, which has some similarities to the one proposed in [Len14], we primarily take advantage of a microkernel, which acts as a near-minimum (i.e., as small and simple as practically possible) hypervisor as well as a software basis for our TEE, because it significantly reduces the amount of privileged code and, more importantly, its complexity.

Furthermore, since modern embedded systems, such as cell phones, become more and more complex and often contain not only an application processor, but also a dedicated baseband processor for their baseband stack, we also consider this multi-processor architecture. As a result, our system architecture features two hardware domains that are described in detail in this chapter, which is structured as follows. In Section 5.1, we present an overview of our system architecture focusing on the two processor domains. Subsequently, Section 5.2 highlights the characteristics of the application processor domain while Section 5.3 describes the baseband processor domain.

Please note that this chapter is a result of a unification of different systems architectures that have already been presented at various security conferences and are published in their corresponding peer-reviewed proceedings [Wag16a; Wag15; Wag12b; Wei14]. Furthermore, some of the results and contributions are joint work with other authors.

5.1 Overview

As shown in Figure 5.1, our system architecture mainly consists of two processor-based domains, the Application Processor Domain (DomA) as well as the Baseband Processor Domain (DomB). While DomA consists of a modern SoC with an application processor and an additional TPM, DomB is an optional domain with an ARM SoC featuring a dedicated baseband processor, which is connected to a Universal Subscriber Identity Module (USIM), a security chip for mobile devices.

DomA, the Application Processor Domain, which is shown on the left of Figure 5.1, features a modern SoC for embedded systems with near-minimum microkernel-based system, which allows to execute virtualized guest operating systems in a distinct Rich Execution Environment (REE). According to the *Nizza* architecture (cf. Figure 5.2), all safety- and security-critical microkernel tasks are executed in an isolated Microkernel Execution Environment (MEE) that is strictly separated by the microkernel, which acts as a near-minimum separation layer or as a hypervisor[1], respectively.

Optionally, our proposed design of a secure system architecture also considers the use of a TEE in the Application Processor Domain, which can enable more advanced remote attestation protocols. As trusted OS, we propose the use of a microkernel-based system, which reduces the complexity and size of the privileged code in the TEE, while still enabling the execution of trusted applications.

In comparison, the Baseband Processor Domain or DomB shown on the right of Figure 5.1 is mainly comprised of a baseband stack running in a Baseband Execution Environment (BEE) on a dedicated processor. Although the entire DomB is optional and mainly relevant for mobile devices, such as smartphones, this domain and its processor can, in practice, be more privileged than the application processor, e.g., by acting as a master for certain critical memory regions or devices.

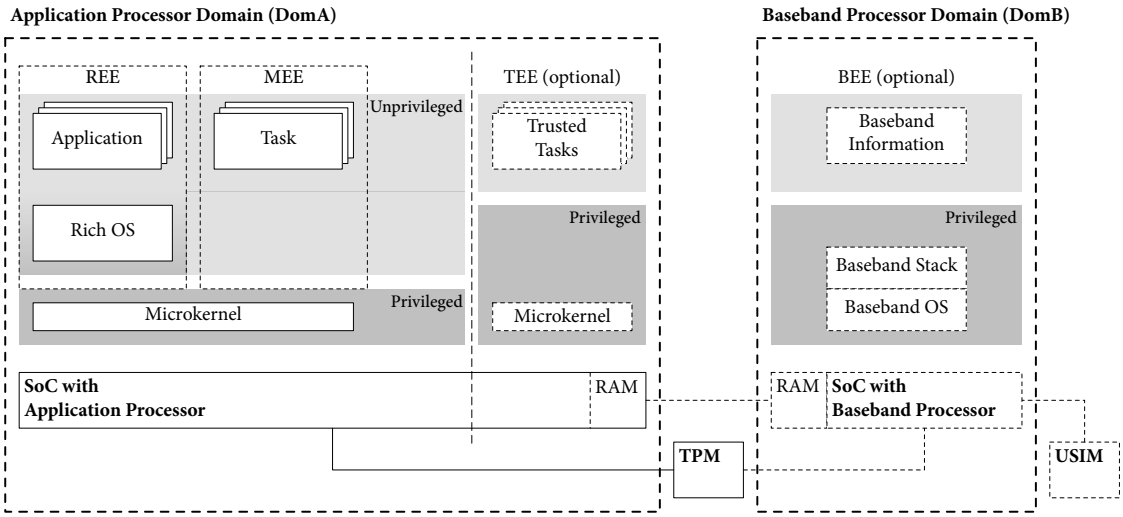

Figure 5.1: Overview of our System Architecture

1 The role of a hypervisor, which puts the kernel of the rich OS in a semi-privileged mode, is shown in Figure 5.4.

5.2 Application Processor Domain

Since most modern application processors, which are part of advanced SoCs for embedded systems, utilize ARM architectures, e.g., because they allow for creating energy efficient, yet capable SoCs, we consider ARM-based embedded systems as the primary targets for our system architecture. Although the core of our Application Processor Domain is based on ideas of the *Nizza* architecture, which means it can be used to create secure software architectures on virtually any modern hardware, our system architecture aims to take advantage of as many hardware security features as possible, be it integrated features of the SoC like TrustZone or dedicated hardware components like an HSM. Consequently, our system architecture aims to combine the advantages of the *Nizza* secure-system architecture, which focuses on separating critical applications from a rich OS using a microkernel, with a hardware security module, more precisely a TPM, enabling hardware-based attestations.

To realize novel hardware-based attestation schemes, our *Nizza*-inspired system architecture, which employs a microkernel as separation/virtualization layer as shown in Figure 5.2, not only significantly reduces the TCB compared to monolithic kernels like Linux, but also enables the implementation of integrity measurement and verification components within the isolated MEE. As a result, our microkernel-based system architecture provides the basis to execute legacy systems in their own REE while isolating critical tasks, like integrity measurement code or the TPM driver.

Furthermore, since modern ARM-based SoCs have been extended with advanced hardware capabilities, such as virtualization or security extensions, as described in Chapter 2, our proposed system architecture naturally evolves during the course of this thesis. Initially, our architecture is heavily influenced by the concept and ideas of paravirtualization and the *Nizza* secure-system architecture, whereas later versions take advantage of hardware-assisted virtualization capabilities, which enable efficient full virtualization of a rich OS. Finally, the implementation of a microkernel-based system within the ARM TrustZone allows for the realization of a full hardware-based TEE, which is optional, but enables more sophisticated remote attestation schemes (cf. Chapter 9).

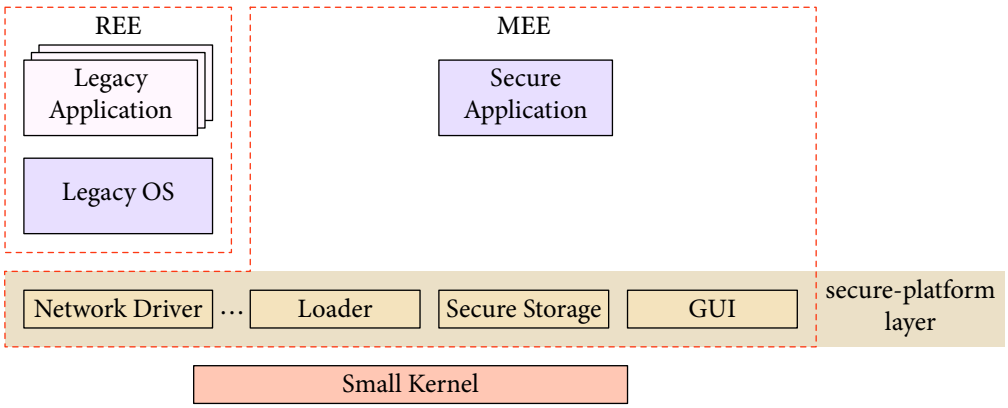

Figure 5.2: *Nizza* Architecture (based on [Här05, Figure 1])

5.2.1 Microkernel-based Operating System

At the core of our system architecture for the DomA, which is inspired by the *Nizza* architecture, a small microkernel-based operating system enables the execution of a legacy/rich OS alongside (security-critical) microkernel tasks as shown in Figure 5.3. In this core system architecture, the microkernel implements a very simple and small separation and virtualization layer that isolates the rich OS and its applications in the REE from the microkernel tasks executed in the MEE, such as OS services, the code to measure the integrity of other components or a TPM device driver.

Depending on the hardware capabilities of the SoC, the rich OS in our system architecture is either paravirtualized (Chapters 7 and 8), which requires modifications to the rich OS (cf. *L4Linux*), or fully virtualized by hardware-assisted virtualization (Chapter 9), where the microkernel acts as a near-minimum hypervisor. This is indicated by a gradient background for the rich OS in Figure 5.3.

In both cases, the microkernel-based OS is responsible to handle the access to privileged and/or security-critical hardware, such as the TPM. As mentioned above, the microkernel also separates security-critical tasks, such as the TPM device driver, virtualizes hardware devices for the rich OS, and enables the implementation of isolated integrity verification and remote attestation components. As a consequence, our TPM-equipped microkernel-based system can securely measure software components and store their integrity values inside the PCRs of the TPM while relying on the strong separation mechanisms of the microkernel and the isolation features of the underlying hardware.

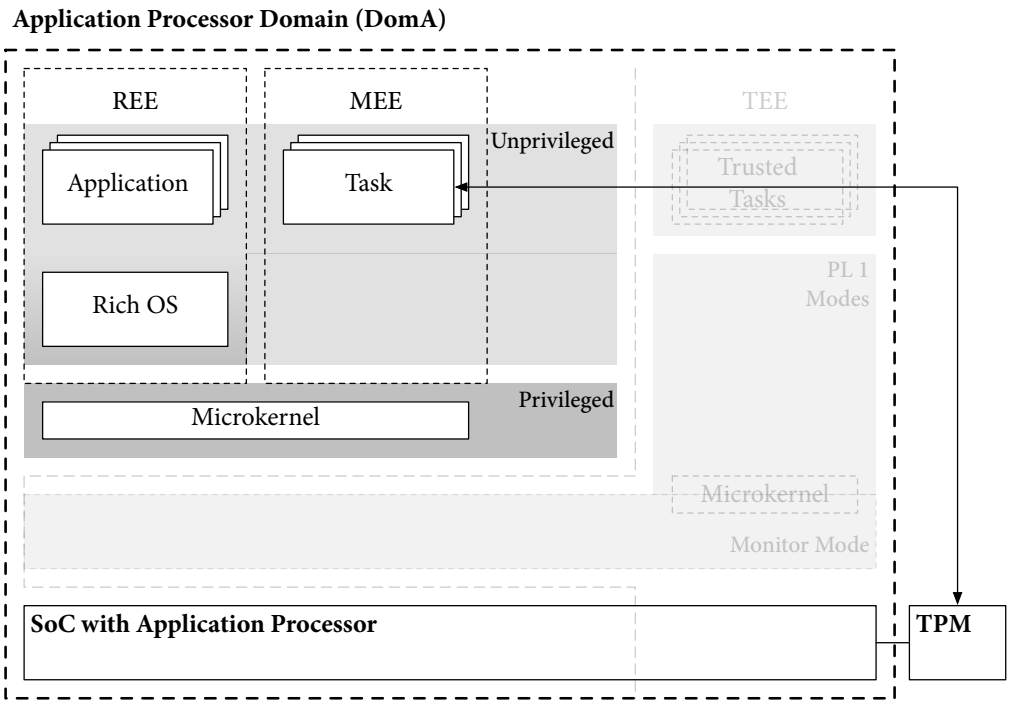

Figure 5.3: DomA: Microkernel-based Partial System Architecture with TPM

5.2.2 Trusted Execution Environment

As an extension, our system architecture also considers the optional use of a TEE, which can provide further hardware-based protections against attacks. As shown in Figure 5.4, an additional TEE, which might also execute a microkernel-based OS, enables the implementation of security-critical microkernel tasks, such as the TPM device driver, within a hardware-separated part of the DomA. Since we focus on ARM-based embedded system, we primarily utilize a TrustZone-based TEE, which provides very simple interfaces: software outside the TEE can either issue an SMC call or trigger a trap, which both lead to defined entry points for the TEE. Those entry points are controlled by the privileged code inside the TEE and, hence, cannot be modified by untrusted software outside the TEE, because any illegal attempt to modify privileged configuration registers is trapped and intercepted through a combination of hardware and software mechanisms.

In comparison to a microkernel-based system architecture, which mainly relies on software isolation techniques to separate individual microkernel tasks within the MEE, this extended system architecture enables the separation and distribution of microkernel tasks between the MEE and the TEE. As a result, it is possible to not only reduce the TCB of the TEE even further, but also assign security-critical hardware, such as a TPM, to the Secure World of the ARM-based embedded system if it supports TrustZone. However, since not all SoCs support a hardware TEE, we will first explore systems without a TEE in Chapters 6 to 8 and focus on benefits of a TEE in Chapter 9.

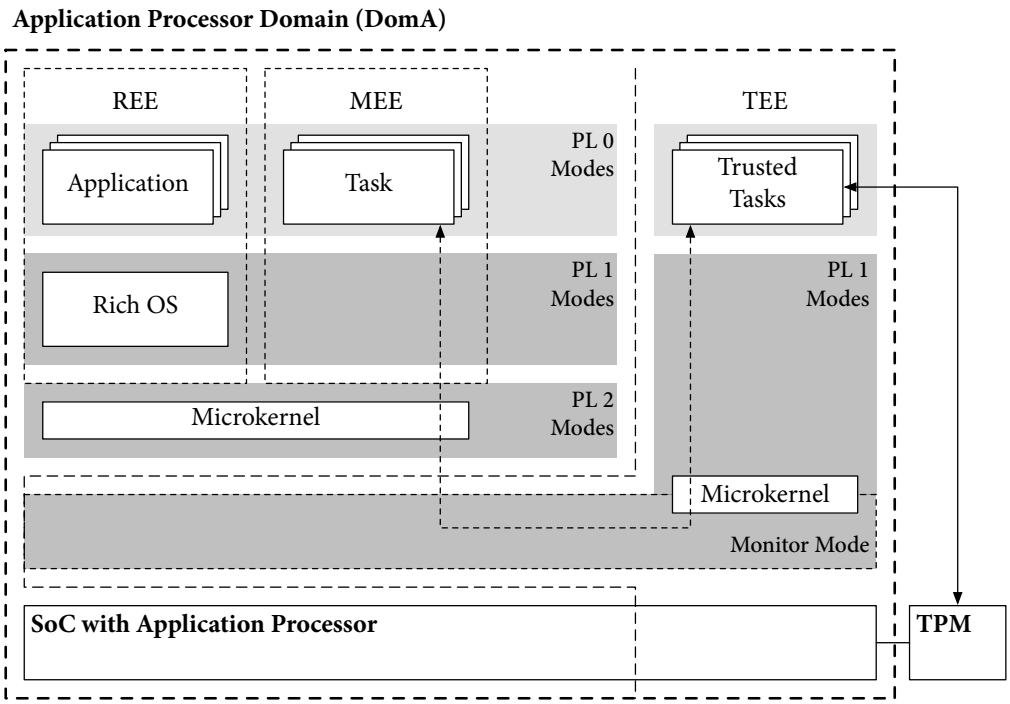

Figure 5.4: DomA: Extended Partial System Architecture with TPM connected via TEE

5.3 Baseband Processor Domain

In addition to the SoC with an embedded application processor, our system architecture also considers the presence of an additional baseband processor, which is common in mobile devices. Although mobile devices in a 3G or 4G network are very heterogeneous mass of devices, most of these devices, e.g., smartphones or 3G USB (Universal Serial Bus) modems, usually have a multi-CPU architecture.

As shown in Figure 5.5, communication between those two processors is often possible, e.g., via a serial connection or shared memory. Interestingly, the baseband processor usually acts as master and not the application processor, which means the baseband software may have full access to sensitive data of the OS executed on the application processor, but not the other way around. Thus, for such a architecture, it is at least as important to measure the integrity of the baseband stack as it is to measure the integrity of the software executed on the application processor.

As a result, our dual-domain system architecture considers the fact that the baseband processor has privileged access to the TPM, which is shared with the application processor in our proposal. Effectively, this means the processor in DomB has access to two separate hardware security modules, which serve, however, two completely different purposes. In addition, the USIM is usually removable and provisioned by the network operator, while the TPM is integrated with the system and owned by the platform vendor, operating system vendor, and/or the user[1].

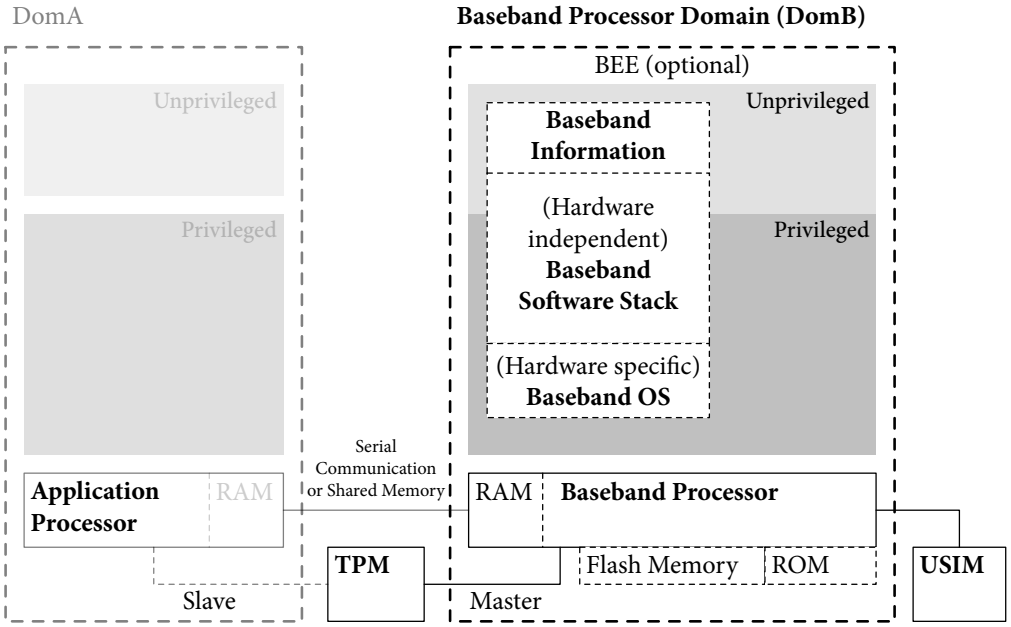

Figure 5.5: DomB: Partial System Architecture with TPM and USIM

1 We do not go into detail about the concerns regarding control of the TPM, although this is an important issue.

5.3.1 Baseband Hardware Architecture

As described above, most devices that enable users to access a mobile network, have some type of baseband hardware, which usually consists of the following parts: radio frequency (RF) front end, analog baseband, digital baseband consisting of a digital signal processor (DSP), and an ARM SoC. In this thesis, we focus on the ARM SoC with the baseband processor, as they communicate with the outside world, although other parts may theoretically be included in a remote attestation, too.

To the baseband processor, an exchangeable USIM is connected as depicted in Figure 5.5, which is a smart card issued by the provider. The USIM holds in its ROM an operating system and the security algorithms for authentication and key generation. In its EEPROM, it stores specific identity information, namely the International Mobile Subscriber Identity (IMSI) and Temporary Mobile Subscriber Identity (TMSI) as well as a pre-shared secret key K_i, which is shared with the provider.

However, today's mobile devices completely lack a comparable secure element to identify the device itself. The existing device-unique International Mobile Equipment Identity (IMEI), for instance, is not stored securely, hence needs to be considered untrusted. For our concept of a remote attestation of the baseband stack, we thus propose to extend the hardware architecture of mobile devices with an TPM. In contrast to existing solutions, we propose to directly connect the TPM to the baseband processor, but share it with the application processor as shown in Figure 5.5.

5.3.2 Baseband Software Components

Today's baseband software usually consists of a small real-time operating system, which is referred to as baseband OS in Figure 5.5 and responsible for parts of layer 1 (hardware specific physical layer) and everything above. For layer 2 and 3, the baseband OS provides an hardware independent software stack with nested implementations of all 2G/3G/4G layers Hence, it is called baseband software stack in Figure 5.5.

For the sake of simplicity, we divide the baseband software in the following two parts: the *baseband binary B*, which contains the baseband OS as well as the baseband software stack, and the *baseband information BI* including device- or operator-specific configuration data. Although there might exist other binaries, such as fail-safe or backup binaries, we assume that only one baseband stack is running on the baseband processor at a given time.

Furthermore, we presume that a boot loader, which is usually very small and simple, loads the baseband binary from memory. We assume that the loader is stored in ROM to prevent unauthorized modifications, securely boot the baseband stack, and authenticate updates that need to have a cryptographic signature created by the *baseband vendor BV*.

For our remote attestation protocol, which is specifically designed for mobile baseband stacks and presented in Chapter 6, we extend the boot loader and the baseband stack with the functionality to communicate with the TPM. Additionally, the USIM software now executes critical parts of the minimal TSS internally.

5.4 Summary

In this chapter, we derived, developed, and described a comprehensive secure system architecture for embedded systems, which incorporates modern hardware- and software-based security features and is designed to enable secure local integrity verification and efficient lightweight remote attestation. To this end, we have combined current research efforts, e.g., focused on microkernel-based systems, with existing security concepts and technologies, such as Trusted Computing, dedicated HSMs, and integrated security features of modern ARM SoCs, e.g., for virtualization or a hardware-based TEE. As a result, our proposed architecture is not only able to separate security-critical software using hardware-based isolation mechanisms, but also handle long-term cryptographic keys, securely store integrity measurements, and provide a trust anchor for security protocols like remote attestation. Thus, based on our system architecture and its particular design, we are able to explore and develop local integrity verification, which rely on a strict separation, as well as more efficient attestation mechanisms, which are presented and described in detail in the following chapters.

6

Attestation of Mobile Baseband Stacks on Dedicated Baseband Processors

Mobile cellular networks provide the necessary infrastructure for location-independent global communications, i.e., phone calls, short messages, and data connections. Many people, businesses, and governments heavily rely on those means to communicate with each other, their customers, and counterparts in different parts of the world. That is why mobile communication networks are considered part of the *critical infrastructures* and need to be secure and highly available, because malfunction or failure can lead to potentially high damage and costs.

In contrast to mobile network nodes like cell towers, which are fully controlled by their operators, the billions of mobile devices are usually beyond their sphere of influence, thus considered untrusted. For that reason, network operators generally issue a smart card, which securely stores the pre-shared authentication information, e.g., keys, to access the network and thereby establishes mutual trust. However, as the complexity and functionality of most software components increase, it becomes easier for an adversary to compromise and remotely control a mobile device [Mull1]. In particular, the baseband stack which is often executed on a dedicated and privileged baseband processor is an interesting attack target, because it implements the software stack to communicate with the mobile network. If an attacker is able to exploit baseband stack vulnerabilities of a large number of mobile devices, those individual devices are not only compromised, but also critical attacks, such as distributed denial-of-service (DDoS) attacks, on the cellular network or related critical infrastructure components like emergency services [Gur16] become possible.

6.1 TPM-based Remote Attestation of Mobile Baseband Stacks

One approach to detect malicious modifications of mobile baseband stacks and protect a mobile network from coordinated attacks by compromised devices is based on TC concepts and a TPM. As described in Chapter 2, a TPM or its mobile version, a Mobile Trusted Module (MTM) [Tru10], can be used to securely handle cryptographic keys, which can be marked as non-migratable to prevent extraction and external usage. A TPM also provides mechanisms for a remote attestation which enables a mobile device platform to prove that no adversary has tampered with its software. However, these security services provided by the TPM primarily focus on operating system and applications that are executed on the application processor. The main reason is that hardware security modules like the TPM, which are attached to a platform (in contrast to smart cards, which are commonly assigned to a person), are usually integrated with the application processor. Hence, a TPM usually cannot be directly used to prove the trustworthiness of the baseband stack and to prevent attacks on the network infrastructure, which are caused by manipulated baseband stacks running on the baseband processor.

In this chapter, we thus present an implicit hardware-based attestation mechanism for mobile baseband stacks according to our Scenario 1 (Secure Mobile Network Access), which is based on a TPM that is connected to the baseband processor. Our attestation enables a mobile device to efficiently prove its trustworthiness towards the network without the need for expensive asymmetric cryptography. Instead, symmetric operations transfer the result of the attestation from the prover to the verifier. Based on the attestation, the mobile network can grant (or restrict) access to mobile network components. As a result, the risk and the potential damage of attacks on the network from devices with a compromised baseband stack can be limited. It even enables the network to enforce a certain baseband version, which prevents attacks that exploit vulnerabilities in a (prior version of the) baseband stack in order to attack the network.

The rest of this chapter is structured as follows. Section 6.2 first briefly explains the infrastructure of mobile networks and discusses possible attacks on those cellular networks based on compromised mobile devices, which motivates the need for a remote attestation of the baseband stack. In Section 6.3, we describe the notation, cryptographic keys, and the concept of our attestation protocol. Finally, we provide a detailed security analysis in Section 6.4. A summary of this chapter can be found in Section 6.5.

Please note that some parts of this chapter, especially the remote attestation mechanism, have already been presented at the *Network and System Security (NSS)* conference in 2012 and are published in its peer-reviewed proceedings [Wag12b]. Although this chapter explores the use of a TPM 1.2, which is still the most widely used version of the TPM, the fundamental idea behind implicit attestation can be transferred to the TPM 2.0, which we will demonstrate in Chapter 9 and shows that the attestation mechanism is flexible in terms of implementation aspects.

6.2 Excursus: Mobile Network Infrastructure and Potential Attacks

In the following excursus, we first briefly describe the architecture of a mobile network and show that mobile devices with a compromised baseband stack can inflict potentially high damage to the mobile network.

6.2.1 Mobile Network Architecture

In general, a cellular mobile network is composed of a Core Network (CN), several Radio Access Networks (RANs), and User Equipment (UE) [3rd12a], such as smartphones or tablets. As shown in Figure 6.1, the UE wirelessly connects to a RAN, which is interconnected with the CN, to access the user's Home Network (HN), usually via a Service Network (SN).

The UE comprises the Mobile Equipment (ME), typically a mobile device with the necessary radio and protocol functionality, and the USIM, which securely stores the authentication information, mainly a shared cryptographic key, to access the mobile network. In order to access the network, the USIM and the network run an Authentication and Key Agreement (AKA) [3rd12c; Nie02], which is a standardized challenge-response protocol that uses symmetric cryptography.

The RAN usually consists of transceiver stations called NodeB (3G), which are managed by a Radio Network Controller (RNC), or eNodeB (short for Evolved NodeB) in 4G, respectively, which embed their own controller functionality. These stations are connected to management components and support gateways, especially to a so-called Mobile Switching Center (MSC) with the Visitor Location Register (VLR) in a 3G and a Mobility Management Entity (MME) in a 4G network.

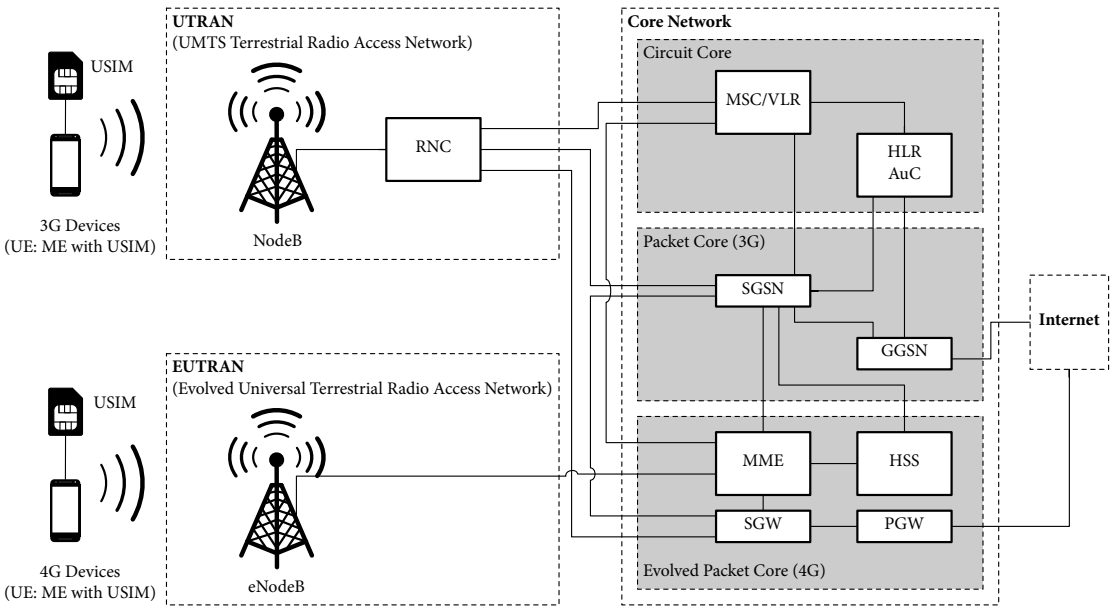

Figure 6.1: 3G/4G Mobile Network Infrastructure (simplified, based on [For10])

Apart from the interface and administration components for RANs, the CN mainly provides the Home Location Register (HLR) or Home Subscriber Server (HSS) depending on the network as well as the Authentication Center (AuC), which manage the relevant information for user authentication and accounting. In particular, the security credentials to access the network, e.g., the standardized pre-shared key K_i between the USIM and the network, are securely stored in the AuC.

6.2.2 Potential Attacks on Mobile Networks (the Verifier $\mathcal{V}$)

As described in the introduction to this chapter, mobile networks are an interesting attack target. They belong to the critical infrastructures, because they provide world-wide communication services though a network of global access nodes. Furthermore, they heavily rely on centralized core components like the HSS and AuC for authentication, as described in the previous section, which could lead to complete network failure in the case of a successful attack against those components.

Based on the centralized architecture of mobile communication networks, an adversary might, for instance, try to attack the network components directly. However, a direct attack against the network is not very promising, since the core components usually only communicate with other trusted network components. Nevertheless, an attacker could still be able to attack the critical components indirectly by launching a DDoS attack.

For such an attack, the adversary needs to successfully compromise a large number of mobile devices, e.g., by sending non-specification-compliant short message service (SMS) messages, which exploit certain weaknesses of the message parser [Mul11]. The attacker could also try to manipulate or replace the baseband stack. If the attacker, for instance, can convince a mobile device to download a malicious version of the baseband binary and install it, the device is compromised as well.

With thousands of mobile devices under their control, attackers could launch a DDoS attack. In contrast to local attacks via low-layer access channels, e.g., RACHell [gru10], such global attacks can potentially produce a system-wide critical overload in the backend components of the mobile network [Tra09]. For example, if all the compromised devices drop off the network [3rd12c, 5.3.8 Detach procedure] and simultaneously re-connect again [3rd12c, 5.3.2 Attach procedure], the HSS might not be able to handle all the authentication requests [Mul11]. Another example is the call-forwarding functionality, which is also handled by the HSS. If an attacker can change the settings for a large number of mobile devices at the same time, the overload could crash the HSS and the network could fail completely.

As a result, we have to acknowledge that attacks on mobile networks based on manipulated baseband stacks are realistic and they can inflict high damage, potentially a complete network failure. Consequently, we explore the use of hardware-based remote attestation to provide cryptographic proof that the baseband stack of a mobile device has not been tampered with. In the following sections, we hence propose an efficient attestation mechanism, which can be integrated into the standardized authentication process for mobile network devices, such as smartphones.

6.3 Providing Verifiable Proof for the Trustworthiness of Mobile Baseband Stacks

To prevent attacks on networks based on compromised or non-specification-compliant baseband stacks, we propose an attestation protocol to verify the trustworthiness of a mobile device before it can communicate with the core components of a mobile network, such as the HSS.

The main idea is that only trustworthy mobile devices are allowed to fully access the critical components of a network. To demonstrate its trustworthiness, the baseband stack running on a mobile phone has to prove its authenticity and integrity. If the attestation procedure fails, the network only allows limited access, e.g., to download a trustworthy version of the baseband stack, which is signed by its vendor, to replace the compromised one. That way, the proposed attestation protocol allows the mobile device to recover from malicious modifications and protects the network from attacks by compromised mobile devices.

In the following sections, we first define the notation in Section 6.3.1 and specify the cryptographic keys in Section 6.3.2. We then explain the concept in Section 6.3.3 and present a description of our protocol in Sections 6.3.4 and 6.3.5.

6.3.1 Notation – Part 1 of 3

In this section, we define the notation for common cryptographic functions, artifacts, and states used in this chapter and throughout this thesis. In particular, we provide definitions for the notation of a cryptographic hash functions, message authentication codes, and a TPM platform configuration. Where required, we will extend the notation in the following chapters.

Cryptographic Hash Functions

A cryptographic *hash function* H is a one-way function with pre-image, second pre-image, and collision resistance that compresses input data of virtually arbitrary length to a fixed sized output of *length* l, that is $H : \{0,1\}^* \rightarrow \{0,1\}^l$. Applying H to *message* m is denoted as $H(\mathrm{m})$, which generates a *hash* h.

Pre-image resistance means that, given a hash value h, it should be hard to find any message m such that $h = H(\mathrm{m})$. Second pre-image resistance means that, given an input message m_1, it should be hard to identify another input message m_2 with $m_1 \neq m_2$, such that $H(m_1) \neq H(m_2)$. This second property is sometimes referred to as weak collision resistance.

A hash function is collision resistant if it is hard to find two different inputs that are hashed to the same output value, i.e., two messages m_1 and m_2 with $m_1 \neq m_2$, such that $H(m_1) = H(m_2)$. In other words, since every hash function with more inputs than outputs will necessarily have collisions (*pigeonhole principle* [Her64]), a cryptographic hash function should ensure that it is hard to find a collision, i.e., two inputs with the same hash value.

Message Authentication Codes

A *message authentication code (MAC)* is a cryptographic value, which is based on a shared symmetric key and allows to verify the authenticity (regarding the data origin) and the integrity of a message. Formally, a MAC algorithm is a function that calculates a message *digest d* with fixed length *l* for a secret key key *K* and a given input *m* with virtually arbitrary size as $MAC(K, m) = d = \{0,1\}^l$.

One method to construct a MAC algorithm is based on cryptographic hash functions, which are one-way functions with pre-image, second pre-image, and collision resistance as described above. As an example, an HMAC function generates a message authentication digest for data *m* based on key *K* as $HMAC(K, m) = H\big((K \oplus opad) \,||\, H((K \oplus ipad) \,||\, m)\big)$, where $||$ denotes a concatenation, $\oplus$ the exclusive OR, *opad* the outer and *ipad* the inner padding [Bel96; Eck14, Sect. 8.1.4, HMAC].

Platform Configuration and the Extension of PCRs

A particular system state is represented by a set of integrity measurements stored in the (usually up to 24) hardware-protected PCRs of an TPM. Such a set of PCR values is often referred to as *platform configuration* $P := (PCR[i_1], \ldots, PCR[i_k])$, where *index* $i \in \{0 \ldots n\text{–}1\}$, $k \leq n$, and n is the number of available PCRs.

After a reset of the platform, the contents of all PCRs is set to zero, i.e., $PCR[i] \leftarrow 0 \;\; \forall i < 24$. To store a fresh *integrity measurement* μ in a PCR with index *i*, the current value inside the TPM is combined with the fresh measurement value using $\mathsf{extend}(PCR[i], \mu)$, which is specified as $PCR[i] \leftarrow H(PCR[i] \,||\, \mu)$ and implemented as `TPM[2]_Extend` in the TPM.

Binding/Wrapping and Sealing

In general, *binding* data *m* to the public portion of an asymmetric TPM key (*pk*) is denoted as $\{m\}_{pk}$ and essentially encrypts data for a private key or even a particular TPM if the key is non-migratable.

Based on the notation for binding data, *wrapping* a cryptographic key *K* with a public key *pk* to a specific platform configuration *P* is denoted as $\{K\}^P_{pk}$ and encrypts a TPM key *K*, e.g., to enable external storage. To decrypt the wrapped key with the corresponding private/secret key *sk*, the *current platform configuration* P' needs to match exactly the specified platform configuration *P*.

To encrypt and bind arbitrary data *m* to a platform configuration *P*, the TPM 1.2 essentially provides a `TPM_Seal` command, which is referred to as seal for the sake of simplicity. With unseal, which is short for `TPM_Unseal`, the TPM 1.2 can decrypt the message *m* if the system is in a state which matches the specified platform configuration *P*. Given a non-migratable asymmetric key $K = (pk, sk)$, we denote the result of *sealing* arbitrary data *m* to the platform configuration *P* with $\{m\}^P_{pk} = \mathsf{seal}(P, pk, m)$. To *unseal* the sealed data $\{m\}^P_{pk}$, it is required that the current platform configuration P' is equal to the specified platform configuration *P*: $m = \mathsf{unseal}(P'{=}P, sk, \{m\}^P_{pk})$. In other words, only if those two platform configurations are cryptographically identical, the sealed blob is decrypted by the TPM returning data *m*.

6.3.2 Cryptographic Keys

In this section, we specify cryptographic keys for our remote attestation of mobile baseband stacks. The keys include a *wrapping*, a *sealing*, and an *integrity key*, which are securely handled in the TPM. The USIM, furthermore, securely stores an *attestation key* in addition to the pre-shared key K_i, which is standardized within the mobile network authentication protocol.

Thus, we define a non-migratable asymmetric wrapping key $K_{wrap} = (pk_{wrap}, sk_{wrap})$ and a non-migratable sealing key $K_{seal} = (pk_{seal}, sk_{seal})$, where pk_{wrap} and pk_{seal} are the public keys while sk_{wrap} and sk_{seal} are the secret keys of the respective keys. Both asymmetric keys are securely generated by the TPM and are supposed to never leave the protected/shielded locations of the TPM, except in encrypted form.

With the public key pk_{seal}, a public signing key pk_{sig}^{BV} is sealed to a *platform configuration* (P_{BL}), where all PCRs are selected, but have the value zero, except for the PCR, which contains the integrity measurement value of the *boot loader BL*. As a result, the sealed key $\{pk_{sig}^{BV}\}_{pk_{seal}}^{P_{BL}}$ can only be unsealed by the boot loader, which is stored in the baseband processor's ROM and executed first. This unsealed key pk_{sig}^{BV} is used in case of an update to verify the *signature (sig) of a baseband update* before installation of a new baseband binary, which we assume is issued by a trusted vendor *BV*.

With the pubic wrapping key pk_{wrap}, an asymmetric integrity key $K_{int} = (pk_{int}, sk_{int})$ is wrapped to a trusted *platform configuration* (P_B) as $\{K_{int}\}_{pk_{wrap}}^{P_B}$. Later on in our protocol, the wrapped integrity key is used to verify the integrity of the baseband. Although *B* denotes the baseband, it does not limit the platform configuration, which is likely to include more measurements.

It is also important to note that the platform configuration P_B invalidates P_{BL}, which means the platform configuration P_{BL} can no longer be used as a valid platform configuration for cryptographic operations, e.g., to unseal a key, but especially to seal a (potentially malicious) public signing key. This can be achieved by extending the PCR of the boot loader *BL* with the unsealed public key pk_{sig}^{BV}.

Together with the authentication value ($Auth_{seal}$) for K_{seal}, both keys are stored in the baseband processor's flash memory in encrypted form (sealed or wrapped) during initialization. However, having $Auth_{seal}$ in flash memory is not a security problem, because the sealing key K_{seal} is cryptographically protected by the platform configuration P_{BL}, hence it is only available to *BL*. The boot loader, in turn, is protected against any modifications, since we assume that it is stored in ROM.

Furthermore, as mentioned in the first paragraph, the USIM holds the pre-shared key K_i and an attestation key $K_{att} = (pk_{att}, sk_{att})$, which are both shared with the AuC. In addition, the USIM securely stores the authentication value ($Auth_{wrap}$) for wrapping key K_{wrap}, which is an important part of the setup for our remote attestation protocol. Since the use of the wrapping key requires authorization based on $Auth_{wrap}$, placing this value in the USIM makes sure that the USIM is involved in the unwrapping of $\{K_{int}\}_{pk_{wrap}}^{P_B}$, which implicitly verifies the platform configuration *B*, i.e., the baseband stack.

6.3.3 Concept and Main Ideas

In contrast to existing hardware-based attestation protocols, which are often based on asymmetric cryptographic operations provided by an TPM, we propose a symmetric approach to efficiently prove the integrity and the trustworthiness of a mobile device, in particular, its baseband stack. As a result, the core of our remote attestation mechanism does not require digital signatures and can be directly integrated into the authentication and authorization protocols for mobile networks.

Like most existing attestation protocols, we rely on an authenticated boot process starting from a CRTM as described in Section 2.1.3.1, where the current software binary calculates a hash of the following binary in the boot chain and extends the integrity measurement value into a PCR of the TPM before executing the measured binary. For the sake of simplicity, the boot loader acts as CRTM in our concept, which is why it must be stored in ROM as indicated in Figure 6.2 (top left), which shows a partial view of our system architecture (cf. Figure 5.5, page 70, DomB with a TPM). Together with the baseband binary (top right), Figure 6.2 also depicts a boot procedure in the top and the relevant hardware components, namely the TPM (bottom left), the flash memory (center), the USIM (right), and the SoC with the baseband processor, in the bottom half.

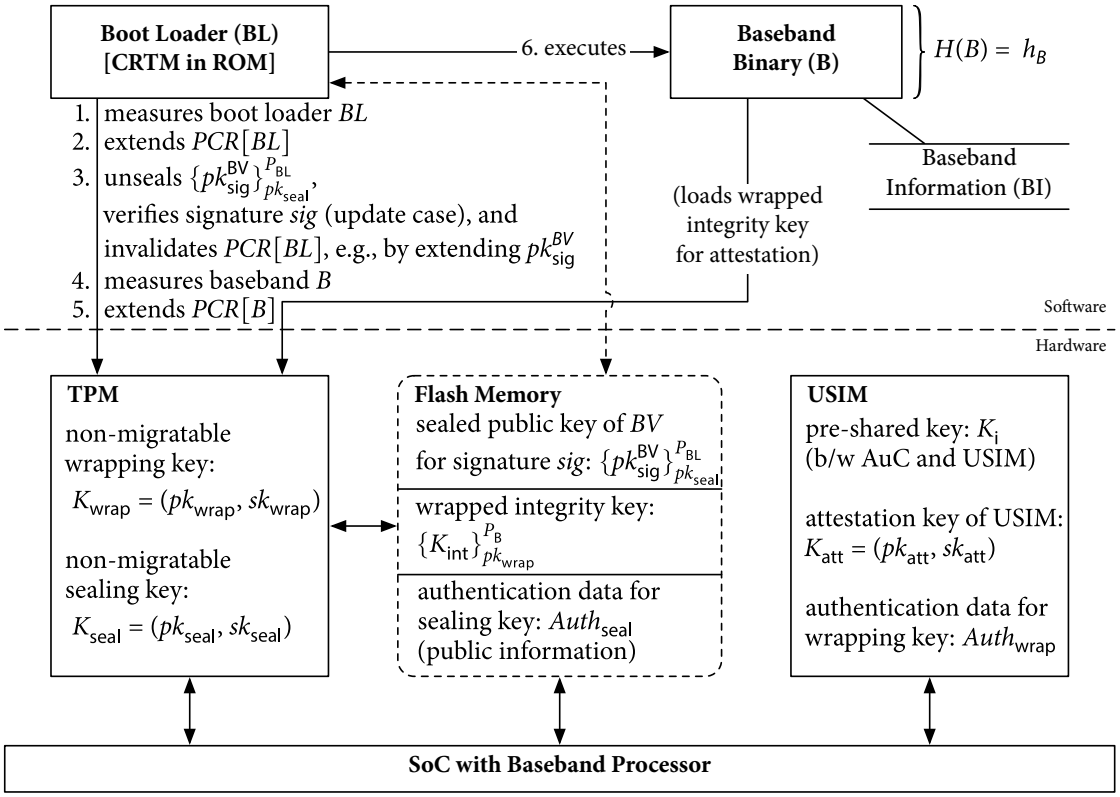

Figure 6.2: Partial System Architecture with a Focus on the Baseband Processor (based on Figure 5.1)

In the boot procedure depicted at the top of Figure 6.2, the boot loader *BL* first measures itself and extends the $PCR[BL]$ (the designated PCR for *BL*, steps 1 and 2) creating the temporary platform configuration P_{BL}. In case of an update, the loader unseals the sealed public signing key $\{pk_{\mathsf{sig}}^{\mathsf{BV}}\}_{pk_{\mathsf{seal}}}^{P_{\mathsf{BL}}}$, verifies the signature *sig* of the new baseband binary with the unsealed $pk_{\mathsf{sig}}^{\mathsf{BV}}$ (step 3) and re-wraps the integrity key K_{int} to the new platform configuration. In the process, the unseal operation implicitly validates the integrity of the boot loader *BL* (represented by P_{BL}) and the public signing key verifies the new baseband with its signature. After that, the boot loader measures the baseband *B* (step 4) and extends the $PCR[B]$ (the designated PCR for *B*, step 5), which creates the trusted platform configuration P_{B} and invalidates the platform configuration P_{BL}. Finally, it executes the baseband binary *B* (step 6).

For a remote attestation, the TPM normally signs the list of PCRs representing the current platform configuration and sends it to a remote verifier. Based on the values and a so called SML, the remote party can then decide whether the platform is still trustworthy. We call this mechanism *explicit attestation*, because the complete measurement log needs to be transferred. For our protocol, however, we adapt the concepts of *implicit attestation*, which does not need to transfer the measurement log. Instead, implicit attestation usually relies on some pre-shared (authentication) information, such as a sealed symmetric key or hash chain [Kra07], which can only be accessed, if the platform is still trustworthy. This approach is, for instance, used to validate whether the boot loader is in a trusted state before a new baseband binary is loaded in case of an update (step 3). So, as long as the prover can successfully authenticate itself, the verifier has implicit proof of the integrity of the prover's system.

However, most existing implicit attestation protocols still rely on relatively expensive asymmetric cryptographic operations, such as signing or unsealing a sealed key. That is why we propose a more efficient approach based on symmetric cryptographic operations to implicitly prove the trustworthiness of a mobile device, especially its baseband stack. The main idea is that the USIM only grants access to the attestation key K_{att} which is necessary to calculate an attestation response for the network if the baseband stack is trustworthy. To prove its trustworthiness, the baseband merely needs to load the integrity key K_{int}, which is cryptographically bound (wrapped) to a trusted platform configuration P_{B} based on a signed baseband stack *B*. Note that the initial wrap operation is necessary only once during initialization and thus has no direct influence on the overall efficiency of our attestation protocol. More important, the TPM only needs to actually unwrap the wrapped key if the key is not yet decrypted and loaded, because the key itself is never used for security critical operations. Finally, to be able to securely verify the baseband stack *B*, we moved the calculation of authentication value, which is an HMAC value based on $Auth_{\mathsf{wrap}}$ and needed to load the key K_{int}, inside the USIM. So, the baseband has to request the correct authentication value from the USIM before it can load the key inside the TPM. If the load operation is successful, the USIM can verify the HMAC-authenticated result that is protected by the long-term authentication value $Auth_{\mathsf{wrap}}$.

6.3.4 Integrity Verification of the Baseband Stack

For the integrity verification or *local attestation* between the baseband stack (prover $\mathcal{P}$'), and the USIM (verifier $\mathcal{V}$'), which is depicted in detail in Figure 6.3, the baseband system (center) loads the wrapped integrity key $\{K_{\text{int}}\}^{P_B}_{pk_{\text{wrap}}}$ into the TPM (left) and the USIM (right) verifies the HMAC-authenticated result. However, since we moved the HMAC calculation for the cryptographic authentication and verification (steps 3 and 10) from the TSS on the baseband SoC inside the USIM, all security critical operations are performed inside one of the hardware secure elements.

In other words, a TSS executed on the host processor is usually responsible to calculate and assemble the necessary parts of the command, such as the parent authentication value ($Auth_{\text{parent}}$) for the wrapping key K_{wrap} (steps 1–4), and to send the complete command structure to the TPM. In our protocol, however, one of the main ideas is to migrate this calculation of the value $Auth_{\text{parent}}$, which authenticates and authorizes the load operation for the wrapping key K_{wrap}, inside the USIM. That way, the passphrase or authentication value $Auth_{\text{wrap}}$ never leaves the hardware-based secure environment of the USIM. As a result, the USIM needs to securely generate the required authentication value $Auth_{\text{parent}}$ on behalf of the usual software component, which requires modifications to the USIM, but can be implemented entirely in software and, hence, easily integrated into the existing USIM firmware.

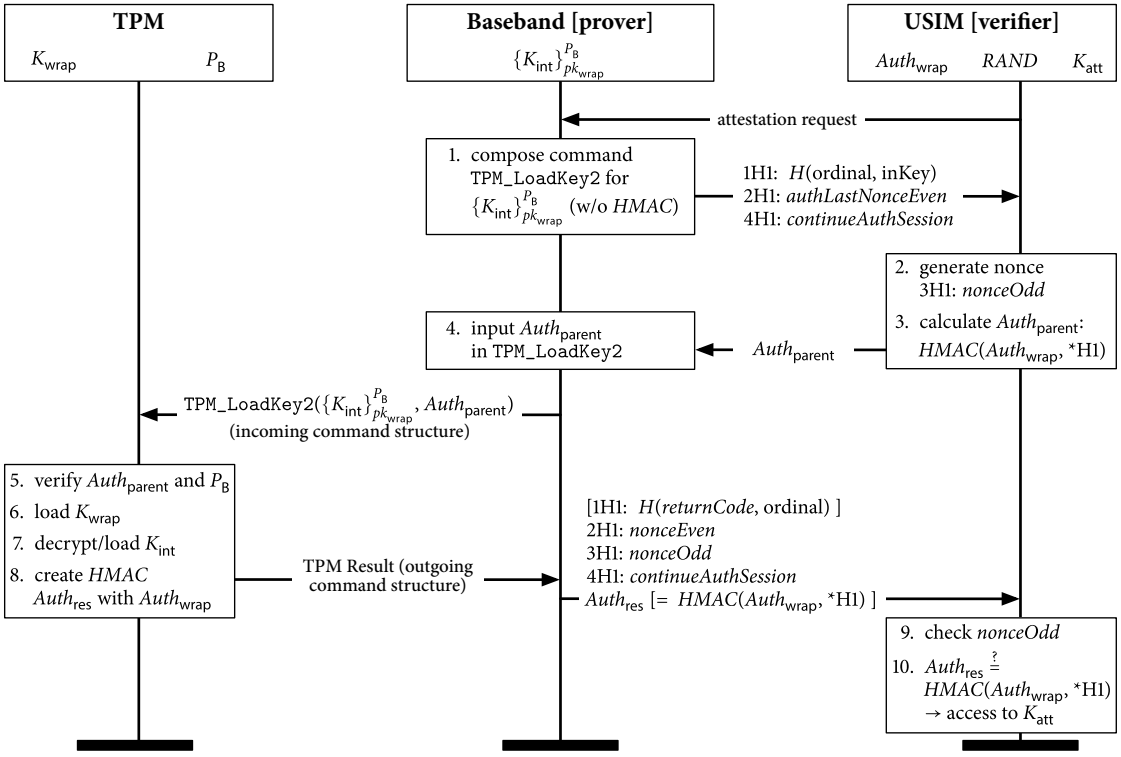

Figure 6.3: Attestation of the Baseband Stack towards the USIM

The value of $Auth_{\text{parent}}$ is generated from concatenated HMAC inputs (denoted by 1H1 to 4H1) as

$$Auth_{\text{parent}} = HMAC(Auth_{\text{wrap}}, 1\text{H1} \,||\, 2\text{H1} \,||\, 3\text{H1} \,||\, 4\text{H1}) \,, \tag{6.1}$$

where

$$1\text{H1} = H\left(\texttt{TPM\_ORD\_LoadKey2} \,||\, \{K_{\text{int}}\}^{P_{\text{B}}}_{pk_{\text{wrap}}}\right)$$

$$2\text{H1} = \textit{authLastNonceEven}$$

$$3\text{H1} = \textit{nonceOdd}, \text{ and}$$

$$4\text{H1} = \textit{continueAuthSession},$$

according to the TCG specification [Tru11, p. 76, Section 10.5 (TPM_LoadKey2)].

When the TPM receives the `TPM_LoadKey2` command, it internally verifies the authentication value $Auth_{\text{parent}}$ and matches the specified platform configuration P_{B} against the current platform configuration P' (step 5). If the equation $P = P'$ holds, the TPM loads the key. For efficiency reasons, the TPM should only verify the pre-conditions, e.g., the platform configuration and authentication data, and not actually decrypt the key if the key is already loaded. The TPM then calculates a result message, which includes a specified return code, e.g., `TPM_SUCCESS`, the *nonceOdd*, and a second HMAC $Auth_{\text{res}}$ to authenticate the response (step 8).

To complete the attestation procedure, the USIM receives the result of the TPM key load operation `TPM_LoadKey2` $\left(\{K_{\text{int}}\}^{P_{\text{B}}}_{pk_{\text{wrap}}}, Auth_{\text{parent}}\right)$ and merely needs to verify the return message: For that purpose, the USIM compares the output *nonceOdd* with the input *nonceOdd*, which must be exactly the same and prevents replay attacks (step 9). By recalculating and checking the HMAC $Auth_{\text{res}}$ (and the return code), the USIM can efficiently verify whether the key was correctly loaded, thus, stating that P_{B} matches P' (step 10). The fresh HMAC $Auth'_{\text{res}}$ is calculated again according to Equation 6.1, where

$$1\text{H1} = H(\textit{returnCode} \,||\, \texttt{TPM\_ORD\_LoadKey2})$$

$$2\text{H1} = \textit{nonceEven}$$

$$3\text{H1} = \textit{nonceOdd}, \text{ and}$$

$$4\text{H1} = \textit{continueAuthSession}$$

as specified by the TCG [Tru11, p. 76, Section 10.5 (TPM_LoadKey2)]. If the load operation has been successful, which is indicated by the *returnCode*, the verifier has implicitly proven that the baseband stack is still unmodified and has not been compromised. As shown in Figure 6.3, the USIM now allows access to the attestation key K_{att}, which is limited to the current AKA protocol run indicated by the *random number (RAND)*.

6.3.5 Generation of Authentication Vectors

Based on the result of a local baseband attestation, the USIM (now prover $\mathcal{P}$") is able to provide proof of the baseband's trustworthiness towards the network (verifier $\mathcal{V}$"). We only need to slightly modify the authentication vectors (AVs) used in the AKA protocol. Depending on the network type (3G or 4G), the AVs are usually generated as

$$UMTS\ AV := (RAND \,||\, XRES \,||\, CK \,||\, IK \,||\, AUTN) \text{ or} \tag{6.2}$$

$$EPS\ AV := (RAND \,||\, XRES \,||\, K_{\mathsf{ASME}} \,||\, AUTN)\,, \tag{6.3}$$

where $RAND$ is a random number, $XRES = \mathrm{f2}_{\mathsf{K}}(RAND)$ is the pre-calculated (and expected) authentication result, $CK = \mathrm{f3}_{\mathsf{K}}(RAND)$ is a confidentiality and $IK = \mathrm{f4}_{\mathsf{K}}(RAND)$ an integrity key, and $AUTN := SQN \oplus AK \,||\, AMF \,||\, MAC$ an authentication token ($\oplus$ means the exclusive OR). These cryptographic components are calculated as depicted in Figure 6.4, where f1 and f2 are MACs and f3 to f5 as well as KDF are key derivation functions (KDFs).

In our concept, we add an *expected attestation value (XATT)* to the AV, which allows the Service Network to verify the trustworthiness of the mobile device. This additional attestation value is generated from the random number $RAND$ with a dedicated attestation key K_{att} (which is only available, if the local baseband attestation has been successful), that is

$$XATT = HMAC(K_{\mathsf{att}}, RAND)\,. \tag{6.4}$$

Since the symmetric attestation key K_{att} is only known to the HSS (and the USIM, of course), the Home Network has to pre-calculate the HMAC for the Service Network. That way, the SN can compare the *attestation value (ATT)* from the USIM with the expected attestation value $XATT$ without the knowledge of K_{att}.

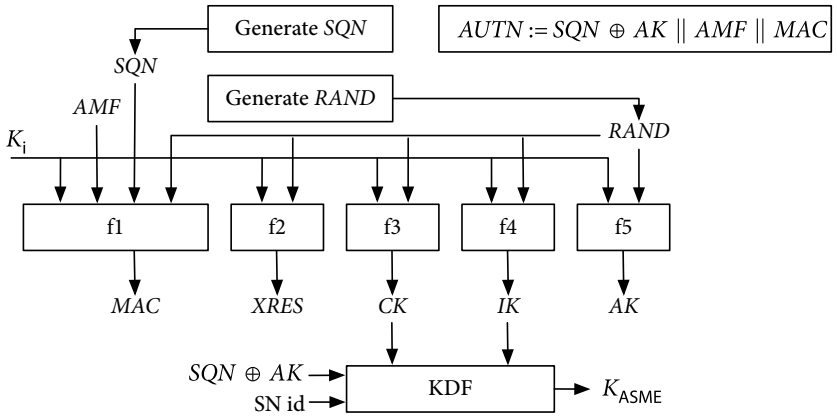

Figure 6.4: Generation of Authentication Vectors (adapted from [3rd12b; 3rd12c; For10])

We also send a second *attestation value for the baseband and the baseband info (ATTB)* from the mobile device to the Home Network, which is generated based on the *hash value of the baseband binary (h_B)* and some information *BI*, for instance, about the version, state, and configuration of the baseband. These values can be used by the home network to further evaluate the baseband stack and enforce a certain version or configuration.

As shown in Figure 6.5, we define the following attestation-based access policy: If the response $RES = f2_{K_i}(RAND)$ from the USIM matches the expected response *XRES* and the attestation value $ATT = HMAC(K_{att}, RAND)$ also corresponds with the expected value *XATT*, the mobile device can fully access the network (steps 8–9). However, if the MME attestation fails, the network only grants limited access, e.g., to download a signed recovery version to replace the modified baseband stack. By sending the hash of the baseband h_B and the baseband information *BI*, which are protected by the second attestation value *ATTB* (step 7), the Home Network can evaluate the configuration of the baseband in detail (steps 10–11). As a consequence, particular services or operations involving critical network components could be allowed (or denied). The Home Network could even enforce a certain baseband version by simply evaluating the baseband version in *BI* and restricting access for unsupported versions.

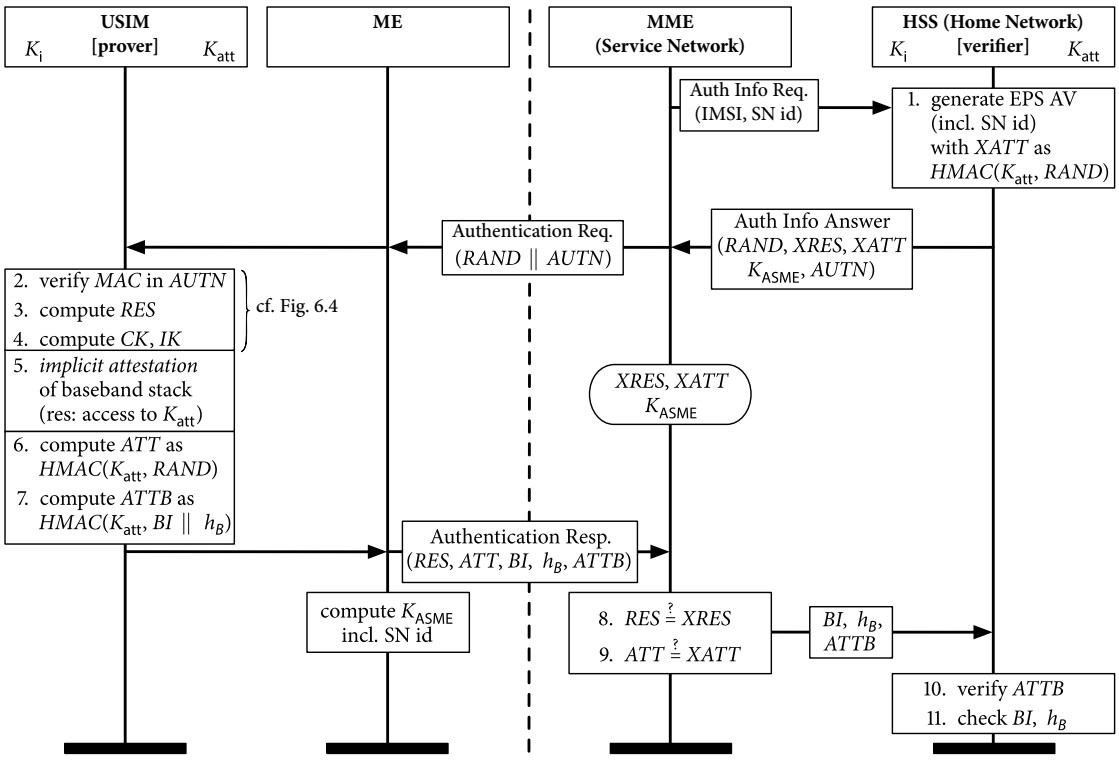

Figure 6.5: AKA-based Attestation of Baseband/USIM towards the Network (simplified)

6.4 Informal Security Analysis and Limitations

In this section, we now analyze the security of our proposed attestation protocol according to our attacker model presented in Section 4.2. Hence, we mainly consider software-based remote attacks, whereas hardware attacks, such as TPM cold boot attacks, are by nature and definition out of scope. In addition, we discuss the security-related limitations and restrictions of our implicit attestation protocol as presented in this chapter.

As most existing protocols, we start from the premise (for the sake of simplicity) that the platform configuration is established during authenticated boot and reflects the actual state of the baseband stack at any time. These two assumptions, which can be considered limitations of our approach, show that we only deal with load-time integrity at this point and result in the fact that the baseband binary should be static most of the time to take advantage of our implicit attestation. However, the second assumption does not mean that the baseband binary cannot be updated, which would be an unreasonable requirement, but the update mechanism obviously must generate a suitable integrity key, which is cryptographically bound to the new platform configuration. A way to create this new key will be described in Chapter 8, which addresses Scenario 3 (Secure Update and Recovery) and can be used for updates of the baseband binary as well.

We also assume that it is not possible to forge a trusted platform configuration, e.g., by exploiting bugs, such as buffer overflows, through code-reuse attacks, or return-oriented programming (ROP), although that would likely require a CFI protection mechanism as well as either a periodical or an on-demand measurement architecture, such as IBM's IMA [Sai04]. As a result, we restrict the attacker at this point for the sake of simplicity and discuss the use and integration of IMA into our system architecture in the following chapter, which also shows a way to isolate such a security-related system component.

In the following paragraphs, we discuss attacks on the adversary's three main targets: the cryptographic keys, the baseband binary, and the attestation value. By extension, these three attack targets indirectly also include the network, because a successful attack against one of these targets would likely be the key aspect for an attack against the mobile network.

Attack 1a: Extraction of Cryptographic Keys

In our first attack scenario, the adversary $\mathcal{A}$ attempts to extract the cryptographic keys. However, that is not possible, because the symmetric keys K_{i}, K_{att}, and the authentication data $Auth_{\mathsf{wrap}}$ are securely stored inside the USIM. The asymmetric keys K_{wrap} and K_{seal} are non-migratable, thus never leave the TPM. As a result, all cryptographic keys are securely handled by a hardware security module, the TPM or the USIM, at all times and since we trust that the hardware security modules are implemented correctly and securely, those keys are protected.

Attack 1b: Replacement of Cryptographic Keys

The attacker could also try to replace the sealed or wrapped keys, namely the sealed public signing key $\{pk_{\text{sig}}^{\text{BV}}\}_{pk_{\text{seal}}}^{P_{\text{BL}}}$ and the wrapped integrity key $\{K_{\text{int}}\}_{pk_{\text{wrap}}}^{P_{\text{B}}}$. In the first case, the sealed key can only be unsealed while the boot loader *BL* is executed and P_{BL} is not yet invalidated by P_{B}. Since the boot loader is stored in ROM and acts as a CRTM, it is always executed first and cannot be modified by software. In addition, the platform configuration P_{BL} is hidden from any attacker, because it is supposed to be invalided by P_{B}. As a result, the adversary cannot seal a different public key, which would successfully verify a signature for a manipulated baseband update. In the second case, the attacker might try to wrap an integrity key to an insecure platform configuration, e.g., with no PCRs selected, to manipulate the baseband stack without the attestation protocol noticing. However, this is not possible, because the authentication data $Auth_{\text{wrap}}$ is stored inside the USIM.

Attack 2: Manipulation of the Baseband Binary

In the next attack scenario, the adversary actually manipulates the baseband binary to attack the network. However, since the baseband binary is measured by the boot loader before it is executed, the manipulation is reflected in the platform configuration P'. As a result, the TPM cannot load the wrapped key, the attestation fails (because of the return code), and the USIM denies access to attestation key K_{att}. That mean the attestation value *ATT* cannot be calculated correctly and the network only allows fail-safe access to network, which can effectively prevent the attack. In the case, where the attacker manipulates the baseband binary, but replays an old TPM result message in order to make the USIM believe that loading the integrity key was successful, the USIM simply needs to check the *nonceOdd* (Figure 6.3, page 82, step 9). Since this value is random and only known to the USIM (Figure 6.3, step 2), the attestation fails, because the replayed TPM result message has a different *nonceOdd*.

Attack 3: Forging the Attestation Value *ATT*

In our last attack scenario, the adversary might try to forge the attestation value *ATT* in order to access and attack the network with a compromised baseband stack. However, if the attacker is able to capture the random number *RAND*, the authentication token *AUTN*, and the authentication response *RES*, it is still not possible to calculate the correct attestation value. The attacker has no knowledge about the attestation key K_{att}, which is securely stored in the USIM. Even in the case where the adversary combines the authentication response *RES* with an old attestation value ATT', the network only grants limited access. Since the attestation value *ATT* is an HMAC over the current random number *RAND*, the pre-calculated attestation value *XATT* does not match ATT' (Figure 6.5, page 85, step 9), so the attestation fails. The network only allows fail-safe access and an attack on the critical network components, such as the HSS, is prevented.

6.5 Summary

With today's increasing use of mobile communication, which might even rise in the near future, attacks from a larger number of mobile devices with a compromised baseband stack can be a serious threat to mobile networks and connected systems. To limit the risk of potentially high damage, we presented a hardware-based implicit attestation protocol focusing on the baseband processor, which enables mobile devices with an TPM to prove the trustworthiness of their baseband stack towards the mobile network.

Furthermore, we have shown that the network is able provide different access levels based on the result of the attestation. That way it can even enforce, for example, a certain baseband version. Finally, our security discussion explains how the network can limit exposure to compromised baseband stacks and reduce the risk of attacks from manipulated devices.

With our attestation protocol, which reports the integrity of the baseband stack, we provide a mechanism to address the attacks against the mobile device, in particular against the baseband stack, as described in Scenario 1 (Secure Mobile Network Access). We also showed how our attestation mechanism can enable the mobile network operator to detect compromised devices on network connect and quarantine those devices using different access and trust levels.

Based on the results of this chapter, which primarily focused on the baseband stack executed within the baseband processor domain, the research efforts presented in the following chapters of this thesis detail implicit attestation protocols focusing on the application processor domain. As proposed in Chapter 5, we link both domains cryptographically and enable a comprehensive attestation by sharing the TPM if the embedded system is equipped with a baseband processor, which then acts as a master. Otherwise the application processor, more precisely, the microkernel-based system in the MEE or TEE, controls the TPM.

7

Attestation of a *Nizza*-inspired System and Secure Loading of Microkernel Tasks

With this chapter, we shift the focus to the application processor, which executes a robust microkernel-based operating system and also uses the TPM to securely store keys and integrity measurements. As described in Chapter 2, microkernels can be considered much more resilient compared to monolithic kernels, because they are very small in terms of code size, have a reduced API, and only provide the most basic mechanisms to enable the implementation of an operating system: address space and thread management, scheduling, and IPC. Unfortunately, microkernel-based systems in practice often rely on a static composition and configuration of their software components in order to fully ensure safety and security. That means dynamic loading of remote binaries is usually not possible or allowed in safety- or security-critical systems, such as vehicles or smartphones used for secure communications. However, since the ability to dynamically and securely load remote binaries can be a desirable property with legitimate benefits, e.g., for offline banking as described in Scenario 2 (Secure Loading), a system which provides that ability must be able to verify the authenticity and integrity of the binary to preserve its trustworthiness. A backend, in turn, must be able to verify that the system is still trustworthy after the binary is loaded. Hence, this chapter explores implicit and local attestation for a microkernel-based OS and secure loading of remote microkernel binaries.

Please note that the research conducted for this chapter is joint work with Weiss et al. [Wei14]. Large parts of this work have been presented at *TrustCom* and are published in the corresponding peer-reviewed proceedings.

7.1 TPM-based Secure Loading of Microkernel Applications

As described in the previous chapters, one approach to measure, verify, and report the integrity of software binaries relies on a combination of authenticated boot, a hardware security module, such as a TPM [Tru11], and remote attestation. However, the authenticated boot mechanism only collects integrity measurements for components in the boot chain and does not prevent the execution of (potentially malicious) binaries. That means, after some time those integrity values stored inside the TPM do no longer represent the actual current platform state, since malicious binaries might have been able to compromised the system in the meantime.

To overcome this limitation, IMA [Sai04] has been proposed for Linux-based systems, where integrity values are calculated during runtime whenever a binary is loaded. Unfortunately, IMA focuses on Linux-based systems and originally cannot prevent loading of remote binaries from an unknown source, which presents a major threat to the system's integrity. Thus, loading such remote binaries is not acceptable for systems with security-critical applications, e.g., banking applications on smartphones, not to mention safety-critical applications for in-vehicle systems or airplanes.

In this chapter, we present a microkernel-based system architecture with a TPM-based integrity verification and attestation service that allows to securely load remote binaries and report the trustworthiness of the system to a remote party, such as a backend server. The proposed mechanism provides the means to establish the authenticity of a remote binary, measure its integrity at load-time, and generate verifiable proof of the system's integrity for a remote verifier. By implementing the integrity verification and secure loading service as native microkernel tasks in the MEE, we can also separate it from the rest of the system, especially the rich OS in the REE. Hence, compared to IMA, our mechanism does not rely on the trustworthiness of a rich OS. That way, our approach not only adopts the main ideas of IMA for Linux-based systems to a microkernel-based system, it also reduces the TCB for the integrity measurement components. Furthermore, our approach only depends on unprivileged user-space drivers needed for the TPM, but does not require, for instance, USB or network card drivers, which are usually built into a rich hypervisor running in privileged supervisor mode of the processor. In our system architecture, we take advantage of the rich OS to provide the necessary network stack for communicating with a backend server. As a result, our embedded system has network connectivity, can provide integrity verification similar to IMA, and use attestation to report integrity measurements to a verifier.

The rest of this chapter is structured as follows. In Section 7.2, we describe further details of our microkernel-based system architecture. In Section 7.3, we then present the concept of our integrity verification, secure loading, and attestation mechanism. Finally, details of our implementation are provided in Section 7.4, while the results of our evaluation are presented in Section 7.5. Section 7.6 concludes the chapter with a summary.

7.2 Microkernel-based System Architecture with Fiasco.OC

Based on our generic system architecture presented in Section 5.2.1, this section details an initial microkernel-based system architecture with Fiasco.OC. As shown in Figure 7.1, the microkernel Fiasco.OC runs in privileged mode directly on the application processor. All non-essential system components, e.g., memory management, I/O services, or file system drivers, are implemented as user-space tasks in the MEE and provide services or device access to other tasks (*clients*) through IPC. In addition, the L4 kernel interface implementation Fiasco.OC includes an object-capability model, which is used for securing access to kernel objects [Lac09]. In our concept, we make use of capabilities, for instance, when we establish communication channels between certain tasks.

7.2.1 REE: *L4Linux* as Rich OS

As described in Chapter 2, a microkernel like Fiasco.OC usually allows to execute a (modified) rich OS kernel, e.g., a paravirtualized Linux like *L4Linux* [Här97], in user space on top of the microkernel, which has to strictly separate it from the native microkernel tasks to protect the microkernel execution environment. In our architecture, that fact enables us to reuse the communication stacks of the rich OS for external communication, since we assume that the network is untrusted anyway. As a result, we keep the TCB small, as we do not rely on network drivers and communication stacks, e.g., for USB, TCP (Transmission Control Protocol), or UDP (User Datagram Protocol), inside the microkernel execution environment.

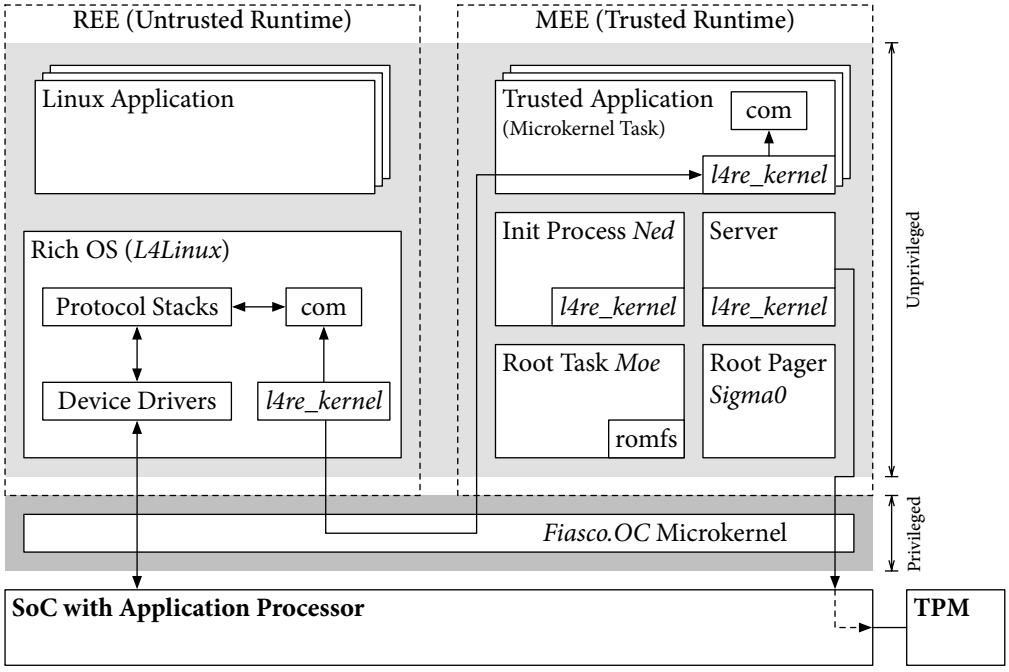

Figure 7.1: Our Initial Microkernel-based Architecture in DomA (based on Figure 5.3)

7.2.2 MEE: L4Re Components as Trusted Microkernel Runtime

The software runtime environment for the Fiasco.OC microkernel, which is known as L4Re [Lac09], primarily consists of *Moe*, *Sigma0*, *Ned*, and *Io* as described in Chapter 2. As the name suggests, L4Re establishes a microkernel runtime in the MEE, which provides the basis for the execution of other native microkernel tasks, e.g., a TPM device driver, or a paravirtualized rich OS like *L4Linux*.

L4 Runtime Environment

In L4Re, the root task *Moe* is the first user-space task started by the kernel as discussed in Chapter 2. *Moe* inherits all resources, which are not required by Fiasco.OC to implement its kernel functionality. The memory management, however, is delegated to *Sigma0*, which acts as the root pager for the entire system. That means *Sigma0* manages those memory regions, which are not claimed by the microkernel or assigned to other application-specific pagers, such as *L4Linux*, which operates as memory pager for the Linux applications.

Based on the combination of Fiasco.OC, *Moe*, and *Sigma0*, the init process *Ned* is able to start other applications by loading a *L4Re runtime binary (l4re_kernel)* in a new application context, which is also known as a *task*. Such a task is represented by a collection of address spaces and consists of at least a memory address space and an object address space [TUD11a]. From within the task, the loader code of *l4re_kernel* is responsible to load the actual application binary, which in turn means that each new application has the L4Re runtime code for communication and memory abstractions mapped into its virtual memory. To enable communication with other tasks, *Ned* provides the ability to configure IPC channels, which includes the creation/transfer of capabilities.

Object-capability Model to Secure the MEE

To provide proper access control to certain kernel objects and IPC channels, our secure loading concept and the corresponding prototype implementation utilize the object-capability model implemented in the Fiasco.OC microkernel. In particular, our system grants and protects access to shared memory (more precisely, *L4Re Dataspaces*) and establish IPC channels according to our protocol using capabilities. This security mechanism is discussed in Section 7.4.4 and imperative to our secure loading concept.

Trusted Microkernel Runtime in Contrast to a Dedicated Hardware TEE

Although the idea and architecture design for a trusted microkernel runtime is based on the high-level specification for a TEE [Bai11; Glo11], the microkernel runtime is not a TEE with hardware-based protection mechanisms as described in Section 2.2.4. Instead, the microkernel runtime defined in this chapter is considered an instance of our MEE as shown in Section 5.2.1, which is mostly software-based and relies on a microkernel as separation layer. As a result, sophisticated attacks like *side channel attacks* might be possible (cf. [Wei16]), which is why we use a TPM that provides elaborate security mechanisms to protect long-term keys on our embedded device.

7.3 Concept of our Secure Loading and Attestation Mechanism

In the following sections, we present the concept of our integrity verification, secure loading, and attestation mechanism. In Section 7.3.1, we first define the relevant cryptographic keys and discuss the necessary steps for the provisioning of our protocols. In Section 7.3.2, we then specify the relevant components involved in verifying and securely loading remote binaries. Finally, we describe the measurement and the remote attestation (challenger) protocol, which provides local attestation abilities inside the microkernel execution environment, in more detail in Section 7.3.3.

7.3.1 Cryptographic Keys and Their Provisioning

Before we describe our secure loading and remote attestation protocol, we define the relevant cryptographic keys for our protocols in this section. Specifically, we define two non-migratable asymmetric keys for the MEE, a binding key $K_{\text{bind}} = (pk_{\text{bind}}, sk_{\text{bind}})$ and an attestation identity key $K_{\text{aik}} = (pk_{\text{aik}}, sk_{\text{aik}})$[1], and their respective authentication values $Auth_{\text{bind}}$ and $Auth_{\text{aik}}$, which are only available to the MEE.

As defined in the specification, the TPM 1.2 provides the command `TPM_CreateWrapKey` to generate a key such as K_{bind}, which not only encrypts the new key with a parent key in the TPM, e.g., the SRK, but can also cryptographically bind the key to a certain platform configuration P. In our concept, we specify P_{MEE} comprising of the integrity measurements of Fiasco.OC and its microkernel runtime including *Sigma0*, *Moe*, *Ned* and others. As a results, our wrapped binding key, which is encrypted with the SRK and cryptographically bound to P_{MEE}, is denoted as $\{K_{\text{bind}}\}^{P_{\text{MEE}}}_{pk_{\text{SRK}}}$.

To create K_{aik}, on the other hand, we use `TPM_MakeIdentity`, which generates an AIK that acts as a pseudonym for the EK. As a signing key, K_{aik} is used to create a TPM quote, which mainly contains a signed composite hash of selected target PCRs, which can be evaluated in an attestation.

For the provisioning of K_{bind} and K_{aik}, our system boots into a secure provisioning state, where only trusted software is executed and while the system is still in a secure environment. In the provisioning state, the TPM service task issues the `TPM_CreateWrapKey` command with the trusted platform configuration P_{MEE} as parameter to create a wrapped binding key K_{bind} and the command `TPM_MakeIdentity` to generate the AIK. Afterwards, the public keys pk_{bind} and pk_{aik} of the corresponding TPM keys are stored in the backend system. If the backend later encrypts data with pk_{bind}, it is ensured that the data can only be decrypted on the device with that specific TPM.

To verify code signatures, we define a certificate $Cert_{BS} = \{BS, pk^{BS}_{\text{cert}}\}_{sk^{\text{CA}}_{\text{sig}}}$ for the backend system, which includes the public portion of its code signing key $K^{\text{BS}}_{\text{sig}} = (pk^{\text{BS}}_{\text{sig}}, sk^{\text{BS}}_{\text{sig}})$. The root certificate $Cert_{\text{CA}} = \{CA, pk^{\text{CA}}_{\text{cert}}\}_{sk^{\text{CA}}_{\text{sig}}}$, which enables the verification of $Cert_{\text{BS}}$, is securely stored in $\mathcal{P}$'s ROM. We also define a certificate $Cert_{\text{AIK}} = \{AIK, pk^{\text{AIK}}_{\text{cert}}\}_{sk^{\text{CA}}_{\text{sig}}}$ for K_{aik} from the CA.

1 Since it is not recommended/possible to use the TPM's EK directly in an attestation (because of privacy concerns and since the EK is an encryption key), the non-migratable AIK acts as alias for the EK as described in Section 2.1.2.3.

7.3.2 Loading External Microkernel Applications

In this section, we describe our concept for securely loading external microkernel application binaries from a trusted remote system into the microkernel runtime provided by the MEE, which is depicted in Figure 7.2. The figure shows our architecture as well as the secure loading protocol, which is described in more detail in Figure 7.3, including all relevant software and hardware components, which enable measuring and loading an external *binary (bin)* in its final context inside the MEE.

On the left, Figure 7.2 shows a trusted backend system, which provides the new remote binary that will be securely loaded by the microkernel runtime of the microkernel-based execution environment. The part on the right depicts our microkernel-based system with the REE executing our designated *Rich OS (R)* and the MEE with the native microkernel tasks. Those tasks include *Moe*, *Sigma0*, *Ned*, as well as our secure loading components, e.g., a loader service, a TPM service, and an integrity service, which are introduced and described in detail in the following paragraphs.

In addition to the backend system and the embedded SoC, which basically represents the DomA with its applications processor, Figure 7.2 also shows a TPM, which is securely handles the private portion of the encryption/decryption key sk_{bind} and the attestation identity key sk_{aik}. Since those keys are non-migratable, they are cryptographically bound to one particular TPM and device, which ensures that the new binary is only executed in the MEE of a known system.

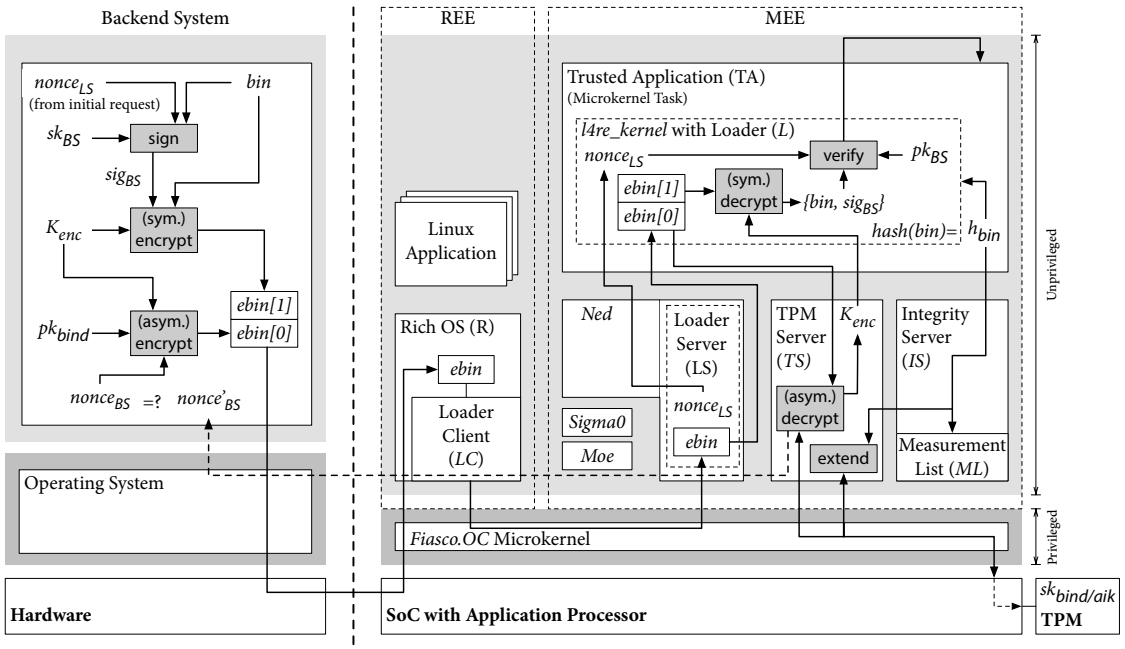

Figure 7.2: Secure Loading of a Remote Binary Into our MEE

Software Components for our Secure Loading Procedure

Before we discuss our secure loading procedure in detail, we give an overview of the software components involved in our protocol (in order of the protocol flow shown in Figure 7.2):

Backend System (BS): The backend system *BS* is a server, which acts as the remote verifier $\mathcal{V}$ and provides an external binary to our microkernel-based system via an untrusted network, such as the Internet. For a secure deployment, *BS* signs the binary *bin* (and a *nonce*) with its private key sk_{sig}^{BS} and encrypts it with the MEE's public key pk_{bind} using a hybrid encryption. More precisely, the binary is encrypted with an ephemeral symmetric encryption key K_{enc} while that key (and a $nonce_{BS}$), in turn, is encrypted with pk_{bind} using asymmetric cryptography. Please note that the corresponding private key sk_{bind} is only accessible to the MEE, which controls and protects $Auth_{\text{bind}}$, and never leaves the TPM, because K_{bind} is non-migratable.

Loader Client (LC): The loader client *LC* is part of *R*, the rich OS in the REE. *LC* acts as a relay between the backend system *BS* and the MEE and basically forwards the encrypted remote binary, which it requests from the backend system, to the microkernel runtime through a defined interface, which we describe in the following sections.

Loader Server (LS): The loader server *LS* handles requests from *LC* and is responsible for starting new applications. For that purpose, it copies the runtime binary *l4re_kernel* into a new application context and grants access to the encrypted binary. Please note that *LS* is also responsible for generating the MEE's $nonce_{LS}$ to later enable freshness verification.

Loader (L): The loader *L* is the part of the *l4re_kernel*, which is executed by *LS* in the context of the new application. *L* first decrypts the symmetric encryption key K_{enc} using the TPM's unbind mechanism. With the decrypted key K_{enc}, the loader is then able to decrypt the second part *encrypted binary (ebin)*, which contains the actual binary *bin*. After a successful signature verification of *bin* and the $nonce_{LS}$ from *LS* using the backend's public key $pk_{\text{sig}}^{\text{BS}}$, the loader *L* measures the new application binary *bin* by computing a *hash value* h_{bin}.

Trusted Application (TA): The trusted application *TA* is a microkernel task [cf. TUD11a] and the final context where the measured binary *bin* is loaded and executed. It includes the loader *L*.

TPM Server (TS): The TPM server *TS* is responsible to handle the hardware TPM by providing a high-level interface to other clients, e.g., the loader *L*. If it is required or otherwise necessary to grant untrusted REE access to *TS*, the TPM server must restrict access to and usage of critical resources stored inside the hardware TPM, e.g., sk_{bind} (cf. Section 7.3.1).

Integrity Server (IS): The integrity server *IS* primarily maintains a *measurement list (ML)*, which contains hash values for each binary once it has been started. For that purpose, it handles requests by the loader *L*, which calculates hashes of new binaries during the loading procedure. To extend the measurements into a PCR of the TPM, the integrity server relies on *TS*.

Main Idea of our Secure Loading Procedure

Our loader concept is inspired by a two-step loading procedure for native microkernel applications, which is based on the loading mechanism provided by L4Re. As described above, instead of directly loading the actual binary, the L4Re *init* process *Ned* first loads a runtime binary into a newly created microkernel task. This runtime binary, in turn, loads the new binary into its own context. Within the context of the new task, the loader can decrypt the binary using the TPM server and device-specific keys, which ensures that the binary can only be executed in the microkernel runtime of a particular system. In addition, strong isolation with respect to other microkernel applications is ensured by the microkernel and its capability-based security features. As a result, the plaintext of every new binary is only available in its own context.

Our concept also allows us to include a verification component, also referred to as *challenger (C)*, in *bin*, which can verify attestation results locally—a benefit over traditional remote attestation as specified by the TCG. After a single remote attestation, the challenger effectively represents the remote verifier on the local system and can verify attestation results on behalf of the verifier as proposed in [Sch12]. As a result, the effects of the TOCTOU problem are significantly reduced. Furthermore, to protect the MEE from being compromised by random binaries generated by an attacker, e.g., after compromising the untrusted rich OS execution in the REE, a signature can be applied before the encryption by the backend system. Finally, to prevent replay attacks, the microkernel runtime generates a nonce, which is included in the request to the backend and specific to a particular binary.

Secure Loading Protocol

Based on the main idea of our concept, a formal protocol representation of our secure loading procedure is depicted in Figure 7.3. First, the rich OS *R* requests a $nonce_{LS}$, ideally generated by the TPM's RNG, from *LS* in step 1. *R* forwards the $nonce_{LS}$ to the backend system in step 2 together with the request for the new binary *bin*, which will be provided in encrypted form, i.e., as *ebin*.

In step 3, the backend system assembles the encryption key K_{enc}, which is used to encrypt the binary *bin*, and a $nonce_{BS}$, which is later used for an *implicit attestation* of our microkernel-based system in the MEE (cf. step 13). Before the encryption with pk_{bind}, the binary and the $nonce_{LS}$ are combined and signed with *BS*'s private signing key $sk^{\text{BS}}_{\text{sig}}$, which creates the signature sig_{BS} in step 3b. Finally, to generate the encrypted binary *ebin*, the backend system *BS* encrypts the key K_{enc} and $nonce_{BS}$ with public binding key pk_{bind} of the embedded system in step 3c, which creates *ebin*[0], and the data structure {*bin*, $nonce_{LS}$ } with K_{enc} in step 3d, which results in *ebin*[1].

In step 4, the encrypted binary *ebin* is then transferred to the rich operating system *R* via network and handed over to the loader client *LC* in step 5 *LC*, in turn, sends *ebin* to the loader service *LS* in step 6, which starts the loader *L* insider the new task *TA* in step 7. In step 8, *L* requests the encrypted binary *ebin* from the *LS*.

Using the TPM service *TS* (step 9), *L* can decrypt the first part of the encrypted binary, i.e., *ebin*[0], which contains the encryption key K_{enc} (and $nonce_{BS}$). To be able to decrypt *ebin*[0], the TPM server loads the wrapped binding key $\{K_{\text{bind}}\}^{P_{\text{MEE}}}_{pk_{\text{SRK}}}$ in step 10 verifying the current platform configuration P'_{MEE}. Using sk_{bind}, *TS* decrypts *ebin*[0] in step 11 resulting in $\{K_{\text{enc}}, nonce'_{BS}\}$, which is returned to *L* in step 12. By returning $nonce'_{BS}$ to *BS* in step 13, which verifies that $nonce_{BS}$ equals $nonce'_{BS}$, the system can prove the trustworthiness of the platform, because loading the wrapped binding key and decrypting the nonce is only possible if the system is still trustworthy. With K_{enc}, *L* can decrypt *ebin*[1] in step 14a, which contains both binary *bin* and its signature sig_{BS}. In step 14b, the loader verifies the code signature sig_{BS} of the binary *bin* using *BS*'s public signing key $pk^{\text{BS}}_{\text{sig}}$. Finally, *L* measures *bin*, extends the hash into the TPM using *IS* in step 14c and runs *bin* in step 14d.

1.	$R \leftrightarrow LS$:	request $nonce_{LS}$	// by TPM's RNG
2.	$R \rightarrow BS$:	$nonce_{LS}$, request for *ebin*	
3.	BS :	assemble *ebin*	
	a)	generate K_{enc} and $nonce_{BS}$	
	b)	$sig_{BS} = sign(sk^{\text{BS}}_{\text{sig}}, \{bin, nonce_{LS}\})$	
	c)	$ebin[0] = encrypt(pk_{\text{bind}}, \{K_{\text{enc}}, nonce_{BS}\})$	// asymmetric
	d)	$ebin[1] = enc(K_{\text{enc}}, \{bin, sig_{BS}\})$	// symmetric
4.	$BS \rightarrow R$:	*ebin*	// untrusted network
5.	$R \rightarrow LC$:	*ebin*	// REE
6.	$LC \rightarrow LS$:	*ebin*	// MEE
7.	LS :	start *L*	
8.	$L \leftrightarrow LS$:	request *ebin*	
9.	$L \rightarrow TS$:	*ebin*[0]	
10.	$TS \leftrightarrow$ TPM :	$[sk_{\text{bind}}] = loadKey(sk_{\text{SRK}}, \{K_{\text{bind}}\}^{P_{\text{MEE}}}_{pk_{\text{SRK}}}, P'_{\text{MEE}})$	// TPM verifies P'_{MEE}
11.	$TS \leftrightarrow$ TPM :	$\{K_{\text{enc}}, nonce'_{BS}\} = decrypt(sk_{\text{bind}}, \{ebin[0]\})$	
12.	$TS \rightarrow L$:	$\{K_{\text{enc}}, nonce'_{BS}\}$	
13.	$L \rightarrow BS$:	$nonce'_{BS}$ (via *R*)	// *BS*: $nonce'_{BS} \stackrel{?}{=} nonce_{BS}$
14.	L :	verify and securely load binary *bin*	
	a)	$\{bin, sig_{BS}\} = dec(K_{\text{enc}}, \{ebin[1]\})$	
	b)	$verify(pk^{\text{BS}}_{\text{sig}}, \{bin, nonce_{LS}\}, sig_{BS})$	
	c)	run measurement protocol with *IS*	
	d)	execute *bin*	

Figure 7.3: Secure Loading Procedure

7.3.3 Measuring a Microkernel Application and Attesting Integrity

Inspired by the IMA approach [Sai04], the calculation of integrity measurements is conducted before a microkernel application loaded and executed. Our integrity measurement concept hashes all microkernel applications executed on the on the system, except for the measurement component, which are measured during authenticated boot.

Measuring the Integrity of Binaries

Figure 7.4 shows the measurement protocol for a binary bin_k, where index k specifies a certain binary. For k=1, the measurement is calculated and extended into a PCR of the TPM as follows:

1. Before the binary bin_1 is started, the loader L calculates an integrity measurement hash as $h_{bin_1} := H(bin_1) = SHA\text{-}1(bin_1)$ over the binary file bin_1, which is suitable for the TPM 1.2. *Note:* For a TPM 2.0, $H()$ could be *SHA-256*, for example, if the PCR banks support *SHA-256*.
2. h_{bin_1} is sent to the integrity server *IS*.
3. *IS* appends the measurement h_{bin_1} to its measurement list *ML* using append.
4. *IS* then uses the TPM server *TS* to aggregate h_{bin_1} into a $PCR[i]$ using extend, where the index i is 10 according to IMA. The new PCR value is securely calculated inside the TPM as $PCR[i] \leftarrow H(PCR[i] \parallel h_{bin_1})$, where $H()$ is $SHA\text{-}1()$ in case of a TPM 1.2.

Similarly, the measurement protocol is also executed for any new binary bin_k with $k > 1 \geq n$. As a consequence, the PCR contains an aggregated value of all measured binaries $bin_1 \ldots bin_n$, which effectively represents the most recent element of a hash chain. Since the measurement protocol is executed before the application is started and the design of TPM prevents software to (re)set PCRs to arbitrary values (except for the debug $PCR[16]$), the measured binary has no influence on its measurement process. In particular, the application cannot change its integrity measurement, which has already been extended into the PCR of the TPM once the application runs. However, if the measurement components itself are compromised, an attacker can extend arbitrary values. Fortunately, this attack effectively prevents the attacker from loading K_{bind} and can be easily detected by examining the chain of integrity measurements, e.g., using attestation.

1. L : $h_{bin_k} := H(bin_k) = SHA\text{-}1(bin_k)$
2. $L \rightarrow IS$: h_{bin_k}
3. IS : append(ML, h_{bin_k})
4. $IS \leftrightarrow TS$: extend$(h_{bin_k}, PCR[i])$

Figure 7.4: TPM-based Measurement Protocol

Attesting Integrity

Since the collection of integrity measurement values for loaded applications alone does not guarantee a trustworthy system, we rely on TPM-based attestation to create proof of the trustworthiness. In a traditional remote attestation as specified by the TCG, the prover's TPM signs PCR values, which are transferred to the remote verifier together with the SML. However, as a result of our secure loading concept, which can ensure that the binary is executed in the microkernel runtime, we can use *implicit attestation* to prove that our base system in the MEE is still trustworthy. By embedding a nonce in the encrypted binary, which can only be decrypted by the target system if the current platform configuration P'_{MEE} matches P_{MEE} specified for K_{bind}, the backend can implicitly verify the system, which is only considered trustworthy if the correct $nonce_{BS}$ is returned to *BS*.

Furthermore, the backend system can also embed a local verifier, also known as challenger *C*, into the application, which then acts as a local representation of the verifier $\mathcal{V}$ and can hence verify the prover's system state directly. For this purpose, a whitelist *WL* with entries for acceptable tasks and their integrity measurement is embedded in the application binary. The corresponding integrity challenge protocol, which is inspired by IMA [Sai04] and adapted to our microkernel-based runtime environment to enable local and remote attestation, is detailed in Figure 7.5.

1. C : generate non-predictable 160-bit $nonce_C$
2. $C \rightarrow IS$: request for a quote $\{targetPCR, nonce_C\}$,
   where *targetPCR* specifies the selection of the quoted PCRs
3. $IS \rightarrow TS$: $\{targetPCR, nonce_C\}$
4. $TS \leftrightarrow$ TPM : load protected $[sk_{\mathsf{aik}}]$ = loadKey(sk_{SRK}, $\{K_{\mathsf{aik}}\}_{pk_{\mathsf{SRK}}}$) into the TPM
5. $TS \leftrightarrow$ TPM : retrieve $Quote = \{pcrData, sig_{\mathsf{aik}}\}$,
   where the TPM calculates $Quote = quote(sk_{\mathsf{aik}}, nonce_C, targetPCR)$
6. $TS \rightarrow IS$: *Quote*
7. $IS \rightarrow C$: *Quote*, *ML*
8. C : verify *Quote*, $nonce_C$, and *ML*
   a) $pk_{\mathsf{aik}} = verifyCert(Cert_{\mathsf{AIK}}, \{Cert_{\mathsf{CA}}\})$
   b) $verify(pk_{\mathsf{aik}}, Quote) \stackrel{?}{=} SHA\text{-}1(\{\text{“}\texttt{QUOT}\text{”}, pcrData, nonce_C\})$
   c) calculate aggregate $A = \biguplus_{k=1}^{n} SHA\text{-}1(ML[k].\text{hash})$
   d) check $A \stackrel{?}{=} pcrData.\text{hash}$ and if $A \in WL$

Figure 7.5: Integrity Challenge and Attestation Protocol

As depicted in Figure 7.5, the integrity challenge and attestation protocol requires the following steps:

1. The challenger *C* generates a non-predictable 160-bit $nonce_C$. This random number prevents replay attackers, where an attacker tries to convince the challenger that the system is still trustworthy by simply returning old or previously signed integrity measurements.
2. The challenger *C* requests the current measurement list from the integrity server *IS*, which will indirectly trigger the generation of a TPM-signed quote of the current PCRs. The request includes the $nonce_C$ and an indication of the PCR numbers (*targetPCR*) to be used.
3. *IS* forwards {*targetPCR*, $nonce_C$ } to the TPM server *TS*.
4. The TPM server loads the (encrypted) attestation identity key $\{K_{\text{aik}}\}_{pk_{\text{SRK}}}$ into the TPM to be able to use the private portion sk_{aik}.
5. *TS* invokes the `TPM_Quote` command specifying the *targetPCR* as well as the $nonce_C$. The TPM, as a result, signs the PCRs involved during boot and the PCR used for protecting the measurement list *ML* with sk_{aik}. The TPM returns the PCR contents, which is part of *pcrData*, as well as the signature sig_{aik} to *TS*.
6. *TS* returns the *Quote* to *IS*.
7. The integrity server transfers the *Quote* and *ML* to the challenger.
8. *C* verifies *Quote*, $nonce_C$, and *ML*
   a) *C* checks if the TPM's pk_{aik} is valid. For this purpose, it traverses the certificate chain until it can validate the public key. In Figure 7.5, the certificate verification is indicated by the function *verifyCert()*.
   b) If the preceding step succeeds, *C* uses pk_{aik} to verify if the signature fits to the signed quote data structure, in particular the PCR contents and the nonce.
   c) In the case of a successful signature verification, the challenger calculates the aggregate *A* based on *ML*. For this purpose, it iterates over the hash values contained in *ML* and freshly calculates $A \leftarrow SHA\text{-}1(A \parallel ML[k].\text{hash})$ for each entry *k* in the measurement list. Please note that the challenger has to aggregate the measurements in the same order as the *IS* calculated and extended them into the TPM during the measurement protocol, since hashing with SHA-1 is not commutative.
   d) At this point, the aggregate *A* should equal the value in *pcrData*.hash returned by the integrity server. If the hash values match and the aggregate is in the whitelist *WL*, the challenger considers the system trustworthy. Otherwise the challenger will not trust the system and might terminate execution in the local case or report back to the backend system.

7.4 Proof of Concept Implementation

As a proof of concept, we have implemented our secure loading and integrity verification concept on the L4 kernel interface implementation Fiasco.OC, which acts as a separation kernel, and L4Re providing the trusted microkernel runtime for native tasks. As hardware platform, we decided to use a *PandaBoard* [Pan10], since it represented modern, widely used, low-cost and mid-range smartphone hardware suitable for our scenario at the time of development. The *PandaBoard* is based a *Texas Instruments (TI) OMAP4*, which features an ARM Cortex-A9-based dual core SoC with 1 GHz and 1 GiB RAM. As shown Figure 7.6, the TPM chip, an Infineon *SLB 9635 TT 1.2*, is connected over the Inter-Integrated Circuit (I2C) bus to our development board.

In the remainder of this section, we describe our proof of concept implementation. In Section 7.4.1, we describe the integration of the TPM into the MEE. In Section 7.4.2, we discuss the IPC Abstraction in L4Re. In Section 7.4.3, the secure loading and verification of binaries within L4Re is discussed, while Section 7.4.4 discusses the access control via capabilities. Finally, Section 7.4.5 describes details about our integrity measurement and attestation protocols.

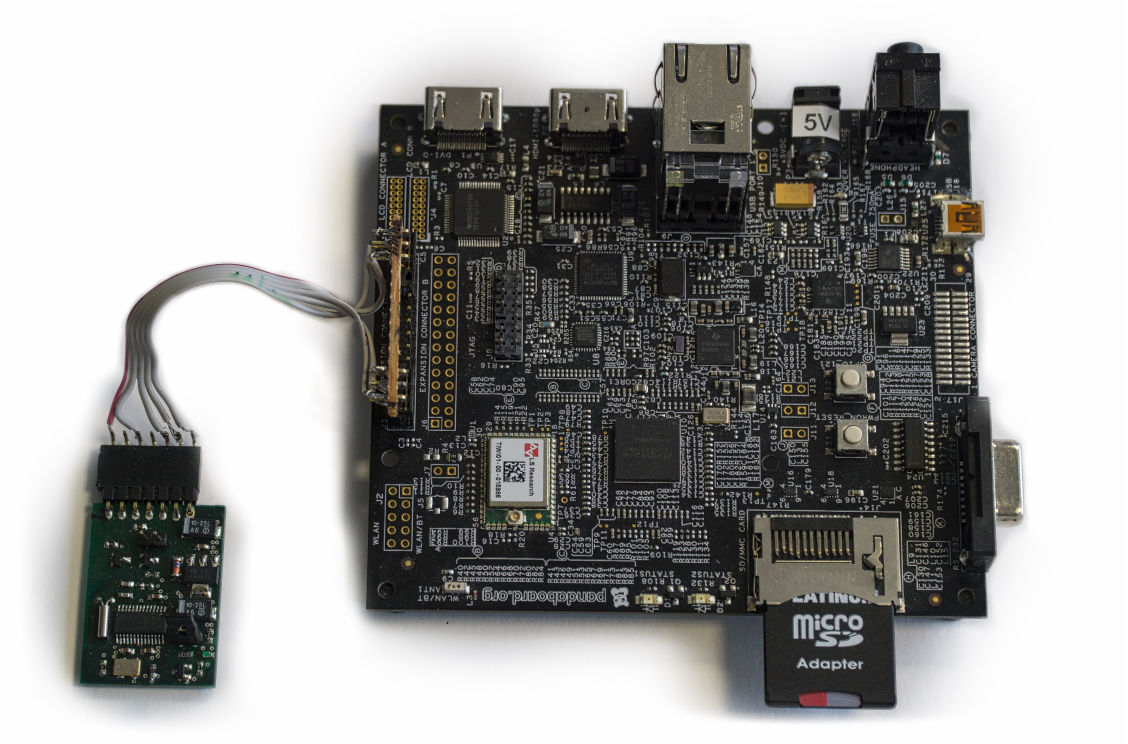

Figure 7.6: Hardware Platform for our Proof of Concept (*PandaBoard* with TPM)

7.4.1 Integration of the Hardware TPM

In this section, we describe the integration of a hardware TPM based on our near-minimum TPM software stack, which is significantly smaller than other libraries like *TrouSerS* [Tro05].

TPM Software Stack

For the TPM server, we implemented a small and simple C library, which still contains all necessary TPM commands for our concept described in Section 7.3.2. This library is designed to be tiny and run standalone, but due to performance reasons, we made one exception. We use an external crypto library for host-side SHA-1 computations instead of our TPM-based SHA-1 implementation since the Infineon SLB 9635 TPM only provides a rather slow software I2C implementation. In case of the boot loader, this is *mbed TLS* [ARM17] and for L4Re we use the *libcrypto* package.

For our TSS, we implemented the following commands: `TPM_Unseal`, `TPM_MakeIdentity`, `TPM_LoadKey2`, `TPM_Quote`, `TPM_Extend`. Those commands, in turn, depend on other commands, e.g., for initializing the TPM and handling OSAP sessions which provides low-level bus encryption. We implemented those commands according to the TPM Main Specification [Tru11]. Whenever possible, command parameters are hard-coded for our specific application scenario. This keeps the function signature as simple as possible and makes our TPM API less error-prone. For instance, our `TPM_MakeIdentity` function only takes pointers for output buffer and size of the resulting key structure. The key parameters for the AIK are directly specified in the function, which reduces errors as wrong key types or encryption and signature schemes cannot be set accidentally.

TPM Integration

To protect the boot chain, we integrated our TPM library in *U-Boot*, which is the boot loader of our prototype board. Further, we implemented the necessary driver to access the TPM inside the boot loader. This implementation is partly based on joint work with Lorenz [Lor12], which describes and implements a TCG-inspired mechanism for secure boot on ARM-based embedded systems. For the low-level connection to the TPM, we ported the minimal necessary parts of the Linux 3.0 driver for OMAP boards and the `tpm_tis` driver, needed for the Infineon TPM chip to *U-Boot*.

In L4Re, we completely reuse the TPM library as well as the low-level I2C device driver of the *U-Boot* implementation. However, we are not able to directly access the memory-mapped I2C device with the physical addresses used in the boot loader code. Fortunately, L4Re provides the *Io* server, which can be used to forward a physical memory region to several user space device servers. Hence, we configured the memory region of the fourth I2C bus where the TPM is connected for *Io*:

```
1 i2c4 => new Device() {
2   .hid = "I2C";
3   new-res Mmio(0x48350000 .. 0x48350fff);     // 4th i2c bus on OMAP4
4 }
```

Listing 7.1: Configuration of the TPM memory region for *Io*

The *Io* Server can also be configured for handling and redistribution of interrupt requests (IRQs). Unfortunately, the Infineon SLB 9635 TPM does not provide interrupt support for asynchronous notifications, but instead requires polling a status registers to determine the completion of a command. As a consequence, it is sufficient to map the corresponding memory region of the I2C bus into user space without mapping any interrupts.

Lastly, our implementation for L4Re comprises of a TPM server *TS* with a dual role. On the one hand, *TS* is a client to the *Io* server handling low level I2C communication. On the other hand, it provides high level TPM functionality as server for other applications. In our system architecture, these software components mainly include the integrity server *IS* and the loader *L* inside the runtime binary *l4re_kernel*. As shown in Figure 7.2, the *IS* uses *TS* to extend integrity measurements into the TPM, whereas *L* relies on *TS* to decrypt the first part of the encrypted binary with the binding key securely stored in the TPM.

The TPM server interface is rather simple and only maps calls to the high-level commands `CreateWrapKey`, `Extend`, `LoadKey2`, `MakeIdentity`, and `Quote`, which have opcodes defined in a header file, which can be included by the clients to easily call those functions through IPC. For TPM commands with small payload, such as `Extend` or `Quote`, the payload can directly be transferred over the IPC *IOStream*, which is described in detail in the following section. However, for key generation commands like `CreateWrapKey` or `MakeIdentity`, we cannot transfer the payload through IPC due to size limitations. Instead, we use a shared memory page and write the encrypted key structure on a previously created shared memory page in the TPM server's context.

7.4.2 IPC Abstraction with L4Re *IOStreams*

For IPC between applications, L4Re provides a C++ abstraction framework based on *IOStreams*. We utilize this framework for communications between our components, e.g., to directly transfer TPM commands with small payloads to the TPM server *TS*. To establish connections between applications, the init process *Ned* creates new communication channels, which have to be specified in its *Lua*-based configuration script. Since *Ned* also sets up the application's initial capabilities, it can grant access to those channels using capabilities. Consequently, servers and clients have to request a capability to particular channels, which creates a reference that allows access to a channel.

However, since this mechanism is usually only available during the initial startup of the system, we have to provide other means to establish the communication between the initial L4Re services and the external binary, which is started at a later time. In our proof of concept implementation, we use a global namespace and provide the capability to access this namespace via *Ned*. The server needs to registers its server object to that namespace instead of the communication channel created by *Ned*'s *Lua* script. Apart from that, the communication between client and server over *IOStream* works the same as if the connections had been accomplished with the *Lua*-based mechanism. Additional details of this approach are described in the following two sections.

7.4.3 Implementation of our Secure Loading Procedure

In our proof-of-concept implementation, we extended the L4Re init process *Ned* with our loader server *LS*, which is realized as a thread inside *Ned*. While the original version of *Ned* only provides the ability to load local binaries, which need to be known at the startup and must be stored in the ROM file system, our *LS*-extended version of *Ned* enables a dynamic loading of remote binaries. For the sake of simplicity, the corresponding loader client *LC* is realized as an L4Re application. However, *LC* can also be wrapped in a very simple *L4Linux* kernel module as depicted in Figure 7.7. In our implementation, the loader client copies the binary from the global ROM file system to a temporary L4Re *Dataspace*, which is referred to as `binfile` and shared between *LS* and *LC*. Access to that shared data space (and IPC between *LC* and *Ned*) is configured by *Ned* using a global namespace, which is discussed in detail in the following Section 7.4.4. In addition, an IPC channel for notifications between *LC* and *LS* is set up during boot using *Ned*'s standard *Lua* mechanism.

When *LC* signals that the binary has been copied to the shared data space, *LS* starts *l4re_kernel* in the new task *TA* by executing a second *Lua* file. This internal default *Lua* file configures the command line arguments and capabilities of the binary, especially access to the global namespace including the shared data space. Before the *Lua* interpreter for the second *Lua* file is started, the capability of the data space is registered to the global namespace, which *TA* is granted access to. Finally, the loader *L*, which is part of *l4re_kernel*, decrypts, measures, and starts the binary in its new task context. Here, one important security issue is the fact that the binary must not be measured and loaded directly from the shared data space, because the untrusted loader client *LC* has immediate write access. We discuss this aspect in our security evaluation presented in Section 7.5.2.

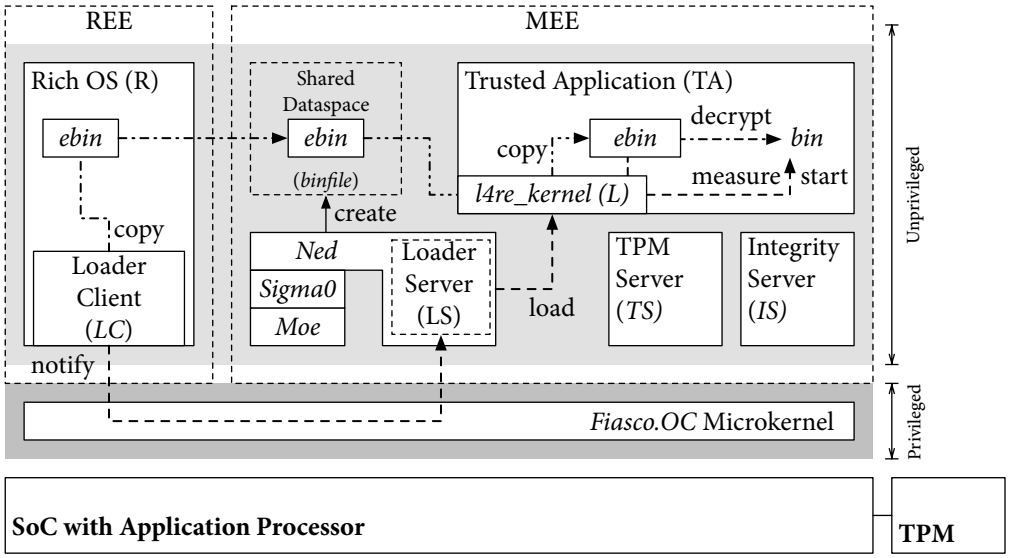

Figure 7.7: Implementation of Secure Loading of an External Binary into the MEE

7.4.4 Capability Transfer and Access Control

To transfer the capabilities required for IPC communication and grant access to shared memory, we utilize two global L4Re *Namespaces*, `ldrs` and `ims`, as depicted in Figure 7.8. We initially create these namespaces in the root task *Moe*, where they can be registered to the global L4Re *Environment*.

As the loader client *LC* is part of the untrusted REE, our main objective is to control access to resources that are part of our integrity and attestation framework in the MEE. As a result, we provide the `ldrs` namespace for communication between *LS* and *LC* (and the `ims` namespace for inter-component communication inside the MEE). Access to both namespaces is configured by *Ned*, which grants the loader client `read-only (ro)` access to the `ldrs` namespace and, thereby prevents *LC*, which prevents the untrusted runtime in the REE, from creating new objects, e.g., additional unspecified communication channels with the microkernel runtime, inside the `ldrs` namespace. However, it is important to note that Ned can register objects inside the `ldrs` namespace as writable. Thus, the loader server *LS* is able to registers the `binfile` object with `read-write (rw)` permissions and the `ldr-obj` for IPC communication to the `ldrs` namespace.

Furthermore, *Ned* grants the integrity server *IS* read-write (`rw`) access to the `ims` namespace, since *IS* needs to be able to register the `im-obj` for the loader. The loader *L*, which is part of *l4re_kernel*, uses `im-obj` to send IPC calls to perform the measurement and attestation protocols. In addition, *Ned* registers the `binfile` object to that namespace, too, and grants *L* read-only (`ro`) access, so that *L* can access it for decryption and loading.

As a result, *Ned* basically acts as firewall between the untrusted REE and microkernel runtime in the MEE. Please note that since the communication between *TS* and *IS* uses local IPC channels configured by *Ned*, it does not require shared memory, but relies directly on the microkernel.

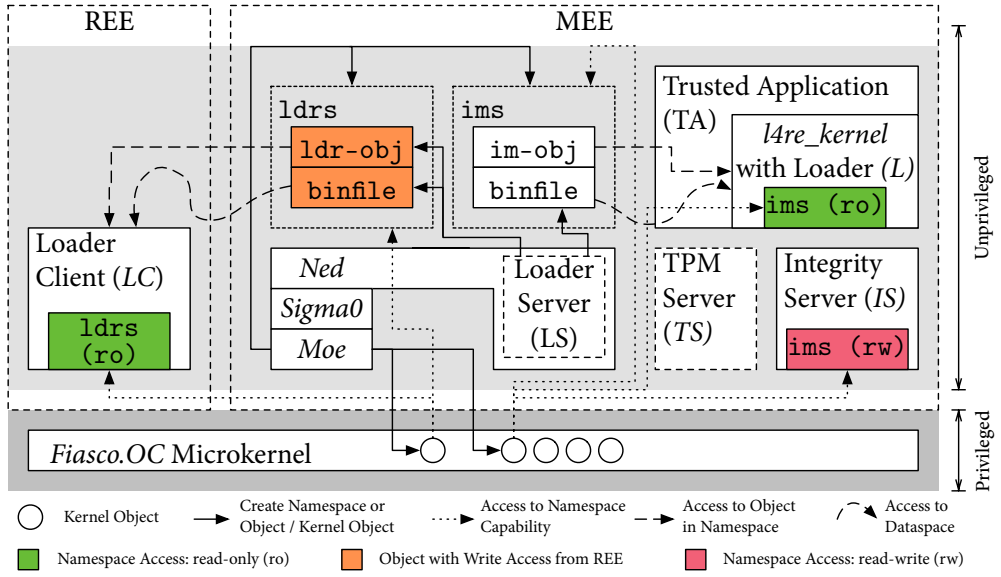

Figure 7.8: Access to Global Namespaces and Capabilities to Server Objects for IPC and Data Exchange

7.4.5 Integrity and Attestation

We implemented *IS* as L4Re server, which provides a public interface for the measurement and the attestation protocol. At startup, *IS* registers the `im-obj` in the global `ims` namespace to export two functions, one for the measurement and one for the attestation protocol, over the C++ *IOStream* abstraction for IPC. The client side of the measurement protocol is handled in *L* inside *l4re_kernel*. We have a simple `Tpm` class inside of *IS*, which provides the client-side implementation of the TPM server interface described above. This class mainly hides the *IOStream* and shared memory communication for a simpler implementation of the attestation and measurement protocol.

Integrity Protocol

We now describe some details about the implementation of the integrity protocol (Figure 7.4). The loader *L* inside of the *l4re_kernel* registers the `ims` namespace and queries for the `im-obj` to establish the connection to the integrity server. *L* copies the received binary from the shared data space into a new *local* data space, decrypts, verifies, and computes a SHA-1 hash of the binary. Then, *L* calls the integrity measurement routine via IPC and sends the SHA-1 hash and filename of the measured binary via *IOStream* to *IS*. Our integrity server, in turn, uses the `Tpm` class method `extend()` to send the SHA-1 hash of the binary via *TS* to the TPM. *IS* also creates a new entry for the measurement list containing the filename and hash value, which is appended to *ML*.

Attestation Protocol

Compared to the integrity measurement protocol, the implementation of the attestation protocol described in Figure 7.5 is more complex. From the client/challenger perspective, the main protocol is mostly hidden and initiated in only one call over IPC, which includes the $nonce_{LS}$ for freshness. The results of the attestation, in turn, is delivered in a single data structure containing the AIK public key as well as the raw signature returned by `TPM_Quote` and the measurement list *ML*. The TPM server returns pk_{aik} in the `TPM_Key12` format. The `Tpm` class abstraction transparently converts the `TPM_Key12` data to the raw 2048-bit RSA key for the signature verification. In our implementation, we also generate a new AIK key pair with `TPM_MakeIdentity` when *IS* requests the pk_{aik} on behalf of the challenger *C* due to the lack of persistent storage in our MEE. At the end of the protocol, when the IPC returns, *C* needs to do the verification item step 8 of the protocol in Figure 7.5. The necessary information is read and parsed from the shared memory. For this purpose we have implemented the `tpm_extend()` function in software which computes the aggregate *A* over all hash values contained in *ML*. To verify the signature from `TPM_Quote`, which is an RSA encrypted hash over a `TPM_QUOTE_INFO` data structure, *C* needs to recreate this structure with its locally computed values for the aggregate and $nonce_{LS}$ and also compute a fresh hash value over this structure. If the comparison of this hash and the decrypted hash match and *WL* contains the hash of the aggregate *A*, the system is trustworthy and the program execution can continue.

7.5 Evaluation

In this section, we evaluate our concept and prototype implementation regarding TCB, security, and performance. As described in Section 7.4, our hardware platform consists of a *PandaBoard*, which we have equipped with an Infineon SLB 9635 TPM 1.2 featuring an I2C interface.

7.5.1 Evaluation of the Reduced Trusted Computing Base

First, we discuss the TCB and show that our approach has a much smaller and simpler TCB compared to other approaches, which often rely on a very complex rich OS kernel like Linux and a large software stack for measuring the integrity of software components.

Microkernel Runtime including Integrity Measurement Components

The trusted microkernel runtime in our MEE allows individual consideration of the TCB for each user level task. For instance, the TPM server uses the *Io* server for mapping the right memory to the TPM chip. Hence, the *Io* server is part of the TCB for the TPM server. Further, *Sigma0* as root pager, *Moe* as the root task, and *Ned* are also part of the TCB for every other task, which results in the following per-application TCB:

***Sigma0*:**	Fiasco.OC, *Sigma0*
***Moe*:**	Fiasco.OC, Sigma0, *Moe*
***L4Re runtime (l4re)*:**	Fiasco.OC, Sigma0, Moe, *l4re*
***Ned*:**	Fiasco.OC, Sigma0, Moe, l4re, *Ned*
***Io*:**	Fiasco.OC, Sigma0, Moe, l4re, Ned, *Io*
***TS*:**	Fiasco.OC, Sigma0, Moe, l4re, Ned, IO, *TS*
***IS*:**	Fiasco.OC, Sigma0, Moe, l4re, Ned, IO, TS, *IS*

All these applications together form the TCB of our secure loading procedure for external binaries. As a consequence, our implementation depends on the trustworthiness of these components, which are therefore measured by the authenticated boot mechanism in *U-Boot* establishing the current P_{MEE}. To estimate the size of the TCB, we used the tool *cloc* [Dan16] to count the lines of source code for all these components in the L4Re source tree and the libraries on which they depend. For the rich OS kernel as well as the microkernel, we striped out the architecture dependent code for other architectures and only count the ARM-specific code to the TCB. As a result, we counted around 9.5 million lines of code (*MLOC*) for *L4Linux*, the rich OS in our implementation, which includes the IMA code. An Android version (*L4Android*) with additional vendor drivers for modern smartphones would even have more lines of code. In comparison to the monolithic *L4Linux* kernel, we measured about 95 thousand lines of code (*KLOC*) for Fiasco.OC. This also includes the in-kernel debugger *jdb* with about 20 KLOC. For the microkernel runtime including all libraries needed for our integrity measurement and loader service we measured ≈750 KLOC.

Code Size of our Addition to L4Re

In this section, we calculate the additional code that we introduce to L4Re through our measurement and attestation setup. The relevant L4Re components comprise of *Moe*, the loader server *LS*, the loader *L*, the TPM server *TS*, and the integrity server *IS*. The results are summarized in Table 7.1.

To register the `ims` and `ldrs` namespaces in *Moe*'s root namespace, we only had to add 3 lines for each namespace, i.e., 6 lines in total. The loader server *LS* adds 215 lines of code (LOC) to the original init task *Ned* of L4Re. Additionally, to integrate the measurement code for *L* into *l4re_kernel*, we added 206 LOC to the original `loader.cc`. Our integrity server *IS* is very small with 703 lines of code, which includes our implementation of the integrity measurement and challenge protocol.

Lastly, the TPM server has 3,308 lines of code. However, since the TPM server shares the TPM library and the I2C driver with the *U-Boot* code, we can subtract those components, because they already counted towards the TCB. As a result, the addition to the TCB is merely 251 LOC. Further, compared to a full TSS implementation like *TrouSerS* [Tro05] with about 75 KLOC in version 0.3.14, our TPM library significantly reduces the TCB by approximately 70 KLOC. As a result, we only added 1,381 LOC in total to L4Re as shown in Section 7.5.3.

	Moe	*Ned/LS*	*l4re_kernel/L*	*IS*	*TS*	Total
LOC	6	215	206	703	251	1,381

	TrouSerS	*TS* (w/o libs)	TPM library	TPM driver	I2C driver	Total
LOC	≈75,000	251	1,908	398	751	3,308

Table 7.1: Code Additions to L4Re with a Comparison to *TrouSerS* [Tro05]

Total TCB Code Size

If we compare our solution, i.e., Fiasco.OC plus L4Re including our additions, to other approaches like *L4Linux*, which relies on a monolithic kernel (Linux), the difference in terms of code size is approximately 845 KLOC versus 9.5 MLOC or around one order of magnitude as shown in Table 7.2. Since the boot loader code is required for both implementations, we can ignore it in our comparison. For our approximation, we subtracted the architecture-dependent code for other architectures except `arm` from the Linux kernel `arch` directory and from the Fiasco.OC `kern` directory.

	L4Linux	L4Re	Fiasco.OC	Total
KLOC	≈9,500	≈750	≈95	≈845

Table 7.2: Difference in Code Size between *L4Linux* (REE) and Fiasco.OC with L4Re (MEE)

7.5.2 Informal Security Analysis

In this section, we analyze the security of our integrity measurement and attestation concept for microkernel-based systems, which enables secure loading of remote binaries. In Section 7.5.2.1, we first discuss our attestation mechanisms. After that, we describe how our attestation and loading concept can prevent various attacks scenarios, e.g., arbitrary code execution, in Section 7.5.2.2.

7.5.2.1 Security Discussion of the Attestation Mechanisms

To realize our secure loading concept, we rely on the fact that the microkernel runtime of the prover is trustworthy before new remote binaries are loaded and executed. At the same time, measuring the integrity of the binaries and extending their integrity measurement value into a PCR of the TPM does not guarantee the system's trustworthiness after a binary has been executed. Hence, we also require proof for the system's trustworthiness after the binaries have been loaded.

As a result, our integrity measurement and secure loading concept for microkernel-based systems incorporates two attestation mechanisms: an implicit attestation towards the backend system before the trusted remote binary is loaded and a local attestation in combination with a challenger at a later point in time. We discuss the security of those attestation mechanisms in the following paragraphs.

Implicit Attestation towards the Backend System

For our secure loading concept, we have defined a wrapped binding key $\{K_{\text{bind}}\}^{P_{\text{MEE}}}_{pk_{\text{SRK}}}$, which is encrypted with the SRK and cryptographically bound to the trusted platform configuration P_{MEE}, which represents the load-time integrity measurements of the software components in the MEE. Since the microkernel runtime needs to load the wrapped binding key to be able to decrypt the $nonce_{BS}$ and the encryption key K_{enc}, which is subsequently used to decrypt remote binary, the integrity measurements of the MEE, in particular the components of our integrity measurement framework, are implicitly verified. Only if the verification of the current platform configuration P'_{MEE} is successful, i.e., matches the specified configuration P_{MEE}, the system is able to decrypt the $nonce_{BS}$ generated by *BS* and return it, which implicitly communicates the current system state to the backend system. If the nonce returned by the prover is not equal to the nonce initially generated by *BS*, the backend system has to assume that the components in the MEE have been modified.

Local Attestation in Combination with a Challenger

Although the attestation based on a local challenger is basically identical to a traditional remote attestation as specified by the TCG, one major advantage is the reduction of the time gap between the generation of the attestation result and its evaluation by the verifier. As described in [Sch12], this approach reduces the attack surface for an adversary to exploit the time gap and compromise the system directly after an attestation. In combination with a microkernel, which has a very small TCB and strictly separates the challenger from the rest of the system, this mechanism also allows for local integrity verifications that even work offline and enable scenarios, e.g., secure offline payments.

7.5.2.2 Attack Prevention and Security of the Secure Loading Mechanism

In this section, we discuss three attacks scenarios: arbitrary code execution in the MEE, illicit execution of the trusted remote binary in the REE, and the compromise of the integrity services and communications. Specifically, we show how our attestation and secure loading mechanism prevents those attacks.

Arbitrary Code Execution

For the first attacker scenario, we define two potential attacks, which are based on our attacker model specified in Section 4.2 and common for integrity verification and remote attestation scenarios: First, a remote attacker might try to run arbitrary code in the MEE with the main objective to compromise the system without being detected by the integrity verification and attestation mechanism.

In the second potential attack, the adversary tries to execute a binary in the MEE and allows for a detection at a later time, e.g., through attestation. That means in the second attack the adversary accepts that a compromise results in a detection, but aims to exploit the time gap between compromise and detection in order to fulfill its attack objective.

Arbitrary Code Execution in the Microkernel Runtime without Detection

To prevent an attacker from running arbitrary code without detection, we have to make sure that our measurement code is called before the binary is executed and that the correct binary is measured. For that purpose, the measurement code is implemented as part of the *l4re_kernel*, which is always loaded before the actual binary. To guarantee that the correct binary is measured, it is essential to take care of access rights for the shared data spaces, i.e., `ldrs` and `ims`.

Further, it is necessary to copy the binary to a private context inside the microkernel runtime. Usually, one would might give the loader client direct access to the data space shared between *Ned* and *l4re_kernel* for transferring the binary. However, the loader client requires write access to that data space to place the binary into this memory location. This write access is permanently granted to *LC*. In this case, the problem arises that the signature check and measuring of the binary is not atomic. If the binary was loaded directly out of that shared data space, the attacker could wait for the right moment where the signature verification and integrity measurement has finished successfully and then replace the binary with arbitrary code.

On the other hand, if the binary is copied to another memory region not accessible by *LC* before it is measured and the code signature of *BS* is verified, the process appears to be atomic for the REE, in particular the loader client *LC*. In our concept and proof-of-concept implementation, that is accomplished by a decryption process, which stores the plain text to a private buffer in the context of the trusted application *TA*. This memory region is only accessible by *TA* and separated by the isolation mechanism implemented by the microkernel.

Arbitrary Code Execution in the Microkernel Runtime regardless of Detection

In contrast to the previous attack, where a challenger or remote verifier can later detect the untrusted system state, which includes the measurement of the compromised binary, the attack is defined successful if a binary controlled by the adversary is executed at all.

In our secure loading concept, we prevent this potential attack by signing the code with the private key $sk_{\text{sig}}^{\text{BS}}$ of the backend system and verifying the resulting signature sig_{BS}. The signature sig_{BS} is checked before execution and only authentic code from a trusted backend system is executed.

However, to verify the authenticity of the code, we rely on the authenticity and integrity of the backend public key $pk_{\text{sig}}^{\text{BS}}$. In our concept, we therefore need the ability to verify the certificate of the backend system, which is ensured through pre-provisioned certificates as described in Section 7.3.1.

Execution of the Trusted Remote Binary in the REE

To prevent an attacker from executing a trusted remote binary in the untrusted rich OS, which is much more likely to be compromised, the binary is encrypted with an ephemeral encryption key K_{enc}. Since the private key sk_{bind}, which is needed for a decryption of the encryption key K_{enc}, is only available from inside the microkernel runtime environment, linked to a particular TPM, and bound to a trusted platform configuration P_{MEE}, it is not possible to decrypt the binary without the TPM. Additionally, the wrapped binding key $\{K_{\text{bind}}\}_{pk_{\text{SRK}}}^{P_{\text{MEE}}}$ is stored in a memory region of the microkernel runtime and access to the TPM is restricted to the microkernel runtime only. As a result, an untrusted rich OS such as *L4Linux* is unable to use both the encryption key and the wrapped key. If the encrypted key was accessible from inside *L4Linux*, for instance through the ROM file system, which is part of every L4 task started by *Ned*, the untrusted rich OS could load the private key into the TPM assuming access to the TPM was available to the rich OS. In our prototype, we therefore restrict the TPM access and only grant access to the microkernel runtime.

Compromise of the Integrity Services and Communications

Communications between our user space components are fire-walled by *Moe* and *Ned*, which includes the loader server *LS*, using separate namespaces. Hence, the main interfaces where an attacker can compromise the microkernel runtime are the ones exported by those components, especially *LS*. These interfaces, however, are strictly controlled and protected by the capability system of the microkernel described in Section 7.4.4. Furthermore, the microkernel protects the boundaries of the shared data space `ldrs`, which is allocated and configured by *LS*.

As described above, the execution of untrusted code, which might compromise our integrity measurement and attestation framework in the MEE, is successfully prevented, e.g., using data encryption and code signing. Similarly, attacks on the communication between the backend and the microkernel runtime are prevented by the cryptographic mechanisms of our secure loading procedure, which ensure confidentiality, authenticity, and integrity of the remote binary *bin*.

7.5.3 Performance Evaluation

To assess the performance of our integrity measurement and secure loading concept, we evaluated our implementation on the *OMAP4*-based *PandaBoard* for various common binary sizes ranging from 200 kB up to 9 MB. The results are depicted in Figure 7.9.

On the *PandaBoard*, we measured a 13.7 MB/s throughput for *AES-256* and around 595 verify operations per second for *RSA-2048* (results of the built-in *OpenSSL* [Ope06] speed benchmark). These timings are labeled *decrypt* for AES and *verify* for RSA, which appear as a single line, because the verification takes almost no time, is independent of the binary size and, hence, can be neglected.

Through our evaluation, we ascertained that the `TPM_Unbind` command used for the decryption of the ephemeral encryption key K_{enc} takes up a constant, but relatively large amount of time. On average, we measured round 700 ms for this command to complete on our implementation as indicated by *unbind* in Figure 7.9. The actual calculation of the integrity measurement, which includes the SHA-1 computation and the extension in a PCR of the TPM, is represented by the line denoted as loader and takes less then 300 ms for binaries up to 1 MB. The plot also shows that the duration of the measurement protocol increases linear with the binary size, which is mainly caused by the SHA-1 calculation, as `TPM_Extend` and the measurement list maintenance operations are independent of the binary size. The external loading of the binary is represented by the line denoted as loader server, which only has a very limited effect on the overall loading time.

In total, we can see a linear increase based on the binary size and that the TPM is a potential bottleneck, especially for small binaries. The main reason is probably that our TPM driver only uses polling, which is not optimal because of we have to use rather conservative timing settings. In addition, TPMs are not designed to be high-performance cryptographic accelerators, but rather optimize cost while still providing a secure execution environment for cryptographic operations.

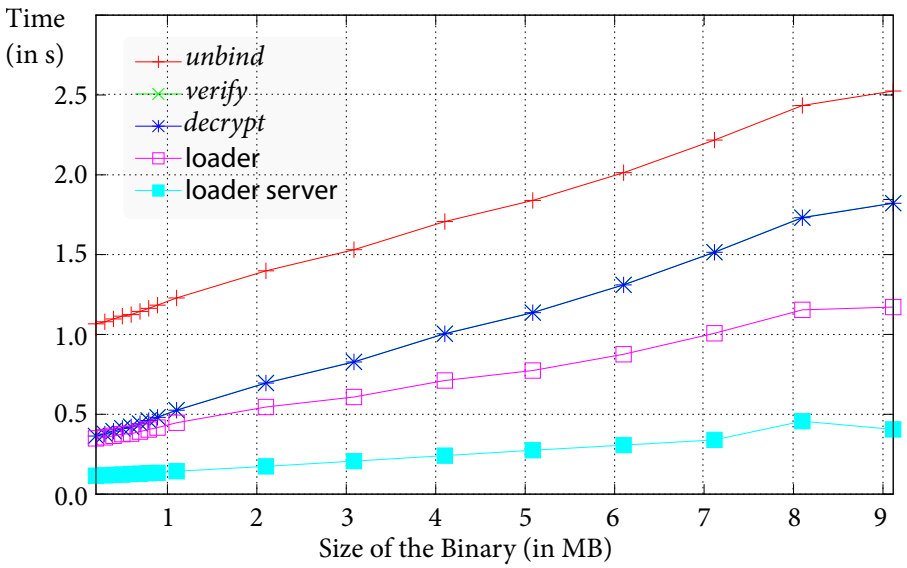

Figure 7.9: Loader Performance Results for Binaries with Different Size [Wei14]

7.6 Summary

In contrast to most extensible firmware implementations or popular rich operating systems, microkernel-based embedded systems are often statically configured to ensure safety and security. Consequently, dynamic loading of remote binaries might be desirable, but is usually not possible, because remote binaries can potentially compromise the integrity and trustworthiness of the system, which is often not acceptable for safety- and security-critical application scenarios and use cases. That is why we proposed a TPM-based integrity verification and secure loading mechanism for microkernel-based systems, which enables dynamic loading and isolated execution of remote binaries in a MEE based on an implicit remote attestation and a local integrity challenge protocol. The implicit attestation, which does not involve digital signatures as required by traditional remote attestation as specified by the TCG, enables the remote verifier to quickly evaluate the platform's trustworthiness during the secure loading procedure and, hence, requires a network connection. After the deployment of the binary, the local integrity challenger can effectively represent the verifier on the prover's system and, although relying on traditional remote attestation, enable a timelier local integrity verification.

For our remote attestation and integrity challenge protocol, we adopted the main ideas of IMA to a microkernel-based system architecture, which strictly isolates the integrity measurement code. As a result, our integrity measurement components can securely collect and store the integrity values of the binaries inside the TPM before the loader executes those binaries in the new context. Consequently, by implementing the integrity measurement and secure loading components as isolated microkernel tasks, our approach separates those components from the rest of the system, significantly reduces the TCB, and, hence, protects them against various attacks, e.g., via the rich OS. In the evaluation of our concept and implementation, we explicitly showed that common attacks, which compromise systems by executing malicious remote binaries, can be prevented or at least detected by verifying and evaluating the attestation response.

As a result, our protocols address the relevant aspects of Scenario 2 (Secure Loading), which focuses on the secure deployment and loading of remote binaries in a microkernel-based execution environment. By utilizing the TPM in our microkernel-based system architecture and adopting the main ideas of IMA, we can provide a way for remote parties, such as banks, to securely deploy their own security-sensitive application in an isolated execution environment, e.g., on their customers' cell phone. Using remote attestation, we also enable the remote party to verify that system, which is a key aspect of the scenario, because the remote party relies on the trustworthiness of the system or at least the microkernel-based OS. If there is no way to prove the system's trustworthiness, most use cases in the scenario, such as secure offline banking, are not be possible, since the bank is likely to require strong isolation of its application, which handles the offline transactions, from the rest of the system.

However, as we mainly focused on a secure loading mechanism for remote binaries in this chapter, we have yet to explore a suitable method to update and recovery outdated or compromised tasks. Since microkernel-based systems are designed to enable recovery after failure or a successful attack if the base system (in particular the microkernel) is still intact, we will focus on update/recovery aspects in the following chapter. More precisely, we will discuss an implicit remote attestation mechanism for selected microkernel tasks with their own cryptographic contexts inside an HSM. This cryptographic context not only stores integrity measurements, but also events collected by an anomaly detection component, which can be equally used in a lightweight remote attestation that enables secure code updates and recovery for microkernel-based systems. Hence, we will focus on a method to combine implicit remote attestation with a mechanism to update and recovery outdated or compromised tasks in the following Chapter 8.

8

Implicit Attestation of Microkernel Tasks for a Lightweight Update and Recovery

While the previous chapter discussed the integrity verification of new microkernel applications based on traditional remote attestation to enable secure binary loading, we focus on a lightweight attestation of microkernel tasks with their own cryptographic context inside the HSM in this chapter. In contrast to an HSM with only one context, such as a TPM 1.2, which, for example, combines integrity measurements into global PCRs, an HSM with multiple cryptographic contexts enables native virtualization support through separate key hierarchies and isolated state information for individual software tasks or VMs. However, since traditional remote attestation subsequently becomes even more inefficient with multiple cryptographic contexts (e.g., because of the use of digital signatures), we propose a lightweight attestation mechanism, which supplements and augments our secure loading protocol with a code update protocol for Scenario 3 (Secure Update and Recovery), which accounts for a multi-context HSM and enables recovery of compromised microkernel tasks.

The rest of the chapter is structured as follows. In Section 8.1, we motivate the attestation scenario. Section 8.2 describes the system design for the attestation and code update presented in Section 8.3. In Section 8.4, we analyze the security of our protocols. Finally, Section 8.5 provides a summary.

Please note that the remote attestation mechanism has already been presented at the *Information Security Conference (ISC)* in 2013 and is published in its peer-reviewed proceedings [Wag15].

8.1 Implicit Remote Attestation of Multiple Cryptographic Contexts

As described in the previous chapters, microkernel-based systems provide separation mechanisms to isolate individual tasks from the rest of the system to increase and ensure safety and security. The system enforces this strict separation by partitioning resources, e.g., CPU time or physical memory, and by virtualizing address spaces and devices. In addition, a microkernel such as *L4*/Fiasco.OC [Lie96; TUD11b] is very small in terms of code size and less complex compared to monolithic kernels, hence considered more trustworthy. However, a strong separation of potentially complex software components by a trusted microkernel does not necessarily imply the trustworthiness of the isolated tasks, which is a desirable property in most security-critical systems.

As we discussed in Chapter 2, a widely used approach to verify the trustworthiness of software components takes advantage of an HSM, such as a TPM. A TPM provides a cryptographic context and mechanisms to securely store integrity measurements, create (a)symmetric keys, and perform certain cryptographic operations, such as encryption. For a remote attestation as specified by the TCG, load-time integrity measurements are signed with a private key inside the TPM and sent to a remote verifier together with a SML in order to prove the integrity of a system. However, since those digital signatures are based on asymmetric cryptography, more precisely RSA (with at least 2048-bit keys), they are quite large and rather expensive[1]. Even though the TPM includes dedicated cryptographic engines for signature calculation, the TPM was never intended to be and, in general, does not act as a cryptographic accelerator. In addition, the TPM was also not designed to handle run-time integrity values, such as events or behavior scores generated by an anomaly detection, which thus cannot be used for a remote attestation.

Furthermore, TPMs do not support virtualization natively, since they generally only provide one cryptographic context for system-wide load-time integrity measurements and keys. That is why most existing concepts rely on the virtual machine monitor or hypervisor to virtualize the TPM [Ber06; Eng08; Stu08]. However, as a consequence of the implementation in software, cryptographic secrets, e.g., keys, or integrity measurement are not always handled inside the TPM. As a result, recent efforts explored and showed the feasibility to realize and manage multiple TPM contexts in hardware [Fel11]. With multiple individual cryptographic contexts, a TPM-based HSM can provide each task with its own security context, which can be used to securely store, for example, keys and integrity measurements on a per-task basis. Unfortunately, as a consequence of the isolated contexts, the number of digital signatures (and SMLs) in a remote attestation as specified by the TCG increases with the number of contexts (i.e., tasks), which makes classical attestation even more expensive and also inefficient, especially on resource-constrained devices.

1 That is because of the exponentiation operations used in RSA's encryption and compared to symmetric cryptography.

To overcome these challenges, we first propose an extension to our microkernel-based system architecture with an integrity verification and anomaly detection component that is enhanced with a multi-context HSM. Within the HSM, the load-time integrity values and events collected by an anomaly detection during run-time (cf. Section 3.1) are stored in distinct contexts, so that task-specific keys, for instance, can be cryptographically bound to these values. As our main contribution, we propose and formally verify a lightweight attestation mechanism, which mainly relies on symmetric cryptography and, thus, is able to efficiently verify multiple tasks in a microkernel-based system. Additionally, our attestation protocol is designed to enable secure code updates based on the integrity of existing security-critical tasks, which extends and complements our secure loading protocol while eliminating the need for digital signatures. For our secure code update protocol, we will show in the following sections how an embedded device with safety- and security-critical tasks, such as an airplane, can create verifiable proof for the integrity of a certain set of microkernel tasks, which enables access to trusted resources, such as emails, confidential documents, or code updates for business applications.

According to our Scenario 3 (Secure Update and Recovery) described in Section 4.1.3, the prover $\mathcal{P}$, an airplane, runs a microkernel-based system executing tasks with different criticality. Those tasks include security-critical microkernel tasks such as (virtualized) device drivers or applications like a secure email or VPN client. Simultaneously, other tasks are not necessarily critical and include, for example, a virtualized rich OS, which only provides regular services and uncritical applications, e.g., for entertainment. Since the security-critical tasks are required to access the resources located on a trusted backend system, e.g., in the airline's network, the integrity of those tasks as well as the underlying system should be cryptographically proven to the verifier. As an attacker might be able to compromise some of the tasks, e.g., by fuzzing the interfaces or exploiting other unknown vulnerabilities of complex tasks, such as the entertainment applications, the attestation protocol enables the verifier to specify which tasks are—in his view—critical and relevant for the trustworthiness of the system, hence enabling recovery of other tasks.

Based on the result of our lightweight remote attestation, the verifier can thus grant access to trusted resources and provide code updates, which are cryptographically linked to a set of critical microkernel tasks. Since our code update protocol is implemented in the MEE, which can limit access to the multi-context HSM, and uses HSM-specific random numbers, we can ensure that the code update is only executed inside the MEE of a particular system based on the results of the previous chapter. To realize this, we can use the integrity key, which we utilize for our implicit remote attestation, to encrypt the code update for the MEE and a particular HSM on a specific device. Finally, since our secure code update protocol implicitly sets up a new integrity key in the cryptographic context of the updated/recovered task, the verifier can directly include that task in future attestations.

8.2 Microkernel-based Architecture with a Multi-Context HSM

In security-critical systems such as described in the scenario, microkernel-based virtualization provides the necessary means to safely execute multiple tasks with different levels of criticality on the same hardware. Nevertheless, most critical systems require hardware-based security mechanisms, since they need to securely store cryptographic secrets, such as private keys, or integrity measurements. That is why those systems are often equipped with an HSM, which acts as a hardware-based security anchor and usually provides an internal context for the system-specific cryptographic information. However, most HSMs with a system-wide cryptographic context, such as a TPM, are not designed to separate and isolate security-sensitive information of individual tasks.

For this purpose, we define a microkernel-based system architecture enhanced with an HSM that supports virtualization by providing multiple task-specific cryptographic contexts, which can be seen as an intermediate design towards the use of a TPM 2.0 as used in the following Chapter 9. As depicted in Figure 8.1, which shows the system architecture that we implemented using the *L4*-based *PikeOS* [SYS91], safety- and security-critical processes are realized as *native tasks*, whereas other non-critical components might be *POSIX (Portable Operating System Interface) tasks* or a *virtualized Linux instance*. All these individual tasks are isolated by the separation mechanisms of the kernel. However, the tasks are still able to communicate with other tasks in a controlled way, i.e., via kernel-based IPC and individually distinct shared memory pages. The HSM proxy, for instance, receives commands via IPC and shared memory. It identifies the origin of the command, forwards it to the HSM, and receives the result, which in turn is communicated back to the respective source. Since all channels are isolated by the kernel, other tasks cannot read command or result messages.

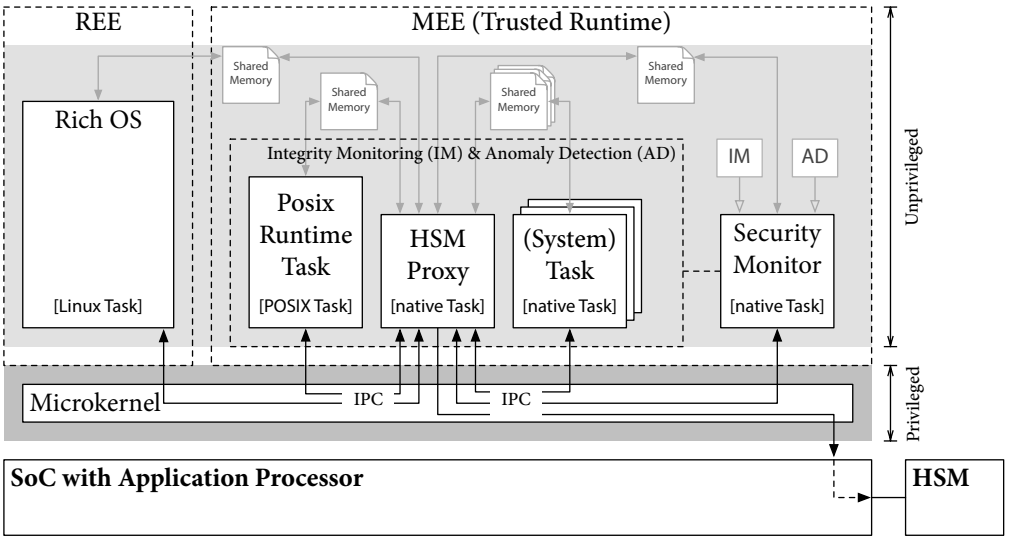

Figure 8.1: Microkernel-based System Architecture with Multi-Context HSM

There is, however, one exception: the *security monitor*. This component is one of the early tasks, which can start other tasks and is able to *measure the integrity* of a selection of tasks as indicated by the dashed box in Figure 8.1. Since the security monitor, which is a critical task and, hence, realized as a native microkernel task, holds advanced capabilities, e.g., the right to directly access the memory of other tasks, it can effectively monitor their behavior and might *detect anomalies*. Please note that because anomaly detection itself and the design of the necessary algorithms are not within the scope of this thesis, we refer to Section 3.1 for related work regarding anomaly detection. Nevertheless, the security monitor is conceptually designed to store the integrity values measured at load-time in the task-specific context as well as keep a log of detected anomalies inside the HSM.

The design of our multi-context HSM, which is schematically depicted in Figure 8.2, implements the functionality of a TPM and includes hardware-based security features, such as protected memory, a true random number generator (TRNG), and cryptographic engines for hash functions, MACs, and encryption algorithms like RSA or ECC.

Based on the hardware components, the HSM firmware realizes *scheduling*, *multiplexing*, and *prioritizing of separate contexts*, which can handle cryptographic keys, store integrity measurements in shared and individual PCRs, and log atypical run-time events in per-context anomaly detection records (ADRs). That way each task can have its individual key hierarchy, anomaly detection status, and its very own set of hardware-based integrity measurement registers. However, the HSM also allows to (physically) share certain PCRs, e.g., to store common integrity measurements for the boot loader or the microkernel, in order to make the HSM design more efficient.

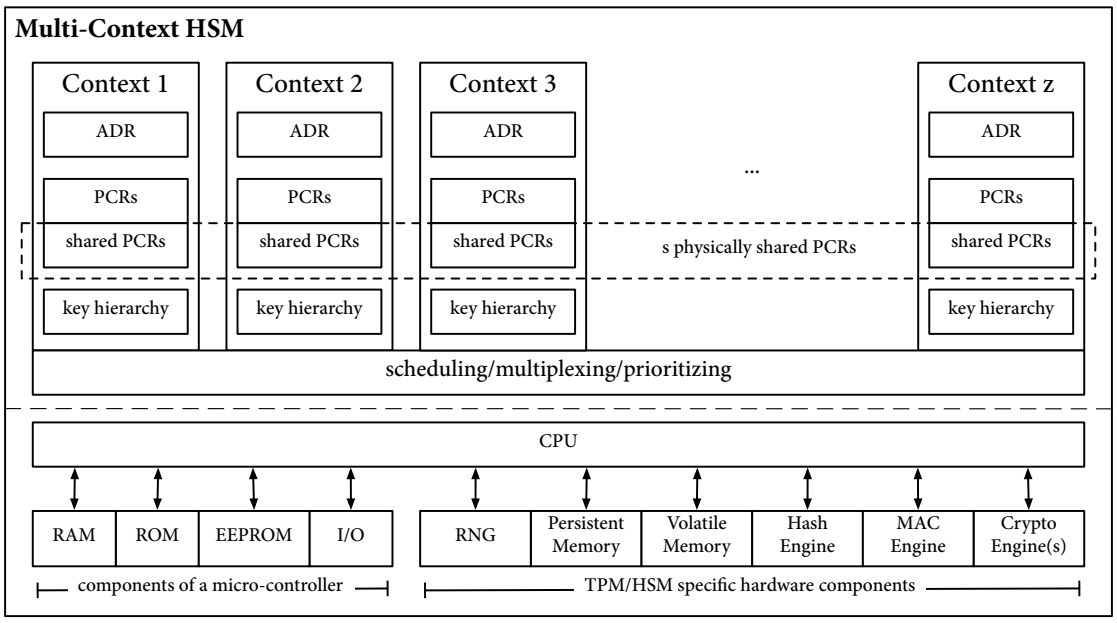

Figure 8.2: Design of a Multi-Context HSM

8.3 Integrity Verification of Multiple Microkernel Tasks as Basis for a Secure Code Update

Based on our extended system architecture, we present our approach to implicitly verify the integrity of multiple separated microkernel tasks and communicate the result to a remote verifier without the need for expensive cryptographic operations. The attestation mechanism enables a secure code update and recovery of (compromised) tasks based on the integrity of the core system, in particular the microkernel and other existing tasks.

The main idea of our lightweight attestation mechanism is to verify the integrity of a number of tasks "locally" rather than sending digitally signed integrity values to a remote verifier, which then has to check the signatures and evaluate the integrity values. Our attestation protocol instead verifies the trustworthiness of tasks by loading task-specific keys into the key slots inside the HSM. The load operation is only possible if the specified tasks have not been tampered with, because the keys have been cryptographically bound to the correct integrity measurements of the tasks and their typical behavior, which is monitored by the anomaly detection component of the security task.

We presented a similar concept of implicit attestation for non-virtualized systems with a baseband stack executed on a dedicated baseband processor in Chapter 6 and also used local attestation (an integrated challenger) in Chapter 7. In contrast to the attestation mechanism presented in this chapter, the attestation for mobile baseband stacks focuses on the baseband processor and only requires one cryptographic context, because the baseband generally does not (yet) use virtualization. Similarly, the secure loading concept focuses on the attestation of a microkernel-based system, which is however handled like a monolithic system. In this section, we propose to specify and attest individual tasks (with separate cryptographic contexts) in addition to the microkernel, which are can be verified separately by the remote party. As a result, we can limit the remote attestation to the tasks, which belong to the relevant TCB for a specific application or use case. Furthermore, the remote attestation mechanism present in this chapter has the ability to include results of an anomaly detection component, which is part of our extended system architecture and provides measurement events that characterize the current run-time behavior of a task.

The rest of the section is structured as follows. First, we revisit our notation in Section 8.3.1, where we extend some of the designations defined in Section 6.3.1. After that, we specify the cryptographic keys for our attestation protocol in Section 8.3.2. Based on the extension of the notation and definitions, we describe the attestation scheme in Section 8.3.3, which is the basis for the secure code update protocol presented in Section 8.3.4.

8.3.1 Notation – Part 2 of 3

In this section, we extend part 1 of our notation, which we presented in Section 6.3.1. First, we adapt the definition of a platform configuration. then we specify anomalies in form of events and a threshold for the probability of an attack. Finally, we extend the wrapping functionality to include the threshold value in a similar way it uses a platform configuration as a condition for unwrapping a key.

Platform Configurations in Multiple Cryptographic Contexts

For our attestation protocol, we presume that the load-time integrity of a microkernel task can be adequately described by a set of measurement values, which are securely stored in the PCRs of our multi-context HSM. Thus, the PCRs values, which cryptographically represent the *context* c, are referred to as *platform configuration* $P_c := (PCR_c[i_1], \ldots, PCR_c[i_k])$, where $i \in \{0 \ldots n{-}1\}$, $k \leq n$, and n is the number of available PCRs. This is a minor extension to the notation presented in Section 6.3.1 and only accounts for multiple tasks/contexts c.

Thresholds for Anomaly Detection

In addition to the load-time integrity measurements, the security task also monitors the run-time behavior of critical tasks. Based on machine learning algorithms [Xia13c; Xia13d], the anomaly detection component of the security task monitors, for instance, the order of system calls and assesses the probability for an attack. In case of an anomaly with *probability* p, an *event* $e = (m, p)$ is recorded by the security task, which securely stores a log message m in the anomaly detection record of task c (A_c) while increasing the *probability for an attack* by using $\mathsf{append}_{ADR}(A_c, e)$. If the probability value in A_c exceeds a *threshold* t_c, the task must or should be considered compromised. In order to compensate for false-positives, the probability decreases over time very slightly, which might, however, be a security weakness and needs further research. As a consequence, we need to exclude false positives for now.

Wrapping Keys to Platform Configurations and Thresholds

To cryptographically bind (or *wrap*) a key K to a particular system state, the HSM links the key to the specified platform configuration P and encrypts it with a public key pk_{wrap}. We extend this definition by also binding the wrapped key to an anomaly detection probability threshold t_c, that is $\{K\}_{pk_{\text{wrap}}}^{P_c, t_c}$. To load this wrapped key, the security monitor must also verify that the current probability stored in A_c is below t_c, which allows for more dynamic run-time verifications. Like the changes to the notation of a platform configuration, this extension is also minor compared to the notation presented in Section 6.3.1 and only includes the threshold value. The threshold and the platform configuration, in turn, account for multiple tasks/contexts c.

8.3.2 Cryptographic Keys

Before we describe the remote attestation mechanism and the secure code update protocol, we define the relevant cryptographic keys in this section. Those keys mainly include a number of integrity/attestation keys, which are cryptographically bound to a trusted platform configuration using a global wrapping key.

For each context c, we thus define an integrity key $K_{\text{int}}^{c} = (pk_{\text{int}}^{c}, sk_{\text{int}}^{c})$, i.e., an ordinary binding key in the HSM key hierarchy, which needs to be loaded into the HSM for a successful integrity verification and attestation of the corresponding task. As shown in Figure 8.3, the integrity keys K_{int}^{c} are encrypted with the public portion of a shared non-migratable wrapping key $K_{\text{wrap}} = (pk_{\text{wrap}}, sk_{\text{wrap}})$ and cryptographically bound to a trusted platform configuration P_c as well as an anomaly detection probability threshold t_c. The wrapped integrity keys are denoted as $\{K_{\text{int}}^{c}\}_{pk_{\text{wrap}}}^{P_c,t_c}$, where c specifies the cryptographic context inside the HSM and the task in the MEE. The assignment of tasks to contexts is enforce by the microkernel, the security monitor, and the HSM proxy.

The necessary authentication value $Auth_{\text{wrap}}$ for loading the wrapping key is only known to the HSM and the remote verifier, which initially provisions that wrapping key. The verifier $\mathcal{V}$ also has access to certain hash values, which combine a particular wrapped integrity key with a `TPM_LoadKey2` ordinal (`TPM_ORD_LoadKey2`) according to the TPM 1.2 Main Specification [Tru11, Part 3, Section 10.5 (`TPM_LoadKey2`), Incoming Operands and Sizes, 1S and 2S]. The resulting hash is denoted as $h_c = SHA\text{-}1(\texttt{TPM\_ORD\_LoadKey2} \parallel \{K_{\text{int}}^{c}\}_{pk_{\text{wrap}}}^{P_c,t_c})$.

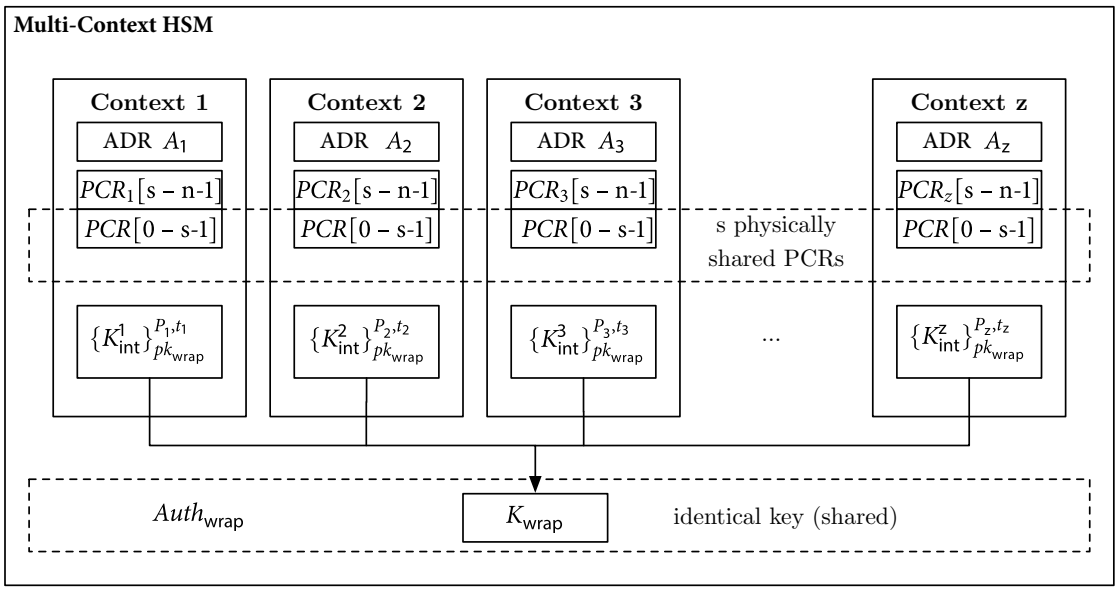

Figure 8.3: Cryptographic Keys, PCRs, and ADRs for Multiple Separated Contexts

8.3.3 Integrity Verification and Attestation of Multiple Tasks

To verify the integrity of one or more microkernel tasks, a remote verifier $\mathcal{V}$ sends an *attestation request (req)* to the prover $\mathcal{P}$ as depicted in Figure 8.4. The attestation request specifies k out of n microkernel tasks, which should be included into the attestation procedure, i.e., $req = \{c_i\}$ with $c,i \in \{1,\ldots,n\}$ and $|req| = k$.

Based on the request, $\mathcal{P}$ first transmits a set of random numbers $nonce_{\mathcal{P}_c}$, which are specifically calculated by the HSM for the selected tasks with context c. The random numbers are required to generate the authentication values $Auth^c_{\text{int}}$ and prevent replay attacks. $\mathcal{V}$ calculates the set of authentication values as

$$Auth^c_{\text{int}} = HMAC\big(Auth_{\text{wrap}}, h_c \,||\, nonce_{\mathcal{P}_c} \,||\, nonce_{\mathcal{V}}\big)\,, \tag{8.1}$$

where $nonce_{\mathcal{V}}$ is a random number selected by the verifier (Figure 8.4, step 1). The calculation of $Auth^c_{\text{int}}$ shown in Equation 8.1 follows the authentication for TPM commands as specified by the TCG, more precisely for the load command [Tru11, p. 76]. The authentication values $Auth^c_{\text{int}}$ and $nonce_{\mathcal{V}}$ are then sent to $\mathcal{P}$, which is now able to generate the command to load the task-specific integrity keys K^c_{int}. It is important to note that $\mathcal{P}$ has neither knowledge about nor access to authentication value $Auth_{\text{wrap}}$, which is only known to $\mathcal{V}$ and the HSM. As a consequence, $\mathcal{P}$ is not able to calculate the authentication HMACs without $\mathcal{V}$.

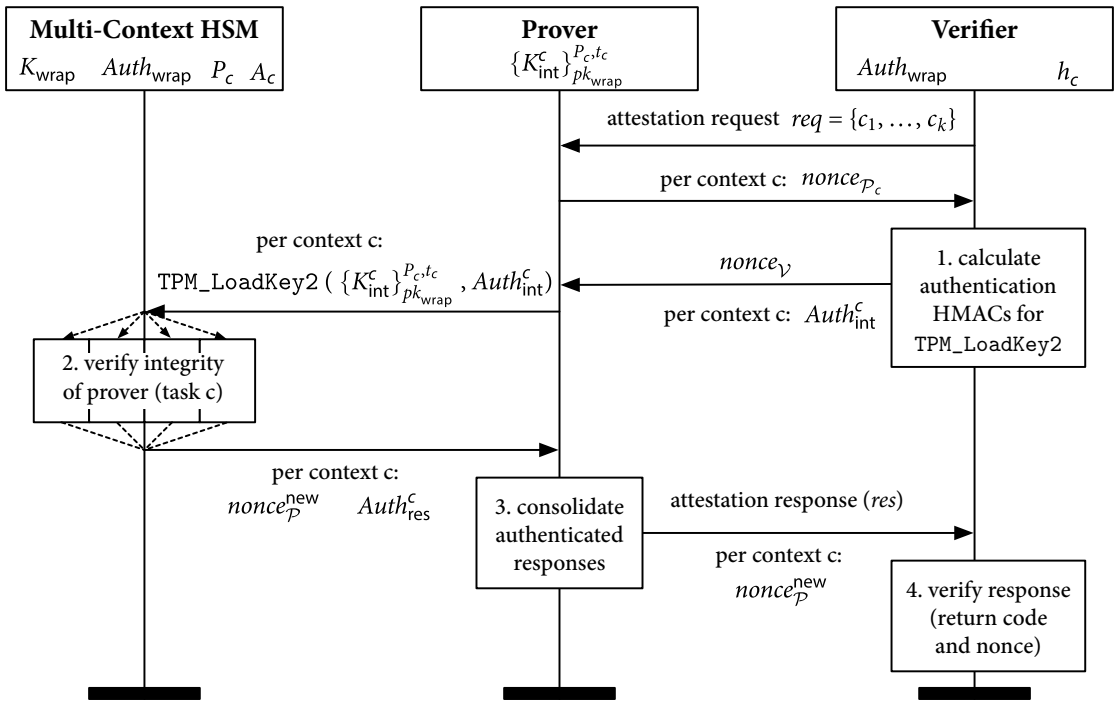

Figure 8.4: Attestation Protocol for Multiple Separated Tasks

To load the keys K_{int}^{c} into the HSM and, thereby, generate the implicit proof for the trustworthiness of the corresponding tasks, $\mathcal{P}$ simply needs to generate a `TPM_LoadKey2` command per context as shown in Figure 8.4. When the HSM receives a command for context c, it verifies the authentication value $Auth_{\text{int}}^{c}$ and compares the current platform configuration P_{c}' with P_{c}, which has been specified when the corresponding integrity key K_{int}^{c} was wrapped (Figure 8.4, step 2). It also checks the anomaly detection record for any log entries (more precisely, whether the current probability value in A_c is above the probability threshold t_c specified during the wrapping step), which might indicate that the task was compromised during run-time. If the verification is successful and no anomalies were detected, the key is decrypted and loaded. However, if the key is already loaded into a key slot, an efficient HSM only verifies the wrapping conditions and omits the decryption.

After a successful load operation, the HSM generates HMAC-protected result messages and a set of new random numbers $nonce_{\mathcal{P}_c}^{\text{new}}$ for each task/context c . The result messages mainly include a *return code (rc)*, which indicates the success of the load operation (e.g., `TPM_SUCCESS`), the fixed command ordinal for the `TPM_LoadKey2` operation (`TPM_ORD_LoadKey2`), a new nonce $nonce_{\mathcal{P}_c}^{\text{new}}$, and the nonce selected by the verifier ($nonce_{\mathcal{V}}$). Those values are protected by an HMAC, which is denoted as $Auth_{\text{res}}^{c}$ and calculated as

$$Auth_{\text{res}}^{c} = HMAC\big(Auth_{\text{wrap}}, (SHA\text{-}1(rc \,||\, \texttt{TPM\_ORD\_LoadKey2}) \,||\, nonce_{\mathcal{P}}^{\text{new}} \,||\, nonce_{\mathcal{V}})\big) \,. \quad (8.2)$$

Again, it is important to note that the HMACs $Auth_{\text{res}}^{c}$ are calculated based on $Auth_{\text{wrap}}$, which is the shared authentication value between the HSM and the verifier $\mathcal{V}$ and not known to the prover.

Before the prover $\mathcal{P}$ sends the new random numbers $nonce_{\mathcal{P}_c}^{\text{new}}$ and the HMACs $Auth_{\text{res}}^{c}$, which carry implicit proof that the attested tasks are still trustworthy, to $\mathcal{V}$, the prover $\mathcal{P}$ can reduce the size of the *attestation response (res)* by hashing the HMACs (Figure 8.4, step 3), i.e.,

$$res = H(Auth_{\text{res}}^{c}) \;\; \forall c \in req \,. \quad (8.3)$$

As a consequence, the efficiency of the transmitted attestation result can be increased and does not lose cryptographic information needed to verify the integrity of the selected tasks if the attestation is successful. However, in case the attestation fails, the prover should temporarily store the individual HMACs and send them separately in order to allow for a more fine-grained diagnosis and possibly a recovery of compromised tasks. For a recovery, the verifier only considers the most basic security-critical tasks, such as the microkernel, the security monitor, and the HSM proxy, and grants access to a trusted version of the compromised software component as a failsafe. This recovery version of a compromised task can be provided using the secure code update protocol, which is described in more detail in the next section and extends our secure loading concept presented in the previous chapter with the implicit setup of an integrity key for the new task.

Finally, in step 4, the verifier $\mathcal{V}$ checks the prover's attestation response *res* by comparing it with a freshly generated hash res' as indicated in Figure 8.4, which is calculated as follows: First, the verifier calculates a static hash value

$$h_{\text{LoadKey2}} = \textit{SHA-1}((rc = \texttt{TPM\_SUCCESS}) \,||\, \texttt{TPM\_ORD\_LoadKey2}) \,, \tag{8.4}$$

where $\mathcal{V}$ assumes that the `TPM_LoadKey2` operation executed by the prover $\mathcal{P}$ has been successful, i.e., the return code must be `TPM_SUCCESS`. In other words, the verifier expects that the load operation is successful on a system, which is unmodified and still trustworthy. As a result, the verifier can easily calculate the static SHA-1 hash value h_{LoadKey2}, which is independent of a particular task.

Corresponding with Equation 8.2, the hash value h_{LoadKey2} defined in Equation 8.4 is then used to freshly calculate the task-specific HMACs $Auth_{\text{res}}^{c\prime}$ based on the random numbers and the authentication value $Auth_{\text{wrap}}$ as

$$Auth_{\text{res}}^{c\prime} = HMAC\big(Auth_{\text{wrap}}, (h_{\text{LoadKey2}} \,||\, nonce_{\mathcal{P}}^{\text{new}} \,||\, nonce_{\mathcal{V}})\big) \,. \tag{8.5}$$

By hashing those HMAC values (cf. Equation 8.3), i.e.,

$$res' = H(Auth_{\text{res}}^{c\prime}) \;\forall c \in req \,, \tag{8.6}$$

and comparing the result res' to the attestation response *res*, the verifier $\mathcal{V}$ can reason about the integrity and trustworthiness of the selected tasks.

At this point, it is important to note that the verifier $\mathcal{V}$ can trust the individual attestation results (and the combined hash) to implicitly and reliably communicate proof for the trustworthiness of the selected tasks, because the results of the `TPM_LoadKey2` operation are protected by an HMAC. This HMAC is based on the shared secret, the authentication value $Auth_{\text{wrap}}$, which is only known to the HSM and $\mathcal{V}$. As such, it is not available or in any way accessible to other parties like the prover, which might execute compromised tasks. Hence, this long-term shared secret must be protected by the HSM as well as the verifier, which can also use an HSM to prevent extraction by an attacker.

In addition, the attestation response messages also include random numbers, which protect against replay attacks by providing verifiable proof that the attestation response message is indeed fresh. It is important to see that those random numbers are part of the HMAC as defined in Equation 8.2, which means they are incorporated in the attestation response generated by the HSM and implicitly validated as part of the HMAC verification.

8.3.4 Updating a Task After Verifying the Integrity of Existing Tasks

Based on the attestation protocol for existing microkernel tasks, we now describe our code update protocol, which allows to *update a task*, *creates verifiable proof*, and *maintains the ability to attest* both, the tasks and the system.

The main idea of the update protocol is that the prover creates and loads a new integrity key, which is specific to the new task, i.e., wrapped to the integrity values of the code update and the corresponding request. To ensure that the key is actually wrapped to the correct integrity measurements, the verifier provides a specific cryptographic authorization value, which only the verifier can calculate. When the HSM receives this authorization, it checks whether the new key will be wrapped to the correct integrity values before creating and wrapping the key. In combination with an efficient attestation of the code update, the verifier can ensure the authenticity of the initial update request and has verifiable evidence for the transaction. The prover, on the other hand, can use the new integrity key to create proof of a successful load operation in order to obtain the code update.

For a code update U, the prover $\mathcal{P}$ initiates the protocol by sending a *update request* (req_U) to the verifier $\mathcal{V}$ as depicted in Figure 8.5 (step 1). Apart from the requested software update identified by ID_U, the message includes the cryptographic values to generate the authorization value referred to as $Auth^{\mathsf{U}}_{\mathsf{pub}}$, which is used to create the new wrapped integrity key $K^{\mathsf{U}}_{\mathsf{int}}$. In more detail, the prover encrypts the authentication value to *use* and *migrate* the new integrity key, which are denoted as $Auth_{\mathsf{usage}}$ and $Auth_{\mathsf{migration}}$ (or *usageAuth* and *migrationAuth*, respectively, following the TCG specification [Tru11]). The request also includes a nonce, which we refer to as $nonce_{\mathcal{P}}$.

When the verifier receives the request from the prover $\mathcal{P}$, we assume that $\mathcal{V}$ might require to verify the trustworthiness of existing tasks for safety and security reasons before allowing to update or install a new software component. That is why the verifier $\mathcal{V}$ first initiates an attestation, which checks the authenticity and integrity of tasks running on $\mathcal{P}$'s system as described in the previous section.

After a successful attestation of the relevant tasks, the verifier returns the encrypted authorization HMAC $Auth^{\mathsf{U}}_{\mathsf{pub}}$ as well as a hash of the code update, denoted as h_{U} (Figure 8.5, step 2). The HMAC authorizes the creation of the new integrity key and is calculated as

$$Auth^{\mathsf{U}}_{\mathsf{pub}} = HMAC(Auth_{\mathsf{wrap}}, (h_{\mathsf{CreateWrapKey}} \,||\, nonce_{\mathcal{P}} \,||\, nonce_{\mathcal{V}_1})) \,. \tag{8.7}$$

In this equation the hash value $h_{\mathsf{CreateWrapKey}}$ is generated as

$$h_{\mathsf{CreateWrapKey}} = SHA\text{-}1(\texttt{TPM\_ORD\_CreateWrapKey} \,||\, Auth_{\mathsf{usage}} \,||\, Auth_{\mathsf{migration}} \,||\, keyInfo) \,, \tag{8.8}$$

where `TPM_ORD_CreateWrapKey` is the fixed ordinal for the `TPM_CreateWrapKey` command.

The structure *keyInfo* defines the cryptographic properties of the new key and follows the TCG specification [Tru11, Part 2, p. 89]. In particular, the structure specifies the trusted platform configuration the key is wrapped to, which includes at least the integrity values of the *microkernel*, the *update* and the *request*. Additionally, the structure also specifies the threshold value t_c.

To create the new key, $\mathcal{P}$ first extends the PCRs of context U with the hash of the code update and the request (Figure 8.5, step 3). After that, the prover creates the new integrity key $K^{\mathsf{U}}_{\mathsf{int}}$, which is encrypted with the wrapping key K_{wrap} and cryptographically bound to the trusted platform configuration, which is implicitly encoded into the HMAC $Auth^{\mathsf{U}}_{\mathsf{pub}}$ (step 4).

After a successful generation of the wrapped integrity key, the prover $\mathcal{P}$ sends a hash of the key, $SHA\text{-}1(\texttt{TPM\_ORD\_LoadKey2} \parallel \{K^{\mathsf{U}}_{\mathsf{int}}\}^{P_{\mathsf{U}},t_{\mathsf{U}}}_{pk_{\mathsf{wrap}}})$ and a random number $nonce_{\mathcal{P}_{\mathsf{U}}}$ to the verifier, which initiates an attestation procedure for the code update (cf. Figure 8.5). The verifier creates the authentication value $Auth^{\mathsf{U}}_{\mathsf{int}}$ and sends it to the prover together with a random number $nonce_{\mathcal{V}_2}$. $\mathcal{P}$ uses both values to load the new integrity key, which implicitly verifies the authenticity and integrity of the code update, more precisely hash h_{U} (Figure 8.5, step 5). The attestation response, in particular $Auth^{\mathsf{U}}_{\mathsf{res}}$, is then sent to $\mathcal{V}$, which can check the result. If the attestation was successful, the verifier sends the actual code update to the prover. $\mathcal{P}$ can check the integrity of the code update by freshly generating the hash value $H(\mathrm{U})$ and comparing it to the previously received hash h_{U}, which has already been implicitly verified. If both values match, the code update is trustworthy and can be installed, which can effectively upgrade an existing task or recover a compromised task.

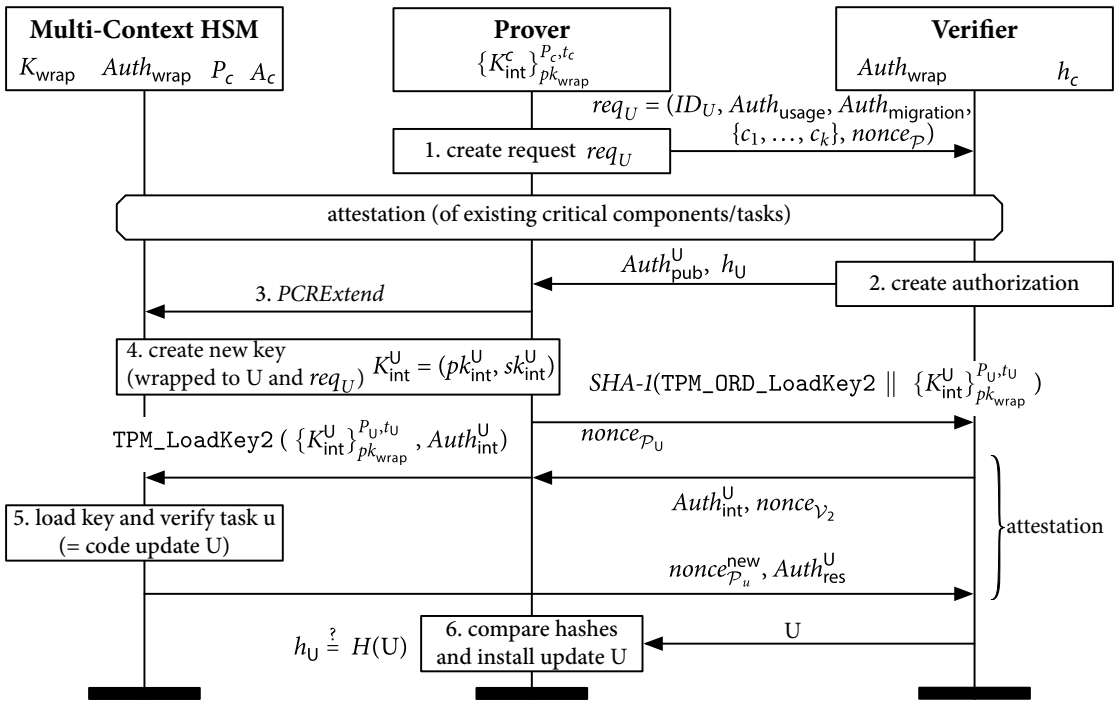

Figure 8.5: Secure Code Update and Recovery Protocol

8.4 Informal Security Analysis and Formal Protocol Verification

In this section, we analyze the security of our two proposed protocols. First, we discuss the assumptions and security properties of the attestation mechanism in Section 8.4.1 and present the results of a formal verification in Section 8.4.2. Based on the discussion of the attestation protocol, we analyze the security of our code update and recovery protocol in Section 8.4.3.

8.4.1 Security Discussion of the Attestation Protocol

We start the analysis of our implicit attestation mechanism from the premise that hardware attacks, such as the TPM *cold boot attack* [Hal09] or manipulations of the HSM communication bus [Win12], are out of scope, especially if those attacks are based on existing flaws in the platform's hardware. For example, the Intel architecture does not provide secure mechanisms to tightly couple an HSM and in particular TPMs to the platform hardware, which enables attacks on the communication bus, especially the Low Pin Count (LPC) bus. As a consequence, we mainly focus on software attacks as specified in Section 4.2 and state certain hardware attacks as fundamental limitation.

Furthermore, we assume that the platform configuration represented by the PCRs correctly reflects the load-time integrity of the system at any time. As mentioned in Chapter 6, this assumption would actually require CFI and periodic or on-demand integrity protections mechanisms, such as IBM's IMA [Sai04] or our integrity measurement concept as presented in the previous chapter. We also assume that the platform configuration is created during authenticated boot starting from an immutable CRTM and cannot be easily forged by exploiting software vulnerabilities, e.g., buffer overflows, by code-reuse attacks, or ROP. Thus, both assumptions show limitations that can, however, be addressed using complementary integrity verification concepts (cf. Section 3.1).

In addition, we exclude the TOCTOU problem, which might affect the validity of the attestation result at the time of use, even in case of a local challenger as described in the previous chapter. In contrast to the attestation mechanism based on a local challenger, which verifies the platform integrity using traditional remote attestation, i.e., by verifying a TPM-signed quote for the PCRs based on a measurement list, the mechanism presented in this chapter relies on implicit attestation. Our implicit attestation mechanism reduces the impact of the TOCTOU problem by eliminating the need for digital signatures and the verification of measurement entries, which decreases the verification time of the attestation result. However, similar to the local challenger concept, our attestation mechanism cannot completely solve the TOCTOU problem, because there is always a time gap between the verification and the use of the attestation result. As a consequence, we exclude this specific problem and rely on the fact our microkernel-based system is much harder to compromise than a generic Linux-based system. Given any average time gap, we assume that the probability of a successful attack on a complex, monolithic system is significantly higher and takes less time compared to our microkernel-based system architecture. Hence, it is reasonable to assume that the TOCTOU problem is less relevant to our microkernel-based system architecture.

Based on the attacker model described in Section 4.2 and the assumptions detailed above, we now analyze our attestation protocol. In the first attack scenario, the adversary $\mathcal{A}$ might try to extract and obtain the authentication values or cryptographic keys in order to compromise the attestation. However, the wrapping key K_{wrap} is a non-migratable key as specified in Section 8.3.2, which means it is always securely stored inside the HSM. In addition, the authentication value for the wrapping key $Auth_{\text{wrap}}$ is only known to the verifier and never made public. Since this fact is very important for our attestation protocol, we will verify it formally in the next section using an automated verification tool for cryptographic protocols.

$\mathcal{A}$ might also attempt to create and wrap a new integrity key to an insecure platform configuration, which does not include, for instance, any PCRs values, and replace the existing wrapped integrity key $\{K^c_{\text{int}}\}^{P_c,t_c}_{pk_{\text{wrap}}}$ for a task c with a compromised integrity key. However, this is not possible, because the non-migratable wrapping key K_{wrap} is securely stored inside the HSM and the corresponding authentication value $Auth_{\text{wrap}}$ is only known to the verifier (cf. previous attacks scenario). In other words, the attacker is not able to create a new integrity key underneath the wrapping key K_{wrap}, because the authentication value is $Auth_{\text{wrap}}$ is a secret. Since this is a highly critical aspect, we will also check in the formal verification that the wrapping key K_{wrap} does not leave the HSM during the attestation.

In a different attack scenario, the adversary $\mathcal{A}$ might try to compromise the task by manipulating its binary code. However, since the task is measured before it is executed, the verifier can easily detect the manipulation, because the integrity measurements in the PCR would be incorrect, the key could not be loaded, and an attestation would ultimately fail. The failure is indicated by the return code in the result message from the HSM. If the attacker compromises the binary and replays an old result message in order to convince the verifier that the load operation has been successful, the verifier can detect the attack by checking the result, in particular the random number. Since the original random number is created by the verifier as $nonce_{\mathcal{V}}$ and the replayed nonce most likely does not match the expected random number, the replay attack can be easily detected through a simple comparison by the verifier.

Finally, the attacker $\mathcal{A}$ might try to compromise the behavior of a task during run-time. In this concept, the security task aims to detect the anomaly, e.g., in the number or the order of certain system calls, by monitoring the behavior of the attacked task. For instance, if a banking application is usually contacting the bank's server for online banking directly and only requires a single connection, attacker might try to change the server's contact address or open additional connections to other servers by manipulating the banking task. In that case, the security monitor adds an event (with a high probability, since it detected an attack) to the ADR, which is securely stored inside our HSM. So, if the prover tries to load the corresponding integrity key for the compromised task, the HSM prevents a successful load operation, since the current probability value is above the threshold t_c for any detected attack. As a consequence, the attestation fails.

8.4.2 Formal Verification of the Attestation Protocol

To substantiate our security discussion, we also formally verified relevant security-critical properties of our attestation mechanism using *ProVerif* [Bla01], an automated verifier for security protocols. For this purpose, we developed a formal model that implements crucial aspects of our attestation protocol (the *secrecy* of the symmetric keys), which are also the core of the code update protocol.

The model for our remote attestation specifies a verifier, a prover, and an HSM, in this case with two contexts for simplicity (cf. Listing 8.1). To initiate the attestation protocol, the verifier first creates a request (`req` of type `attestationRequest`), sends it to the prover, and receives the corresponding random numbers (`noncePc1` and `noncePc2`). The verifier then creates a new `nonceV`, which is later used to implicitly verify the freshness of the attestation result, since it is part of the result values returned by the HSM. Lastly, the verifier calculates the authentication values `Authc1` and `Authc2` (lines 9 and 11) based on the hash of the wrapped integrity keys, i.e., `hWrappedKint1` and `hWrappedKint2`, the random numbers `noncePc1` and `noncePc2` from the prover as well as `nonceV` from the verifier. The key for the both HMACs is `AuthWrap`, which is only known to the verifier and the HSM. This critical aspect is modeled using a global symmetric key `AuthWrap`, which is declared as `private` and provided to the verifier and the HSM only.

The prover receives the HMACs and loads the wrapped keys, `wrappedKey1` and `wrappedKey2` (lines 21 and 23). The HSM verifies the nonces (lines 34 and 40), compares the current platform configuration `cpc` with the trusted one `tpc` (lines 35 and 41), and checks if the probability value `p` matches `T` (lines 36 and 42; simplified with $T = 0$, i.e., no anomalies tolerated). Finally, the HSM generates the result message. The prover forwards the result and the nonces `noncePc1New` and `noncePc2New` to the verifier.

Once the attestation result (`res` of type `hash`) has been received, the verifier assumes a successful `TPM_LoadKey2` operation (hence, the hardcoded value `true` in lines 14 and 15), which is a necessary requirement for a trustworthy system. To verify the attestation result, the verifier finally calculates a fresh SHA-1 hash based on its `nonceV` using `AuthWrap`, hashes the individual HMACs, and compares it to `res` (line 16). This verification of the attestation result implicitly checks `nonceV` and ultimately the trustworthiness of the selected tasks, because the prover's HSM verified the platform configuration when it loaded the integrity keys.

In *ProVerif*, our formal verification is automated with queries, which check if the authentication value `AuthWrap` (cf. line 1) or the wrapping key `Kwrap` (cf. line 2) are disclosed during the attestation. A third query additionally checks if the attestation was successful (cf. line 3), which is indicated by an event that is gernerated if and only if the attestation response can be successfully verified. Our results show that *ProVerif* cannot find an attack path for the `AuthWrap` or `Kwrap`, because they are kept secret and are never transferred via network, and that the attestation is successful if all verification steps are successfully passed.

```
free AuthWrap:symKey [private]. query attacker(AuthWrap).
free Kwrap:sKey [private ].     query attacker(Kwrap).
event successfulAttestation .   query event( successfulAttestation ).

let Verifier (AuthWrap:symKey, hWrappedKint1:hash, hWrappedKint2:hash) =
    new req:attestationRequest; out(c2, req);
    in(c2, noncePc1:bitstring ); in(c2, noncePc2:bitstring );
    new nonceV:bitstring;
    let Authc1= MAC(c(c(h2Bs(hWrappedKint1), noncePc1), nonceV), AuthWrap) in
    out(c2, Authc1);
    let Authc2= MAC(c(c(h2Bs(hWrappedKint2), noncePc2), nonceV), AuthWrap) in
    out(c2, Authc2); out(c2, nonceV); in(c2, res:hash);
    in(c2, noncePc1New:bitstring); in(c2, noncePc2New:bitstring);
    let resAuth1 = MAC(c(c(h2Bs(SHA1(cBoolBs(true, load))), noncePc1New), nonceV), AuthWrap) in
    let resAuth2 = MAC(c(c(h2Bs(SHA1(cBoolBs(true, load))), noncePc2New), nonceV), AuthWrap) in
    if res = SHA1(c(MAC2Bs(resAuth1), MAC2Bs(resAuth2))) then event successfulAttestation ; 0.

let Prover(wrappedKint1:wKey, wrappedKint2:wKey, noncePc1:bitstring, noncePc2:bitstring ) =
    in(c2, req:attestationRequest); out(c2, noncePc1); out(c2, noncePc2);
    in(c2, Authc1:mac); in(c2, Authc2:mac); in(c2, nonceV:bitstring );
    let cmd1 = createLoadKeyCmd(wrappedKint1, Authc1, noncePc1, nonceV) in
    out(c1, cmd1);
    let cmd2 = createLoadKeyCmd(wrappedKint2, Authc2, noncePc2, nonceV) in
    out(c1, cmd2);
    in(c1, res:hash);
    in(c1, noncePc1New:bitstring); in(c1, noncePc2New:bitstring);
    out(c2, res); out(c2, noncePc1New); out(c2, noncePc2New); 0.

let HSM(AuthWrap:symKey, noncePc1:bitstring, noncePc2:bitstring, cpc1:PConf, p1:ADR, cpc2:PConf, p2:ADR) =
    new noncePc1New:bitstring; new noncePc2New:bitstring;
    in(c1, cmd1:LoadKeyCommand); in(c1, cmd2:LoadKeyCommand);
    if getAuthc(cmd1) = MAC(c(c(h2Bs(SHA1(c(load, wKey2Bs(getWrappedKey(cmd1))))), noncePc1),
                            getNonceV(cmd1)), AuthWrap) then
    if noncePc1 = getNoncePc(cmd1) then      (* comment: check noncePc1 *)
    if cpc1 = pc(getWrappedKey(cmd1)) then   (* comment: check P_1 *)
    if p1 = ard(getWrappedKey(cmd1)) then    (* comment: check ADR_1 *)
    let resAuth1 = MAC(c(c(h2Bs(SHA1(cBoolBs(true, load))), noncePc1New), getNonceV(cmd1)), AuthWrap) in
    if getAuthc(cmd2) = MAC(c(c(h2Bs(SHA1(c(load, wKey2Bs(getWrappedKey(cmd2))))), noncePc2),
                            getNonceV(cmd2)), AuthWrap) then
    if noncePc2 = getNoncePc(cmd2) then      (* comment: check noncePc1 *)
    if cpc2 = pc(getWrappedKey(cmd2)) then   (* comment: check P_2 *)
    if p2 = ard(getWrappedKey(cmd2)) then    (* comment: check ADR_2 *)
    let resAuth2 = MAC(c(c(h2Bs(SHA1(cBoolBs(true, load))), noncePc2New),getNonceV(cmd2)), AuthWrap) in
    let res = SHA1(c(MAC2Bs(resAuth1), MAC2Bs(resAuth2))) in
    out(c1, res); out(c1, noncePc1New); out(c1, noncePc2New); 0.

process
    new noncePc1:bitstring; new noncePc2:bitstring;
    new Kint1:intKey; new Kint2:intKey;
    let wKint1 = wrapKey(Kint1, pk(Kwrap), tpc1, T1) in    (* cf. Sect. 8.3.1 *)
    let wKint2 = wrapKey(Kint2, pk(Kwrap), tpc2, T2) in    (* cf. Sect. 8.3.1 *)
    (! Verifier (AuthWrap, SHA1(c(load, wKey2Bs(wKint1))), SHA1(c(load, wKey2Bs(wKint2))))) |
    (! Prover(wKint1, wKint2, noncePc1, noncePc2)) | (! HSM(AuthWrap, noncePc1, noncePc2, tpc1, T1, tpc2, T2))
```

Listing 8.1: *ProVerif* Code for the Attestation Mechanism (excerpt)

8.4.3 Security Discussion of the Code Update Protocol

To evaluate the security of our code update protocol, we now discuss different attack scenarios, where an adversary might either try to compromise the authenticity and integrity of a code update. Please note that the security discussion focuses on integrity and authenticity, because we assume that the confidentiality of the code update is adequately protected with a symmetric encryption key as described in the previous chapter. That means, we assume that our microkernel-based system can use the integrity keys $K^{\mathsf{U}}_{\mathsf{int}}$ as binding keys to decrypt an ephemeral encryption key as detailed in Chapter 7. As a consequence, we concentrate on attacks against the integrity and authenticity of the code update, which are directly related to our implicit attestation mechanism.

For the first attack, we presume that the adversary $\mathcal{A}$ tries to compromise the integrity of the code update by replacing it with a manipulated version, e.g., during transfer from the verifier $\mathcal{V}$ to the prover $\mathcal{P}$. However, this attack can be easily detected, since $\mathcal{P}$ generates a new integrity key, which $\mathcal{V}$ wraps to the integrity measurement of untampered code update, i.e., h_{U}, by specifying the platform configuration that includes the integrity measurement of the code update. To receive the actual code update, $\mathcal{P}$ needs to successfully load the new integrity key in order to be able to send the correct attestation result to $\mathcal{V}$. To verify the integrity of the code update, $\mathcal{P}$ compares a fresh hash of the code update with the previously received hash h_{U}, which must have been part of the wrapped key and the trusted platform configuration. $\mathcal{P}$ also implicitly verifies the authenticity of the code update, because only $\mathcal{V}$ knows the authentication value $Auth_{\mathsf{wrap}}$ for wrapping key K_{wrap}. As a result, only $\mathcal{V}$ is able to generate the correct authorization value $Auth^{\mathsf{U}}_{\mathsf{pub}}$ for creating the new wrapped integrity key.

In a second attack, the adversary $\mathcal{A}$ might try to compromise the code update request (*req*) sent by the prover $\mathcal{P}$. To ensure the authenticity of the code update request, the verifier $\mathcal{V}$ authorizes the creation of a new integrity key $K^{\mathsf{U}}_{\mathsf{int}}$, which is cryptographically bound not only to the trusted platform configuration of the system and the task, but also the code update request *req*. That way, the prover has to extend the hash of the code update request to the PCRs of the task in order to be able to load the newly created integrity key during the attestation procedure. If an adversary manipulates the code update request during transfer to the verifier, $\mathcal{V}$ creates an authorization value $Auth^{\mathsf{U}}_{\mathsf{pub}}$ for a modified/compromised request. As a result, the prover cannot load the key, because $\mathcal{P}$ extended a different hash and the attestation fails. As a consequence, the verifier does not provide the actual code update in the final step of the protocol. In contrast to the secure loading protocol presented in Chapter 7, this motivates the prover to cooperate and deters an adversary from attacking the attestation protocol.

8.5 Summary

In this chapter, we have presented a mechanism for attesting the trustworthiness of multiple microkernel tasks with their own cryptographic context inside a multi-context HSM and a protocol, which utilizes that mechanism for secure code updates and the recovery of compromised tasks. In the code update and recovery protocol, the verifier is able to select a subset of tasks, which constitute the trusted base system, and provide a update/recovery binary for a compromised task if the base system is still trustworthy according to the attestation result. As a result, the protocol provides a mechanism to solve the challenges of Scenario 3 (Secure Update and Recovery), which focuses on the need to enable updates and the recovery of compromised tasks. Incidentally, this mechanism can also be used to update the baseband stack on the baseband processor, even though it is not necessarily a microkernel-based OS.

Compared to most existing attestation schemes, which mostly rely on expensive cryptographic operations, we showed that our lightweight attestation mechanism can implicitly verify the integrity of multiple isolated microkernel tasks while eliminating the need for digital signatures. Furthermore, our lightweight attestation enables secure code updates and recovery of compromised tasks for microkernel-based systems, which are designed to be highly resilient against failures and attacks, particularly by implementing and separating tasks in user space.

In comparison to existing attestation protocols, in particular traditional remote attestation as specified by the TCG, we showed that our cryptographic integrity proof, basically a hash value, is more than ten times smaller than a digital signature, because our protocol mostly relies on symmetric cryptographic operations. When attesting more than one task with their own cryptographic contexts, the difference is significant, especially in environments where lightweight protocols are required. Without even considering SMLs, we can say that our cryptographic integrity proof only needs a constant size, whereas the number of digital signatures used in classical remote attestation increase with the number of tasks. If we also include SMLs, which store a list of all characteristics or objects that were measured and, thus, can be very large, the difference might be even more significant.

Finally, because of the benefits of separating the tasks in a microkernel-based system, we also proposed to separate the "monolithic" cryptographic context within most HSMs. As a result, multi-context HSM not only provides each task with its own isolated context, e.g., for cryptographic keys and integrity measurements, but also helps to improve the current level of native support for virtualization in general. By implementing separate platform configuration registers and anomaly detection records in each context, systems utilizing virtualization can transparently use the multi-context HSM for virtual machines, because the HSM supports virtualization natively by providing those multiple contexts. Since virtualization, especially hardware-assisted virtualization, and other hardware separation technologies, e.g., ARM TrustZone, are available for embedded systems, too, we will focus on an implicit attestation mechanism for such systems in the following Chapter 9.

Implicit Attestation of Microkernel Tasks for a Lightweight Update and Recovery

9

Policy-based Implicit Attestation and Data Integrity Protection

Based on the implicit/local attestation protocols presented for baseband stacks (Chapter 6) and *Nizza*-inspired microkernel systems with a TPM 1.2 (Chapter 7) or a multi-context HSM (Chapter 8), we explore the use of a TPM 2.0 and a TEE in this chapter. As a TPM 2.0 provides advanced authentication and authorization mechanisms (cf. Section 2.1.2.5, *Enhanced Authorization*), we focus on integrating those features into our implicit attestation in order to enable use cases in Scenario 4 (Secure Data Access). The main objective is a TPM 2.0-based implicit attestation to prove the trustworthiness of a data producer, which provides integrity-protected data to a verifier. In combination with a full, hardware-based TEE, our comprehensive system architecture further increases security in DomA through hardware isolation with a minimal interface and, thereby, enables a policy-based attestation mechanism, which benefits the verifier as well as the prover.

The rest of the chapter is structured as follows. In Section 9.1, we motivate the attestation scenario. In Section 9.2, we outline the details of our TEE-enhanced system architecture. We then present our main contribution, the integrity protection and policy-based attestation protocol, in Section 9.3, while reserving details about the prototype implementation for Section 9.4. Finally, we discuss the security of our protocol in Section 9.5 and conclude with a summary in Section 9.6.

Please note that the remote attestation mechanism presented in this chapter has already been presented at the *Information Security Conference (ISC)* in 2016 and is published in its peer-reviewed proceedings [Wag16a].

9.1 Policy-based Implicit Attestation for Secure Data Access

With hardware-based virtualization technologies, such as Intel VT [Nei06] or ARM's Virtualization Extensions [ARM12; ARM10], isolating rich operating systems like Linux from each other and the rest of the system is a very effective way to ensure overall system security. This level of security can be even further increased if a microkernel, such as *L4*/*Fiasco.OC* [Lie96; TUD11b], serves as the basis for an "unprivileged" hypervisor. Since microkernels implement all non-essential system components as user-space tasks, strictly separate those tasks, and have a very small code size, a microkernel-based hypervisor in user space reduces the system's attack surface significantly. As a result, such microkernel-based virtualized systems are suited even for the most security-critical applications. However, since the virtualized rich operating systems usually still require some degree of access to physical hardware, they cannot be completely isolated and a VMM must provide mechanisms to make selected hardware components available to the virtualized systems.

One mechanism to give a virtualized system access to a physical hardware component, such as a display or camera, directly maps the component's physical memory address to the address space of a virtualized system, which is then able to exclusively use this component. However, not all components can be directly assigned to a specific virtualized system, because some hardware components, such as the physical network interface, a mobile broadband modem, or sensors, are shared. In addition, this simple and naive mechanism does not allow for inspecting, dynamically restricting, and multiplexing the access to a component. That is why hardware components are usually virtualized, which means that access requests, i.e., read and write operations, have to go through the virtual machine monitor and to device drivers, which support virtualization.

On the other hand, virtualizing a hardware component, such as a sensor, also presents a number of challenges. For example, a virtualized system cannot be sure that the access to a component was handled as requested, because it is not able to directly access that component and make the request itself. Using *device emulation* techniques, a hypervisor could simulate a hardware component, particularly a sensor, and modify, for example, the result of a read operation before it is returned to the virtualized system. That is why a (remote) user or system, which interacts with the virtualized rich operating system, needs to be able to verify the integrity of the underlying system to be able to trust the data from a hardware component, such as a sensor.

Unfortunately, most existing mechanisms to (remotely) verify the integrity of a system, such as IBM's IMA [Sai04] in combination with a TPM, are not able to attest a microkernel-based system acting as hypervisor. In Chapters 7 and 8, we have shown protocols and designs for microkernel-based systems that enable a verification, but they focus on systems without hardware-assisted virtualization and without a hardware TEE, which is rooted in the SoC, such as ARM TrustZone. Since those technologies implement additional (higher) privilege level(s), they must be considered as well, because a microkernel alone might not be able to enforce separation on those systems.

To overcome these challenges and limitations, we first present an extended and enhanced version of our microkernel-based system architecture, which uses ARM's Virtualization Extensions to run multiple rich operating systems and a TPM 2.0 [Tru16] together with ARM TrustZone to securely handle critical operations and store sensitive information, like keys. While hardware-assisted virtualization primarily enables the execution of an unmodified rich OS, it also extends the application processor in DomA with an additional privilege level that can host the hypervisor. As a hypervisor, our system takes advantage of the microkernel design, which implements the virtual machine monitor in user space and only handles critical operations, such as mode transitions in and from the VMs. As hardware security module, our enhanced system architecture, which is described in the following section, utilizes the TPM 2.0, which enforces critical aspects of our implicit attestation protocol in cooperation with the ARM TrustZone, which provides the basis for our TEE.

Based on the TPM 2.0 and the TrustZone-based TEE, our main contribution presented in this chapter is a security protocol, which leverages our system design to protect the integrity of data as described in Scenario 4 (Secure Data Access) while implicitly verifying the trustworthiness of the system. Since the design is based on our previously described idea of efficient implicit attestation, the proposed protocol also uses symmetric cryptographic operations, such as hashing and HMACs, and only optionally utilizes asymmetric cryptographic operations during the setup phase. The details of this attestation mechanism and protocol are presented in the following sections. With regard to the scenario, we highlight the dual use of the attestation protocol to protect the access to data, e.g., generated by sensors on the prover's system, which is described as an industrial control system in Scenario 4. The main benefit for the scenario is the ability to verify the integrity of the data, which simultaneously enables the remote verifier to reason about the origin and trustworthiness of the data. This is especially important on systems, which use virtualization, because of the additional layer of indirection, which allows for device emulation as mentioned above.

Our third contribution, which is details in this chapter, is a proof-of-concept implementation of our enhanced microkernel-based system architecture and our proposed attestation protocol based on modern ARM-based hardware. Due to the lack of a dedicated hardware TPM 2.0 at the time of development, however, the prototype features a fully functional software simulator, which we have extracted from the public PDF version of the *TPM 2.0 Library Specification* [Tru14; Tru16] using a Python script [Wag16b] that we have made open source. The description of the proof-of-concept implementation, which focuses on complexity, code size, and information security (because we deal with a very small microkernel-based system with tightly controlled IPC), is complemented with a security discussion, which analyzes critical aspects of our protocol, e.g., the protection of keys and the prevention of various potential attacks against the protocol.

9.2 Microkernel-based System Architecture with TPM 2.0 and TEE

In this section, we describe the enhanced design of our microkernel-based system architecture, which specifically includes a TPM 2.0 and makes use of the hardware-based virtualization and security mechanisms of modern ARM SoCs. The complete architecture, the relevant software components for DomA, and the flow of executions (black arrows) are shown in Figure 9.1, in particular, the two microkernel-based systems.

The design of our TEE-enhanced system architecture separates a *secure* environment (right side) from a *non-secure* environment (left side) as shown in Figure 9.1. In contrast to the previous chapters, this classification into secure and non-secure is based on hardware separation mechanism, e.g., provided by the ARM TrustZone technology. Hence, the classification extends the previous notion of software-based trusted microkernel runtime in the MEE (in comparison to the untrusted rich OS in the REE) with a trusted microkernel runtime protected by hardware isolation mechanisms and executed in a TEE rooted in hardware. Both runtime environments, the secure and the non-secure environment, accommodate a microkernel-based system, which consists of at least a kernel component referred to as *core* in privileged levels `PL1+` and a user-space component *init* (which is similar to *Moe*/*Ned* in L4Re) in the unprivileged level `PL0`.

In detail, the DomA consists of a TEE with a microkernel-based system in the Secure World as well as several subdomains, Dom*0...n*, in the Non-secure World. As depicted in Figure 9.1, the MEE basically establishes a Dom*0*, whereas the rich operating systems of the REE are executed in subdomain 1 to n (Dom*1...n*). As a result, our DomA is divided into a hardware-based TEE on the one hand and several software-based subdomains in the Non-secure World on the other hand.

The separation into two isolated execution environments can be realized, for example, through the ARM TrustZone mechanism, which basically assigns system components, devices, and memory to either *Secure World* or *Non-secure World*. In the *Secure World*, the microkernel-based system or, more precisely, its TrustZone VMM (*tzvmm*), handles all request to security-critical tasks and devices, such as the TPM 2.0. In comparison to our Fiasco.OC-based system, the TPM server, which is referred to as *TS* in Chapter 7 and responsible for the TPM 1.2 communication, is called *tpm2* in this chapter and provides a native interface and device driver for the TPM 2.0.

In the *Non-secure World*, a second virtual machine monitor (*vmm*) similarly handles calls (and traps) to the hypervisor, in this case however, with the goal to virtualize rich operating systems like Linux. Since we do not restrict or otherwise limit the structure of the microkernel-based system in the MEE, it could basically be the same system as described in Chapter 7, i.e., Fiasco.OC with L4Re. As long as the underlying hardware supports a technology like ARM TrustZone, our concept can be applied to any monolithic or microkernel-based system running in the Normal word. However, without loss of generality, we will use the same microkernel-based systems for the Secure and the Non-secure World in our design and proof-of-concept implementation for the sake of simplicity.

The virtualized rich operating system in the REE, which can be a conventional Linux or Android, provides the usual services for a user to interact with the system. That means a (remote) user can access the rich operating system and, for example, read data from a hardware device, e.g., a sensor. If the hardware device is virtualized, access is trapped to the virtual machine monitor. The VMM, in turn, either forwards the request to a device driver, which also resides in the *Non-secure World*, or it uses a hardware-based interface, a so-called SMC, to access a device driver implemented in *Secure World* as depicted in Figure 9.1. If the request is handled by the virtual machine monitor in the TrustZone, *tzvmm* forwards it to the corresponding device driver that is also located in the TEE. That way, a security-critical device like a TPM 2.0 can be accessed and shared between multiple rich operating systems, while the VMM in the TEE is able to monitor, restrict, and deny access if necessary.

At this point it is very important to note that the minimal hardware-based TrustZone interface, the SMC, which is even less complex than the small set of system calls used by microkernels, is also the reason for having two microkernels in our system architecture. That way, there is a strong possibility that the system can still function and actively recover, even if the *Non-secure World* was compromised. Assuming the system architecture was implemented correctly, an adversary needs to successfully attack the *core* component in the *Secure World* (while using mainly SMC calls or traps) in order to fully compromise the system.

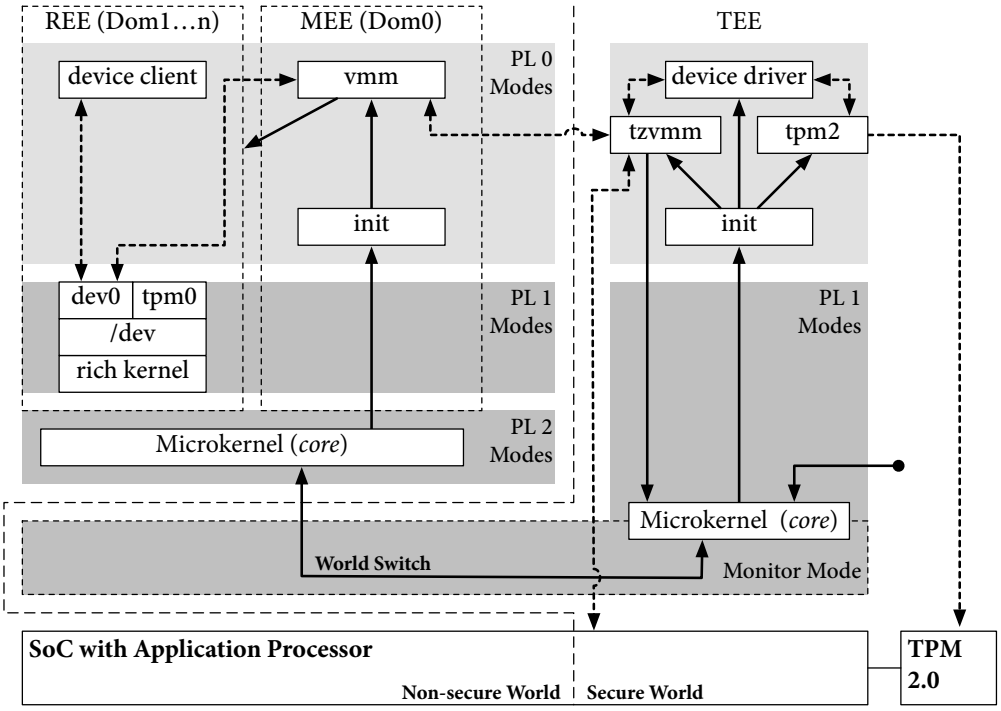

Figure 9.1: Microkernel-based System Architecture with TPM 2.0 on a TrustZone-enhanced ARM SoC

However, even with a separation of resources, such as memory or devices, though hardware-based virtualization and security mechanisms like TrustZone, the design of our system architecture also includes a TPM 2.0, which is connected to the TEE and used to securely create and store sensitive information, particularly cryptographic keys. As a reminder, a TPM in contrast to the ARM's TrustZone is a dedicated non-programmable hardware security module, which not only implements cryptographic engines in hardware, but also establishes trust, precisely because it provides assurances that its firmware cannot be easily modified by any user or remote attacker.

In addition to acting as a trust anchor, a TPM 2.0 provides mechanisms to store cryptographic integrity measurements as described in Chapter 2. Those measurements are usually collected during *authenticated boot*, extended into one of the PCRs (cf. Section 2.1.3.1), and digitally signed with an asymmetric key to create proof about the trustworthiness of the system for a remote attestation. Since the PCRs are only reset when the system is reset and can only be updated or, more precisely, extended with new measurements, an attacker is not able to modify a boot component without detection. On top of that, PCRs can also be used to cryptographically bind a key to specific values, which means that the key can only be used if the current values in the PCRs match the ones specified when the key has been created.

With the TPM 2.0, the TCG generalized this idea of "usage" policies and developed the concept referred to as Enhanced Authorization, which we described in detail in Section 2.1.2.5. With EA, the TPM 2.0 allows the use of a cryptographic key (or other protected assets, such as NV memory) if the user can recreate and, hence, satisfy a specified policy. That is why we use TPM 2.0 policies as the basis for our implicit attestation, because Scenario 4 (Secure Data Access) focuses on trustworthiness, integrity protection, and access control, which often relies on policies as well.

In summary, the main features of our most advanced system architecture are listed in Table 9.1. Based on the security features of our system architecture, i.e., the hardware-assisted virtualization, the TEE, and the TPM 2.0, the table shows the relevant architectural features and their locations.

Feature of the System Architecture	Location	Comment
Virtualized rich operating systems in isolated VMs	SoC → DomA → REE	
Microkernel	SoC → DomA → MEE	
VMM for VMs with the rich OS	SoC → DomA → MEE	user space
Microkernel	SoC → DomA → TEE	
VMM for the *Non-secure World*	SoC → DomA → TEE	user space
Secure device drivers (e.g., for sensors)	SoC → DomA → TEE	user space
TPM 2.0 driver / software stack	SoC → DomA → TEE	user space
Policy enforcement	TPM 2.0 → firmware	non-programmable
Longterm key storage (for the attestation)	TPM 2.0 → shielded locations	

Table 9.1: Division of the Main Architectural Features

9.3 Data Integrity Protection with Implicit Attestation

In this section, we present our data integrity protection and attestation protocol. The attestation mechanism enables the secure data access as described in our Scenario 4, where the verifier $\mathcal{V}$ requests integrity-protected data from an embedded system with multiple data generators and a network connection, such as an industrial control system, which acts as the prover $\mathcal{P}$.

In contrast to a traditional remote attestation as specified by the TCG, our protocol uses efficient symmetric operation instead of relying on expensive asymmetric cryptographic operations to create verifiable proof of the system's integrity. As the main contribution, however, our implicit attestation protocol makes use of the Enhanced Authorization mechanism provided by the TPM 2.0, which allows for a flexible definition of authorization and attestation policies.

The main idea of our proposed protocol is based on the fact that both parties, the prover $\mathcal{P}$ and the verifier $\mathcal{V}$, each control a cryptographic key with separate TPM 2.0 policies. The prover controls a key in its TPM storage hierarchy, which serves as a parent key for the verifier's key, which is initially created in the verifier's own TPM and migrated to the prover's TPM. For a successful policy-based implicit attestation, both parties have to cooperate to enable the prover to load both keys, which is ultimately necessary to protect data requested by the verifier.

In other words, $\mathcal{P}$ has to satisfy the verifier's policy in order to be able to load $\mathcal{V}$'s key, when $\mathcal{V}$ requests integrity-protected data. Since the verifier expects the data to be protected with its own integrity key and only accepts data after a successful verification, the prover has to be able to use $\mathcal{V}$'s key to protect the integrity of the requested data. $\mathcal{V}$'s policy, thus, may include trusted PCR values for the prover's system, which means the key can only be loaded if $\mathcal{P}$'s system is still in a trustworthy state. In turn, $\mathcal{V}$ must act according to the prover's policy, which convinces $\mathcal{P}$ to load the integrity key on behalf of $\mathcal{V}$ and create integrity-protected data. Hence, $\mathcal{P}$'s policy could, for example, specify that $\mathcal{V}$ may only access a virtualized hardware resource if the device is enabled and the access pattern meets certain criteria.

As a result, each policy must be satisfied by the other party for a successful implicit attestation when requesting integrity-protected data from a virtualized device. Effectively, the prover and the verifier have to cooperate to provide secure access to trusted resources, protect the integrity of data, e.g., generated by sensors, and be able to verify the trustworthiness of the data origin. Similar to most other hardware-based attestation protocols, the TPM acts as a trusted hardware component. In contrast, however, the TPM not only ensures the protection of integrity measurements and keys, but also enforces the policies set by the verifier as well as the prover.

In the following sections, we first define the notations and the cryptographic keys in Section 9.3.1 and Section 9.3.2. After that, we specify the setup phase in Section 9.3.3 and then focus on the integrity protection and attestation mechanism in Section 9.3.4.

9.3.1 Notation – Part 3 of 3

In this section, we extend the first two parts of our notation, which we already presented in Section 6.3.1 and Section 8.3.1, with a specific platform configuration for the TEE and policies.

Trusted Platform Configuration for the TEE

We define a trusted platform configuration P_{trusted} for our microkernel-based system, i.e., *core*, *init*, *tzvmm*, etc., in the TEE. As before, we assume that the load-time integrity of a microkernel-based system can be adequately described by a set of measurement values, which are securely stored in the PCRs of a TPM. For the sake of simplicity, we assume that the PCR values of our microkernel-based system in the TEE, which compose P_{trusted}, are public and, hence, known to a verifier in advance.

Policies

Similar to integrity measurements, a policy Pol is represented as a cryptographic hash, which can be used to authorize TPM operations. For example, to use a key K for signing, the initial empty policy hash Pol_0 is extended with the command code for signing (`TPM2_CC_Sign`), i.e., $Pol'_{\text{Sign}} = H(Pol_0 \parallel \texttt{TPM\_PolicyCommandCode} \parallel \texttt{TPM2\_CC\_Sign})$.

If the TPM can verify that the resulting policy hash Pol'_{Sign} matches the policy hash Pol_{Sign} assigned to a key K, this key can be used according to that specific policy. Since multiple policies can also be combined, e.g., with `TPM2_PolicyOR`, it is possible to create larger, more complex, yet flexible policies.

9.3.2 Cryptographic Keys

For our integrity protection and attestation protocol, we assume that a primary storage key K_{PSK}, which is created from a TPM Primary Seed, as well as a storage key K_{SK} already exist in the prover's TPM. As shown in Figure 9.2, we only require for the sake of simplicity that K_{SK} can be loaded with a policy, i.e., $Pol_{K_{\text{SK}}} = H(Pol_0 \parallel \texttt{TPM\_CC\_PolicyCommandCode} \parallel \texttt{TPM2\_CC\_Load})$.

In addition to the storage key K_{SK}, the prover needs a second key parent key K_{p}, which acts as a intermediate parent key for the integrity key K_{int}. The key allows for a policy-based import of child keys, which is encoded by specifying the command code `TPM2_Import` as shown in Figure 9.2. Like K_{SK}, this key is bound to the prover's TPM as well as its dedicated parent, which is enforced by the TPM though the key attributes `fixedTPM` and `fixedParent`. The second policy attached to K_{p} is used in our protocol to control access to hardware devices, such as sensors, by binding the content of NV memory areas to the load command. More precisely, if a bit in NV_{access} is set, the hardware component assigned to this bit is enabled, which is a requirement to access it. In $NV_{\text{granularity}}$, a minimum threshold, e.g., for the number of sensor values used in a average function or a anomaly detection score, can be stored. That way, the administrator of the prover's system can restrict access to a device if the number of access requests stored in $NV_{\text{granularity}}$ do not match the specified threshold of the policy or the pattern indicates malicious behavior.

Finally, we specify a keyed hash key K_{int}, which is used to symmetrically sign data with a HMAC to protect the integrity on behalf of the verifier. This key is initially created by the verifier's TPM and migrated to prover's system during the setup phase of our protocol. Consequently, the verifier alone is able to define the policies that have to be satisfied in order to be able to use the key as indicated in Figure 9.2. For our protocol, we at least require that the key allows for the calculation of a HMAC if the current PCRs values, which must include the microkernel-based system, i.e., *core*, *init*, *tzvmm*, etc., match the specified ones of a P_{trusted}. For the sake of simplicity, we also specify a policy-based authorization for a *duplication*, which is the TPM 2.0 term for migration, from the verifier's TPM to the prover's TPM. Since we encrypt K_{int} during the duplication, we optionally define a decryption key $K_{\text{dec}} = (pk_{\text{dec}}, sk_{\text{dec}})$. The public portion of this key, pk_{dec}, is used to encrypt the AES key K_{aes}, denoted as $\{K_{\text{aes}}\}_{pk_{\text{dec}}}$, while the secret key sk_{dec} is used by the TPM to decrypt $\{K_{\text{aes}}\}_{pk_{\text{dec}}}$ in order to obtain K_{aes}. However, please note that this encryption and decryption step is only required once during the setup phase.

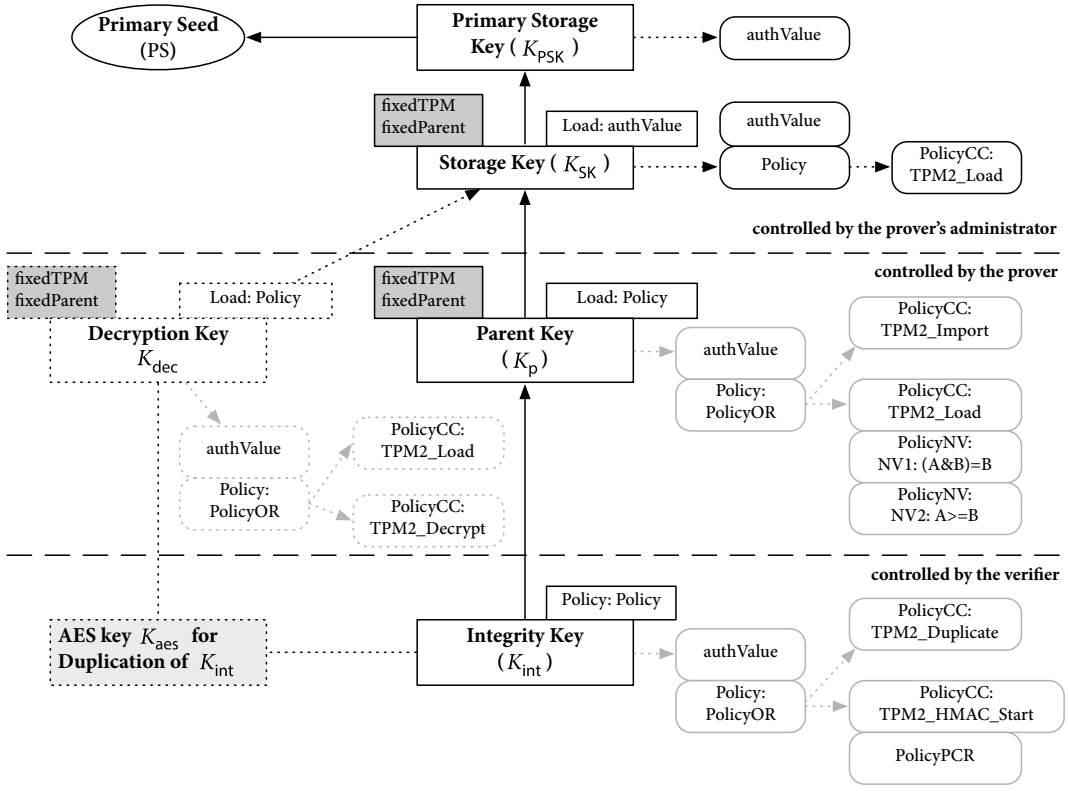

Figure 9.2: Cryptographic Key Hierarchy for our Policy-based Implicit Attestation Protocol

9.3.3 Phase 1: Setup

In the setup phase of our protocol, the administrator of the prover's system first creates two non-volatile memory areas, NV_{access} and $NV_{\text{granularity}}$, inside the TPM using `TPM2_NV_DefineSpace`. For our Scenario 4 (Secure Data Access), the first NV location is used to enable access to a device, whereas the second one allows for a fine-grained access control to data produced by the device(s). With the help of those NV locations, which can be cryptographically linked to scenario-specific policies, the TPM 2.0 can effectively act as a gatekeeper for devices and their data or, more precisely, for the keys used to protected the integrity of the data generated by those devices. As a consequence, both NV areas, which enable the access and usage control, can only be modified by the administrator of the prover's system, but can be read with a policy $Pol_{\text{NV\_Read}}$ and used with `TPM_CC_PolicyNV` by the TEE without manual authorization.

In NV_{access}, for example, the administrator of the prover's system can set certain bits to enable the corresponding hardware components. For instance, if bit 0 of NV_{access} is assigned to a device with an index 0, the administrator can set this bit to `1` in order to enable access to the device. Additionally, the administrator can use $NV_{\text{granularity}}$ in a policy to specify a minimum granularity or threshold. For example, if we assume the device is a sensor as described in our scenario and the verifier should only be able to read the average value of at least n sensor values, the current value in $NV_{\text{granularity}}$ could be compared to the reference value n specified in the policy.

$NV_{\text{granularity}}$ could also be used to store the result of an anomaly detection algorithm as described in Chapter 8, which needs to be below a certain threshold t defined in a policy to be able to load and use the key K_{int}. As a matter of fact, $NV_{\text{granularity}}$ implements exactly one of the ADRs as described in the previous chapter. The corresponding policy furthermore enables the specification of the threshold t, which was used to bind one integrity key in Chapter 8. To fully implement multiple ADRs as described in the previous chapter, the administrator of the prover's TPM only has to create multiple NV locations, i.e., $NV^{c}_{\text{granularity}}$, where c is the context or index of the tasks. For the sake of simplicity, however, we focus on only one of these NV locations.

After setting up the non-volatile memory locations, which are used as part of the policies for K_{p}, the storage key K_{SK} is loaded using the policy of K_{PSK}. Then, K_{p} is created with the policy described in the previous section, which is

$$Pol_{K_{\text{p}}} = H(Pol_{\text{Base}} \,||\, \texttt{TPM\_CC\_PolicyOR} \,||\, Pol_{\text{Import}} \,||\, Pol_{\text{Load\_NVs}}) \,, \tag{9.1}$$

where

$$\begin{aligned} Pol_{\text{Import}} &= H(Pol_0 \,||\, \texttt{TPM\_CC\_PolicyCommandCode} \,||\, \texttt{TPM\_CC\_Import}) \,, \\ Pol_{\text{Load\_NVs}} &= H(Pol_{\text{NVs}} \,||\, \texttt{TPM\_CC\_PolicyCommandCode} \,||\, \texttt{TPM\_CC\_Load}) \,, \end{aligned} \tag{9.2}$$

and Pol_{Base} is either Pol_{Import} or $Pol_{\text{Load\_NVs}}$.

The value Pol_{NVs}, in turn, is calculated based on the following equations, which use a cryptographic hash value generated during the initialization of the NV indices as the respective name of NV_{access} and $NV_{\text{granularity}}$:

$$Pol_{\text{NV\_Access}} = H(Pol_0 \,||\, \texttt{TPM\_CC\_PolicyNV} \,||\, args \,||\, \text{nvIndex}\rightarrow\text{Name}) \tag{9.3}$$

$$Pol_{\text{NVs}} = H(Pol_{\text{NV\_Access}} \,||\, \texttt{TPM\_CC\_PolicyNV} \,||\, args \,||\, \text{nvIndex}\rightarrow\text{Name}) \tag{9.4}$$

with

$$args = H(operandB.buffer \,||\, offset \,||\, operation)$$

where *operandB* is the value used for the comparison, *offset* is the start value of the NV data, and *operation* is the type of comparison. For NV_{access}, the operation is (A&B) = B, which checks that all bits in B are set in A, while the operation for $NV_{\text{granularity}}$ is A ≥ B, which enables the prover to specify a minimum value.

Once the policy Pol_{K_p} is successfully generated, the key K_p is created with `TPM2_Create`, which calculates a new ordinary key (cf. Section 2.1.2.3). For this command, a public template specifies the properties of the key to be generated by the TPM, e.g., the type of key and the associated policy. The command returns the public and encrypted private key as well as data about the creation, which can be certified.

When the intermediate parent key K_p was created, an asymmetric decryption key pair K_{dec} can be optionally generated using `TPM2_Create` as described in Section 9.3.2 while the storage key K_{SK} is still loaded. Like K_p, this key is also attached with a combined policy, which allows for loading and decryption:

$$Pol_{K_{\text{dec}}} = H(Pol_{\text{Base}} \,||\, \texttt{TPM\_CC\_PolicyOR} \,||\, Pol_{\text{Load}} \,||\, Pol_{\text{Decrypt}}) \,, \tag{9.5}$$

where

$$Pol_{\text{Load}} = H(Pol_0 \,||\, \texttt{TPM\_CC\_PolicyCommandCode} \,||\, \texttt{TPM\_CC\_Load}) \,,$$
$$Pol_{\text{Decrypt}} = H(Pol_0 \,||\, \texttt{TPM\_CC\_PolicyCommandCode} \,||\, \texttt{TPM\_CC\_Decrypt}) \,,$$

and Pol_{Base} is either Pol_{Load} or Pol_{Decrypt}.

As described in the previous section, this asymmetric key is only used to securely transfer an AES key, which is used to encrypt $\mathcal{V}$'s integrity key K_{int}, from the verifier to the prover. This optional step of transferring the AES key is only executed once and, hence, has no significant impact on our protocol. Alternatively, K_{int} can be pre-provisioned by the verifier, which can be handled in a secure environment, e.g., during production, and makes duplication process unnecessary. A secure duplication mechanism, however, enables updates of K_{int} in the field at a later time.

On the verifier's system, $\mathcal{V}$ generates the keyed hash key K_{int} as part of the setup process. This key is a symmetric signing key, which can be migrated to a new TPM (`fixedTPM` is `CLEAR`) and is cryptographically bound to the integrity measurements of the prover's microkernel-based system. As a result, it can only be loaded if the current values of the PCRs match the ones specified by the verifier. The policy for this key, which—for the sake of simplicity—allows for a duplication without a strong authentication, is calculated as

$$Pol_{K_{\text{int}}} = H(Pol_{\text{Base}} \,||\, \texttt{TPM\_CC\_PolicyOR} \,||\, Pol_{\text{Dup}} \,||\, Pol_{\text{PCR\_HMAC}}) \,, \tag{9.6}$$

with

$$Pol_{\text{Dup}} = H(Pol_0 \,||\, \texttt{TPM\_CC\_PolicyCommandCode} \,||\, \texttt{TPM\_CC\_Duplicate})$$

and Pol_{Base} is either Pol_{Dup} or $Pol_{\text{PCR\_HMAC}}$. In turn, $Pol_{\text{PCR\_HMAC}}$ is calculated by the TPM as

$$Pol_{\text{PCR}} = H(Pol_0 \,||\, \texttt{TPM\_CC\_PolicyPCR} \,||\, pcrs \,||\, digestTPM)$$
$$Pol_{\text{PCR\_HMAC}} = H(Pol_{\text{PCR}} \,||\, \texttt{TPM\_CC\_PolicyCC} \,||\, \texttt{TPM\_CC\_HMAC\_Start})$$

where *pcrs* is a structure specifying the bits corresponding to the PCRs and *digestTPM* is the digest of the selected PCRs provided by the verifier using a so-called *trial session*. This type of session allows for specifying the expected PCR values, whereas in a *non-trial session* the TPM would use the internal PCR values to calculate the digest.

Once the policy $Pol_{K_{\text{int}}}$ is successfully generated, the key K_{int} can be created using the command `TPM2_Create`. Again, the policy and key type (keyed hash key) can be specified in the public template, which is used by the TPM to create a key accordingly. To duplicate or migrate the keyed hash key K_{int} to the prover, the TPM cryptographically binds the key to its new parent key K_{p}, whose integrity and authenticity can be verified using the certified creation data produced by the prover's TPM, when K_{p} was created. For the actual duplication of K_{int} to the prover's system, the verifier runs `TPM2_Duplicate` with an optional AES encryption key and the public portion of the storage key K_{p} as input. Note that this implicitly also restricts the duplication to the prover's TPM, since the attribute `fixedTPM` is `SET` for K_{p}. The result of `TPM2_Duplicate` is an AES-encrypted key structure, which includes all necessary information to import the key into the target TPM. To complete the migration of K_{int}, the AES key K_{aes} is encrypted with the public portion of K_{dec} and sent to the prover together with the encrypted K_{int}.

On the prover's system, the AES key is decrypted using the private portion of K_{dec} and, in turn, used to decrypt K_{int} while it is imported to its new parent K_{p}. Please note that since all of those commands, `TPM2_Duplicate`, `TPM2_Decrypt`, and `TPM2_Import`, are part of the policy of their respective keys, this process does not require any interactive authorization by an administrator.

9.3.4 Phase 2: Data Integrity Protection with Implicit Attestation

In this section, we describe our data integrity protection and attestation protocol, which implicitly creates verifiable proof that $\mathcal{P}$'s system is still in a trustworthy state while protecting the integrity of data from a virtualized device for $\mathcal{V}$.

To read data from a device, such as a sensor, which is virtualized by the microkernel-based system, the verifier $\mathcal{V}$ first configures the device through a mechanism provided by the rich operating system, e.g., `ioctl`. This configuration includes the setting of the *granularity* g and a $nonce_{\mathcal{V}}$. However, since the prover $\mathcal{P}$ does not allow the kernel or device drivers of the rich OS to configure the device directly, the operation is trapped to the hypervisor in the *Non-secure World* as shown in the top half of Figure 9.3 above the dashed line. The hypervisor, in turn, evaluates the configuration—in particular, the value of the granularity g—which is used, for example, in an average function or simply to limit access to the hardware device. If the configuration is valid and matches the criteria set by the prover's administrator, the hypervisor forwards the request to the secure device driver implemented in the *Secure World*, which is able to configure the physical hardware device. In parallel, the VMM in TrustZone stores the granularity g in $NV_{\text{granularity}}$, which is a critical step for using K_{p} and part of a correct behavior of *tzvmm* that is assumed to be reflected in the PCRs.

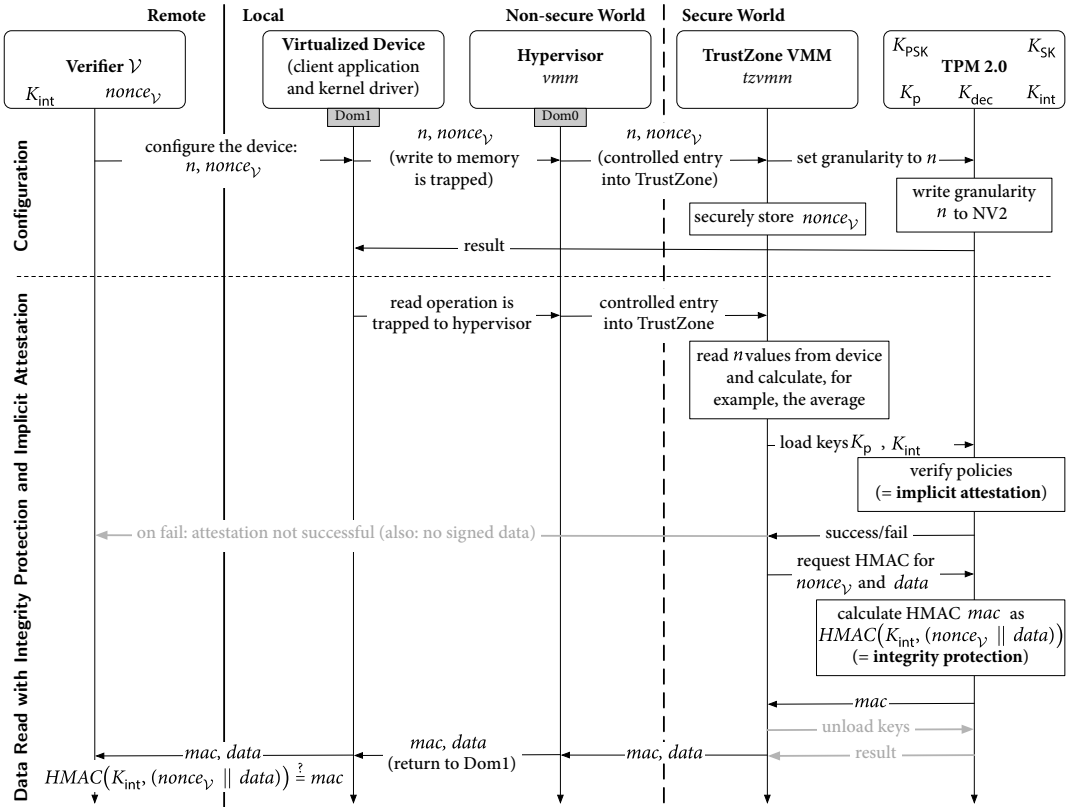

Figure 9.3: Data Access with Policy-based Implicit Attestation

After the configuration, the client application in the rich operating system can read data from the device, which is, again, trapped and forwarded to the secure device driver in the *Secure World*. For a sensor, the device driver in the *Secure World* then reads the necessary sensor values based on the granularity g and, for example, calculates average. To protect the integrity of the result, the TrustZone VMM then requests the TPM to calculate a HMAC *mac* as

$$mac = HMAC\big(K_{\text{int}}, (nonce_{\mathcal{V}} \,||\, data)\big) \,. \tag{9.7}$$

This is only possible if the keyed hash key K_{int} is loaded and the PCRs of the microkernel-based systems, i.e., *core*, *init*, *tzvmm*, *vmm*, etc., match the specified values. K_{int}, however, can only be loaded under the parent key K_{p}, if the device is enabled in NV_{access} and the granularity g stored in $NV_{\text{granularity}}$ is above the threshold specified by the prover's administrator in the policy $Pol_{\text{NV\_Access}}$, Pol_{NVs}, and $Pol_{K_{\text{p}}}$.

Consequently, the prover needs to re-create the policy $Pol_{K_{\text{p}}}$ (Equation 9.1) in a *policy session* inside the TPM to be able to load the key K_{int} on behalf of the verifier. More precisely, $\mathcal{P}$ has to calculate $Pol_{\text{Load\_NVs}}$ (cf. Equation 9.2), which is only possible if the values in the NV indices satisfy the respective policies that are calculated as follows:

$$Pol'_{\text{NV\_Access}} = H(Pol_0 \,||\, \texttt{TPM\_CC\_PolicyNV} \,||\, args \,||\, \text{nvIndex}\rightarrow\text{Name})$$

$$\text{with} \qquad args = H(operandB.buffer \,||\, offset \,||\, operation) \,,$$

where *operandB* is the value in NV_{access}, *offset* is the start value (0), and *operation* is the type of comparison, i.e., $(A\&B) = B$ for NV_{access} in our example. This specific policy verifies that the device is enabled and accessible, which is entirely controlled by the prover. If the comparison returns true, $Pol'_{\text{NV\_Access}}$ equals $Pol_{\text{NV\_Access}}$, which means the device is enabled. The policy Pol'_{NVs} can then be calculated as

$$Pol'_{\text{NVs}} = H(Pol'_{\text{NV\_Access}} \,||\, \texttt{TPM\_CC\_PolicyNV} \,||\, args \,||\, \text{nvIndex}\rightarrow\text{Name})$$

$$\text{with} \qquad args = H(operandB.buffer \,||\, offset \,||\, operation) \,,$$

where *operandB* is the value g in $NV_{\text{granularity}}$, *offset* is the start value (0), and *operation* is a comparison of $A \geq B$, that is $g \geq g_{Pol_{\text{NVs}}}$. This policy checks that the granularity g is above the value specified in the policy Pol_{NVs}. Again, if the comparison returns true, Pol'_{NVs} equals Pol_{NVs} and the device access pattern is accepted. Based on the freshly generated Pol'_{NVs}, the policy $Pol'_{\text{Load\_NVs}}$ can be calculated as

$$Pol'_{\text{Load\_NVs}} = H(Pol'_{\text{NVs}} \,||\, \texttt{TPM\_CC\_PolicyCommandCode} \,||\, \texttt{TPM\_CC\_Load}) \,.$$

If $Pol'_{\text{Load\_NVs}}$ equals $Pol_{\text{Load\_NVs}}$, this policy can satisfy the `OR`-policy Pol_{K_p} as specified in Equation 9.1, which enables the prover to load K_{int}. The prover only has to combine $Pol'_{\text{Load\_NVs}}$ with the pre-calculated value of Pol_{Import} using `TPM2_PolicyOR` to generate Pol_{K_p}. By specifying the session with the freshly generated policy Pol_{K_p}, which should be equal to Pol_{K_p}, the prover is able to load the key K_{int} on behalf of the verifier.

To use K_{int} to protect the integrity of the device data and implicitly verify the integrity of the system, the prover simply has to re-create the policy $Pol_{K_{\text{int}}}$. By creating a new policy session inside the TPM and using `TPM2_PolicyPCR`, the prover first creates Pol_{PCR} as

$$Pol'_{\text{PCR}} = H(Pol_0 \,||\, \texttt{TPM\_CC\_PolicyPCR} \,||\, pcrs \,||\, digestTPM) \; .$$

For this policy, the PCRs of the microkernel-based system, which are usually stored in one of the lower PCRs, must be specified. We assume that the prover and the verifier agree on the selection of PCRs, since both aim for a successful attestation. The policy Pol'_{PCR} is then used in `TPM2_PolicyCommandCode` to calculate $Pol'_{\text{PCR\_HMAC}}$, which is combined with the pre-calculated value Pol_{Dup} to generate $Pol'_{K_{\text{int}}}$:

$$Pol'_{\text{PCR\_HMAC}} = H(P'_{PCR} \,||\, \texttt{TPM\_CC\_PolicyCC} \,||\, \texttt{TPM\_CC\_HMAC\_Start})$$
$$Pol'_{K_{\text{int}}} = H(Pol'_{\text{PCR\_HMAC}} \,||\, \texttt{TPM\_CC\_PolicyOR} \,||\, Pol_{\text{Dup}} \,||\, Pol'_{\text{PCR\_HMAC}}) \; .$$

If $Pol'_{K_{\text{int}}}$ equals $Pol_{K_{\text{int}}}$, the policy can finally be used to create the HMAC *mac* over *data* and $nonce_{\mathcal{V}}$, which is used to prove freshness, as described in Equation 9.7.

The *data* and the HMAC *mac* are then returned to the rich operating system. The HMAC-protected *data* can then be transferred to the verifier's system, where a fresh HMAC *mac'* can be generated with the verifier's K_{int} as

$$mac' = HMAC\big(K_{\text{int}}, (nonce_{\mathcal{V}} \,||\, data)\big) \; . \tag{9.8}$$

If the freshly generated HMAC *mac'* matches the HMAC *mac* from the prover and $nonce_{\mathcal{V}}$ is the expected nonce, the verifier does not only know that the data has not been modified, but also that the prover's system is still trustworthy. More precisely, $\mathcal{V}$ is able to reason about the trustworthiness of the prover's system, because the $\mathcal{P}$ has been able to use the verifier's key to protect the data, which requires the prover to satisfy the policies specified by the verifier, especially the compliance with a trusted platform configuration.

9.4 Implementation

In this section, we present details about our proof-of-concept implementation, which we realized on an *Arndale* board. This development board, which is a so-called single-board computer, features an *Exynos 5250* SoC with a *Cortex-A15 MPCore* [ARM11] that includes both ARM's Virtualization and Security Extensions.

The main components of our prototype comprise a microkernel-based system in TrustZone, which includes a TPM 2.0 simulator (*tpm2sim*) as native microkernel tasks, and a similar system in the *Non-secure World*, which acts as a hypervisor and virtualizes rich operating systems.

As our microkernel-based system, we employ the bare-metal kernel of the Genode [Gen17] *base-hw* project, which combines Genode's *core* component with a small kernel library, only has a size of about 17 KLOC and supports TrustZone as well as virtualization. On top of the microkernel, we use regular Genode user-space components, such as *init*, which is about 3 KLOC, and the virtual machine monitors, *vmm* in the *Non-secure World* and *tzvmm* in the *Secure World*. For our protocol, we mainly adapted the trapping and the *World Switch* mechanism to be able to transfer data to and from the virtual machine monitors in *Non-secure World* and in TrustZone.

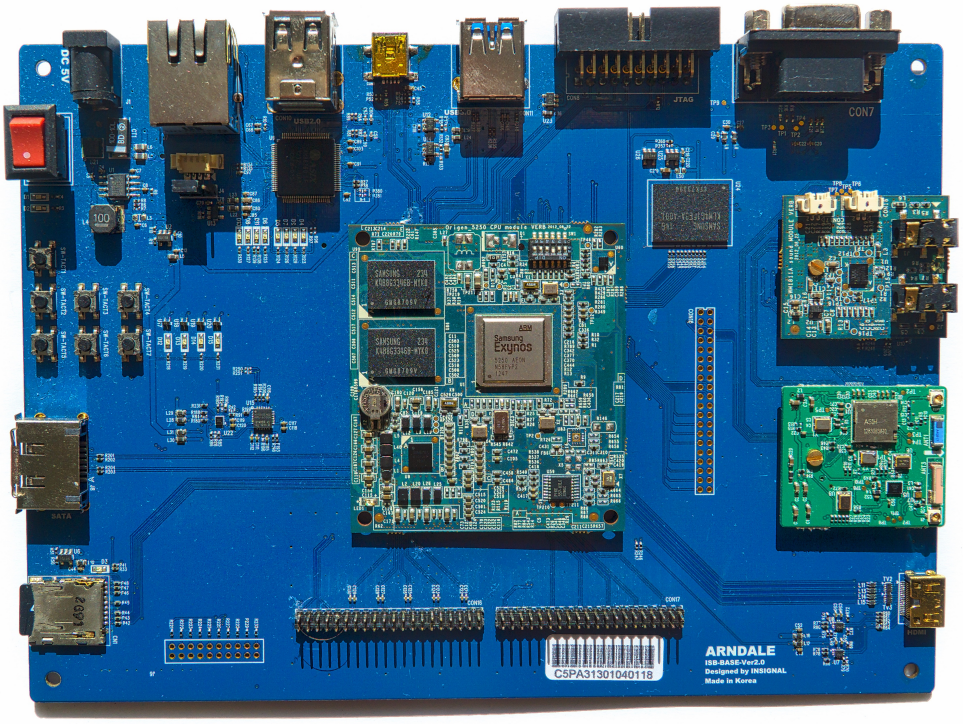

Figure 9.4: Arndale Board as Basis for our Proof-of-Concept Implementation

In the *Secure World*, we have extended the virtual machine monitor *tzvmm* to handle requests from the *Non-secure World* and also be able to execute the appropriate TPM commands. Since we did not have a hardware TPM 2.0 when we started the implementation, we created a Python script to extract a working simulator from the public PDF version of the TPM 2.0 Library Specification, which we have made open source [Wag16b]. By integrating the simulator into the Secure World, our TEE architecture is similar to the one discussed by ContainerX/Microsoft researchers [Raj16], who show an implementation of a firmware TPM on devices with ARM TrustZone and published their work at the *25th USENIX Security Symposium (USENIX Security 16)* around the same time. In our case, however, we had to port the code to Genode to be able to run the simulator as a native microkernel task, since the code of the TPM specification is primarily written for Windows. This included modifications to the random number generation, the NV memory subsystem, and the communication, which was socket-based and uses IPC with a shared memory area for the commands and responses in our Genode port.

For our rich OS, which is a conventional unmodified Linux 4.0 with a small *BusyBox* [And16], we created a device client application and a kernel module to implement the device driver for a hardware component. In our prototype, this device driver, which would normally configure and access that hardware component directly, is trapped to the hypervisor of the *Non-secure World*. To be able to transfer data from the rich operating system to the hypervisor, we added a memory trap to the hypervisor configuration and additionally implemented an `smc`-based *World Switch*, which uses shared memory locations to transfer data to and from the TrustZone.

To put our prototype implementation in perspective, the modifications to the existing Genode components, such as *core*, *tzvmm*, or *vmm*, only amount to a few hundred lines of code per component as shown in Table 9.2. The reason for that is the fact that most of the protocol is handled by the TPM 2.0 simulator, which we extracted from the specification and has about 40 KLOC, and the VMMs in the *Secure* and *Non-secure World*. The rest of the system uses mechanisms, such as IPC and shared memory, which are part of the microkernel-based system provided by Genode.

Component	Original Size	Difference		Total
core (Secure World)	17572	+215	+1,2%	17787
tzvmm	651	+956	+146,9%	1607
tpm2sim	0	+40469	+100,0%	40469
↳ *tpm2sim_server*	0	+305		305
↳ *tpm2sim_libplatform.lib.so*	0	+448		448
↳ *tpm2sim_libCryptoEngine.lib.so*	0	+5501		5501
↳ *tpm2sim_libTPM.lib.so*	0	+22258		22258
↳ *include* and *tpm/include*	0	+11957		11957
core (Non-Secure World)	17572	+154	+0,9%	17726
vmm	1132	+698	+61,7%	1830

Table 9.2: Code size of relevant native components (calculated with cloc [Dan16])

9.5 Informal Security Analysis

In this security analysis, we discuss the key security aspects of our protocol. Since our proposed protocol combines data access and integrity protection with an implicit attestation, we first focus on the integrity of the data, which is transmitted from the prover to the verifier. After that, we discuss the security of our policy-based implicit attestation mechanism in detail.

9.5.1 Security of the Data Access and Integrity Protection

In our protocol, the access to the devices is protected by the prover's TEE and the TPM, which provides the NV index NV_{access} to enable access to specific devices. Without setting the bit in NV_{access}, the corresponding device is considered disabled and the TPM prevents the prover from loading the verifier's integrity key K_{int}. In addition, $NV_{\text{granularity}}$ prevents access that is too detailed.

To protect the integrity, the prover's TPM calculates a message authentication code over the data, e.g., from a sensor, using the shared HMAC key K_{int}. This key, which is created and controlled by the verifier, is encrypted and migrated from the verifier's TPM to the prover's TPM during the setup phase and can only be used inside those respective TPMs. That way, an attacker is not able to easily forge a correct HMAC for data with unauthorized modification, because it is not able to intercept the HMAC key, decrypt it, and use it in an arbitrary TPM. As a result, the attacker cannot modify the data without detection, since the calculation of a valid HMAC requires access to the integrity keys and the attacker is not able to create a convincible forgery without that key according to our attacker model presented in Section 4.2.

In addition, please note that the identity of the prover is implicitly included in the HMAC if the verifier creates a distinct key for each prover. More precisely, since the HMAC key, which has been created for a particular prover, is duplicated specifying the public key of that prover's K_{p}, the HMAC key is cryptographically bound to the identity of that prover and its TPM. Similarly, the AES encryption key, which is used during duplication, is also cryptographically bound to the prover's K_{dec}, which is fixed to the $\mathcal{P}$'s TPM and, thus, cannot be migrated to an arbitrary TPM.

Furthermore, our protocol includes a nonce for freshness, which has to be checked by the verifier to make sure that the HMAC has been generated for the most recent request. For an adversary, the verification of the nonce eliminates the possibility to replay old data, which has been protected with a correct HMAC, but for data that is potentially no longer valid. This is particularly relevant for devices, such as a heartbeat sensor, where the verifier must be able to detect a replay attack, where an attacker (or even the prover) might try to convince the verifier that the system still functions without any downtime or anomalies. Another example of such devices are fire detectors, which most of the time create very similar data that often include the current temperature, or rotation sensors, which track the current speed of centrifuges or turbines. If an attacker was able to replay old data, a compromised system might be able to pretend that is functions as specified, which enables the attacker to hide from detection.

9.5.2 Security of the Policy-based Implicit Attestation Mechanism

For a successful policy-based implicit attestation, the prover must create a valid HMAC for the data produced by a hardware device and a nonce provided by the verifier. To create a correct HMAC, the prover needs to load the HMAC key K_{int} into the TPM 2.0, which verifies the attached policies and implicitly creates verifiable cryptographic proof for the trustworthiness of the prover's system. As a consequence, the HMAC not only enables the verifier to evaluate the integrity of the data and detect unauthorized modifications, but also to reason about the prover's system state, because the verifier can trust in the fact that the TPM enforces the specified policies, in particular for K_{int}.

Control of the Integrity Key K_{int} used for the Implicit Attestation

In our attestation protocol, the verifier creates and controls the integrity key K_{int}, which means the verifier is the only one that is able to define the policies, which have to be satisfied by the prover. That allows the verifier to specify, for example, the exact PCR values, which we assume reflect a known/trusted platform configuration, particularly the microkernel-based systems in *Secure World*. Since the authentication value $Auth_{int}$ for the key K_{int} is only known to the verifier, which controls the key and its policies, the prover (or an attacker) cannot change the policy at a later point in time. However, the prover is able to define policies for the parent key K_p, which enables the prover to restrict access to certain data sources as described in the previous section. Furthermore, the policies for K_p can indirectly affect the capability to load the integrity key K_{int}, which is by design and enables the prover to limit the access to devices and extract integrity protected data using policies enforced by the TPM 2.0 and the microkernel-based system in the hardware TEE.

Evaluation of the Policies for the Attestation

During the attestation, the prover's policies are evaluated first. If the policies cannot be satisfied, the verifier does not get access to a device, which effectively allows the prover to limit access to devices. However, if the prover's policies can be met, the policies defined by the verifier, which include at least a trusted set of PCR values, are evaluated before the HMAC for the device data is calculated. At this point, it is important to note that the policies are not verified by the operating system, as it is usually the case in policy-based authorization schemes. In our protocol, the policies are instead verified by the TPM, which also moves the point of enforcement inside the TPM. Consequently, a successful attestation is only possible if the TPM ensure that policies are satisfied, which means, for example, that the prover's system is in a trustworthy state as reflected by the PCRs. If the prover or any attacker has modified the system, the final policy of the verifier cannot be met and the prover is not able to load the HMAC key K_{int} on behalf of the verifier. As a result, the prover cannot protect the integrity of the data and the attestation eventually fails, because the verifier does not receive a fresh and valid HMAC.

9.6 Summary

In this chapter, we extended our implicit attestation mechanism, which mainly relies on symmetric cryptographic operations rather than digital signatures, and presented a policy-based implicit attestation for microkernel-based systems, which feature virtualization, a TPM 2.0, and a TEE. By combining the TPM 2.0 with a microkernel-based system in the TEE, our implicit attestation mechanism allows for the integration and verification of policies that are enforced by the TPM. In addition, since the prover and verifier have to cooperate for our proposed attestation protocol, our mechanism enables a secure data access scenario, where the prover (as the data provider) can specify its own usage policy for data sources, such as sensors.

In detail, our policy-based approach enables the verifier to create a key, which is used for integrity protection, and cryptographically bind a policy, which specifies the characteristics of a trustworthy system, to that key. For a successful attestation, the prover is expected to use that key to protect the integrity of the requested data from a virtualized hardware component, such as a sensor. Consequently, the verifier can implicitly evaluate the trustworthiness of the prover's system, whenever it accesses a virtualized device and the requested data is protected with the key, which has been bound to an attestation policy.

As a result, our approach enables the verifier to not only ensure that the data requested from a virtualized device has not been modified, but also to implicitly verify the integrity and trustworthiness of the prover's system. In addition, our attestation mechanism enables the prover to specify policies, which have to be satisfied by the verifier. Consequently, the prover and the verifier have to cooperate, which allows for the enforcement of access control rules by the prover as well as the remote attestation constraints by the verifier. In combination, this mechanism enables secure data access as described in Scenario 4 (Secure Data Access).

10

Conclusion

Based on existing research, concepts, and technologies, such as Trusted Computing, which aim to improve security, we explored in this thesis if there exist lightweight mechanisms to attest the trustworthiness of resource-constraint embedded devices, which do not rely on digital signatures and extensive measurement logs. In contrast to traditional remote attestation as specified by the Trusted Computing Group, we researched and developed a novel *implicit attestation mechanism* that mainly uses efficient symmetric cryptographic operations instead of asymmetric cryptography. As a result, our lightweight attestation method relies on hash-based message authentication codes, which effectively embody the cryptographic evidence for the trustworthiness of a remote system and ideally reduce the size of the attestation result by an order of magnitude.

To explore possible applications, we focused our research on the question how to integrate our implicit attestation mechanism into existing authentication protocols and improve the security of typical *secure access* scenarios. First, we discussed the remote attestation of mobile baseband stacks, which are an interesting attack target, because they are usually privileged software components running on the baseband processor of a mobile device and interact via the baseband hardware with the mobile network. In this *secure network access* scenario, we showed that our implicit attestation mechanism can be used to attest the trustworthiness of the baseband stack towards the USIM, which can integrate the attestation result into the authentication protocol of the mobile network. As a result, our research provides a lightweight mechanism for attesting a mobile baseband stack, which enables the mobile network to grant access based on the attestation result and to protect the network against coordinated attacks, such as DDoS attacks, by compromised mobile devices.

By shifting the focus of our research to the application processor domain, we then explored the possible use of the implicit attestation mechanism in our microkernel-based system architecture, which evolved during the course of this thesis. For our *secure loading* scenario, we started by adopting the main ideas and concepts of IMA, which is only available for Linux, to a microkernel. Microkernels are less complex than monolithic kernels, hence reduce the TCB, and strictly separate microkernel tasks, such as the integrity measurement components, from the rest of the system. For the microkernel-based system, we proposed a secure loading protocol, which integrates a version of our implicit attestation and enables local attestation through an integrity challenge protocol.

Based to those results, we further discussed the *secure update and recovery* of compromised microkernel tasks based our implicit attestation mechanism. More precisely, we show how our mechanism can be used to attest multiple separated microkernel tasks with their own cryptographic context in a multi-context HSM. By providing separate cryptographic contexts, which can store integrity measurements and events generated by an anomaly detection component, for each microkernel task, we demonstrate the benefits of our hash-based implicit attestation mechanisms in comparison to traditional remote attestation, which relies on digital signatures.

Finally, for the *secure data access* scenario, we extended the design of our system architecture with a hardware TEE. In the resulting architecture, we were able to integrate our implicit attestation mechanism into a second trusted microkernel-based system in the TEE equipped with a TPM 2.0. As a result, we demonstrated that our implicit attestation mechanism can benefit the verifier as well as the prover, because both parties can specify policies which are enforced the TPM 2.0.

Conclusions Drawn

In short, the following conclusions can be drawn from the research conducted for this thesis:

- Remote attestation as specified by the TCG is not always suitable for resource-constraint embedded devices, because it relies on digital signatures and extensive measurement logs.
- With a TPM, symmetric cryptography can be used to implicitly attest the trustworthiness of the prover's system towards a verifier.
- As a lightweight attestation mechanism, implicit attestation can ideally reduce the size of cryptographic proof by an order of magnitude in comparison to TCG-based attestation.
- Implicit attestation can be used to attest mobile baseband stacks running on the baseband processor towards the USIM and the mobile network by integrating the attestation result into the authentication protocol.
- Since microkernels are less complex, reduce the TCB, strictly separate integrity verification components, and can also acts as very small hypervisors, e.g., for rich operating systems, microkernel-based systems are ideally suited for implicit attestation, in particular because of their stable configuration.

Outlook and Future Research Possibilities

As embedded systems, especially mobile and IoT devices, become ubiquitous and continue to gain new hardware and software features, the complexity and code base of such systems are likely to grow, which in turn increases their attack surface and, consequently, the probability of a successful attack. Therefore, it is going to be critical and imperative to more effectively harness existing security concepts, components, and mechanisms, such as a resilient system architecture based on a robust microkernel, a security module like a TPM, and cryptographic protocols like remote attestation. Furthermore, there is a tremendous research opportunity to more intrinsically combine those strict security technologies and "unambiguous" cryptographic mechanisms with more "fuzzy" concepts like *machine learning*, especially *anomaly detection*.

One of the many possible research questions could be how to handle classification errors by an imperfect model and translate the fuzzy results of anomaly detection algorithms to conclusive cryptographic protocols. In terms of security, future research might also focus on the question how a model of an anomaly detection component can be protected against potential attacks by using a hardware security module. While there exists a wide range of related work on protecting the accuracy and validity of a model, security researchers could explore hardware-based mechanisms to verify the integrity of a model, which might then in turn be reported using remote attestation protocols.

Conclusion

Notation

$\mathcal{A}$	The adversary $\mathcal{A}$	63
$\boldsymbol{A}$	The aggregate A of n hash values	99
$\boldsymbol{A_c}$	The anomaly detection record of task/*context c*	121
AMF	The value of the *Authentication Management Field*	84
ATT	The attestation value *ATT* calculated by the USIM	84
ATTB	The attestation value *ATTB* for the baseband and the baseband info	84
Auth	A generic authentication value	
↳ $\boldsymbol{Auth_{aik}}$	The authentication value for the attestation identity key K_{aik}	93
↳ $\boldsymbol{Auth_{bind}}$	The authentication value for the binding key K_{bind}	93
↳ $\boldsymbol{Auth_{int}}$	The authentication value for the integrity key K_{int}	153
↳ $\boldsymbol{Auth_{migration}}$	The authentication value $Auth_{migration}$	126
↳ $\boldsymbol{Auth_{parent}}$	The authentication value for the parent wrapping key K_{wrap}	82
↳ $\boldsymbol{Auth^{U}_{pub}}$	The authentication value $Auth^{U}_{pub}$	126
↳ $\boldsymbol{Auth_{res}}$	The authentication value $Auth_{res}$	83
↳ $\boldsymbol{Auth^{U}_{res}}$	The authentication value $Auth^{U}_{res}$	127
↳ $\boldsymbol{Auth_{seal}}$	The authentication value for the sealing key K_{seal}	79
↳ $\boldsymbol{Auth_{usage}}$	The authentication value $Auth_{usage}$	126
↳ $\boldsymbol{Auth_{wrap}}$	The authentication value for the wrapping key K_{wrap}	79
AUTN	The authentication token *AUTN* (part of the AV)	84
B	The baseband binary *B*	71
BI	The baseband information *BI*	71
bin	A binary *bin*	93
BL	The boot loader *BL*	79
BS	The backend system *BS*	93
BV	The baseband vendor *BV*	71
C	The challenger *C*	96

c	A context c	121
$Cert$	A generic certificate	
↳ $Cert_{\mathsf{BS}}$	The certificate $Cert_{BS} = \{BS, pk^{BS}_{\mathsf{cert}}\}_{sk^{\mathsf{CA}}_{\mathsf{sig}}}$	93
	where BS is the name, $pk^{\mathsf{BS}}_{\mathsf{cert}}$ is BS's public key and $sk^{\mathsf{CA}}_{\mathsf{sig}}$ is CA's private key	
↳ $Cert_{\mathsf{CA}}$	The root certificate $Cert_{\mathsf{CA}} = \{CA, pk^{\mathsf{CA}}_{\mathsf{cert}}\}_{sk^{\mathsf{CA}}_{\mathsf{sig}}}$	93
	where CA is the name, $pk^{\mathsf{CA}}_{\mathsf{cert}}$ is CA's public key and $sk^{\mathsf{CA}}_{\mathsf{sig}}$ is CA's private key	
↳ $Cert_{\mathsf{AIK}}$	The certificate $Cert_{\mathsf{AIK}} = \{AIK, pk^{\mathsf{AIK}}_{\mathsf{cert}}\}_{sk^{\mathsf{CA}}_{\mathsf{sig}}}$	93
	where TPM is the name, $pk^{\mathsf{AIK}}_{\mathsf{cert}}$ is TPM's public AIK key pk_{aik} and $sk^{\mathsf{CA}}_{\mathsf{sig}}$ is CA's private key	
$\|\|$	concatenation	78
d	A digest d	77
DomA	The Application Processor Domain *DomA* in our system architecture	65
DomB	The Baseband Processor Domain *DomB* in our system architecture	65
e	An anomaly detection event $e = (m, p)$	121
$ebin$	An encrypted binary $ebin$	95
$\{\cdot\}_{pk}$	The generic notation of the result of encrypting data with a public key pk	142
g	The granularity g, e.g., of measurements from a sensor	147
H	A hash function H	77
h	A hash h	77
↳ h_B	The hash value of the *baseband binary B*	84
↳ h_c	A hash value that includes the wrapped integrity key $\{K^c_{\mathsf{int}}\}^{P_c,t_c}_{pk_{\mathsf{wrap}}}$ of tasks c	122
↳ h_{bin}	The hash value of the *binary (bin)*	95
HMAC	A generic hash-based message authentication code function denoted as $HMAC(K, m)$	165

i	An index i	78
ID_U	An identity for the software update U	126
Io	The L4Re input/output server *Io*	38
ipad	The inner padding *ipad* in an HMAC calculation	78
IS	The Integrity Server *IS*	95
K	A generic cryptographic key K	77
↳ K_{ASME}	the UMTS key K_{ASME}	84
↳ K_{asym}	An asymmetric cryptographic key K_{asym}	
	where pk is the public portion and	
	sk is the private portion of the asymmetric key pair	
↳ K_{aik}	The attestation identity key $K_{\mathsf{aik}} = (pk_{\mathsf{aik}}, sk_{\mathsf{aik}})$	93
	where pk_{aik} is the public portion and	
	sk_{aik} is the private portion of the asymmetric key pair	
↳ K_{att}	The attestation key $K_{\mathsf{att}} = (pk_{\mathsf{att}}, sk_{\mathsf{att}})$	79
	where pk_{att} is the public portion and	
	sk_{att} is the private portion of the asymmetric key pair	
↳ K_{bind}	The binding key $K_{\mathsf{bind}} = (pk_{\mathsf{bind}}, sk_{\mathsf{bind}})$	93
	where pk_{bind} is the public portion and	
	sk_{bind} is the private portion of the asymmetric key pair	
↳ $K^{\mathsf{BS}}_{\mathsf{bind}}$	The wrapping key $K^{\mathsf{BS}}_{\mathsf{sig}} = (pk^{\mathsf{BS}}_{\mathsf{sig}}, sk^{\mathsf{BS}}_{\mathsf{sig}})$	93
	where pk^{BS}_{sig} is the public portion and	
	sk^{BS}_{sig} is the private portion of the asymmetric key pair	
↳ K_{dec}	The decryption key $K_{\mathsf{dec}} = (pk_{\mathsf{dec}}, sk_{\mathsf{dec}})$	142
	where pk_{dec} is the public portion and	
	sk_{dec} is the private portion of the asymmetric key pair	
↳ K_{int}	The integrity key $K^{[s]}_{\mathsf{int}} = (pk^{[s]}_{\mathsf{int}}, sk^{[s]}_{\mathsf{int}})$	79
	where pk_{int} is the public portion,	
	sk_{int} is the private portion of the asymmetric key pair, and	
	s optionally specifies the context c	
	or the code update U	
↳ K_{p}	The parent key K_{p}	142
↳ K_{PSK}	The primary storage key K_{PSK}	142
↳ K_{seal}	The sealing key $K_{\mathsf{seal}} = (pk_{\mathsf{seal}}, sk_{\mathsf{seal}})$	79
	where pk_{seal} is the public portion and	
	sk_{seal} is the private portion of the asymmetric key pair	

↳ K_{SK}	The storage key K_{SK} 142
↳ K_{wrap}	The wrapping key $K_{wrap} = (pk_{wrap}, sk_{wrap})$ 79
	where pk_{wrap} is the public portion and
	sk_{wrap} is the private portion of the asymmetric key pair
↳ K_{sym}	A symmetric cryptographic key K_{sym}
↳ AK	A UMTS anonymity key 84
↳ CK	A UMTS confidentiality key 84
↳ IK	A UMTS integrity key 84
↳ K_{aes}	The symmetric AES key used during key duplication 142
↳ K_{enc}	The symmetric encryption key K_{enc} 95
↳ K_i	The UMTS pre-shared secret key K_i 71
k	A integer greater than 0 78
keyInfo	*keyInfo* defines the cryptographic properties of the new key 126
L	The Loader L 95
l	The length l 77
l4re	The L4Re runtime *l4re* 107
l4re_kernel	The L4Re runtime binary *l4re_kernel* 92
LC	The Loader Client *LC* 95
LS	The Loader Server *LS* 95
m	A generic message m 77
MAC	A generic function to calculate a message authentication code denoted as
	$MAC(K, m)$ 166
ML	The measurement list *ML* 95
Moe	The L4Re root task *Moe* 38
μ	The integrity measurement μ 78
n	A positive integer 78
Ned	The L4Re init task *Ned* 38
NV_{access}	The NV index inside the TPM 2.0 used for storing access to a hardware
	component 142
$NV_{granularity}$	The NV index inside the TPM 2.0 used for storing a granularity ... 142
opad	The outer padding *opad* in an HMAC calculation 78

$\mathcal{P}$	The prover $\mathcal{P}$ 60
p	An anomaly detection probability p 121
P	The platform configuration P 78
↳ P'	The current platform configuration P' 78
↳ P_{B}	The platform configuration for the *baseband binary B* 79
↳ P_{BL}	The platform configuration for the *boot loader BL* 79
↳ P_{c}	The platform configuration for a *context c* 121
↳ P_{MEE}	The platform configuration P_{MEE}, which comprises Fiasco.OC, *Sigma0*, *Moe*, *l4re*, *Ned*, *TS*, and *IS* 93
↳ P'_{MEE}	The current platform configuration P'_{MEE}, which comprises Fiasco.OC, *Sigma0*, *Moe*, *l4re*, *Ned*, *TS*, and *IS* 96
↳ P_{trusted}	The trusted platform configuration P_{trusted} 142
Pol	A policy Pol ..... 142 where Pol' is the freshly calculated version of this policy
↳ Pol_0	The initial empty policy Pol_0 142
↳ Pol_{Base}	The base policy Pol_{Base} for a complex OR-policy 144
↳ Pol_{Decrypt}	The policy Pol_{Decrypt} 145
↳ Pol_{Dup}	The policy Pol_{Dup} 146
↳ Pol_{Import}	The policy Pol_{Import} 144
↳ $Pol_{K_{\mathsf{dec}}}$	The policy $Pol_{K_{\mathsf{dec}}}$ 145
↳ $Pol_{K_{\mathsf{int}}}$	The policy $Pol_{K_{\mathsf{int}}}$, ... 146
↳ $Pol_{K_{\mathsf{p}}}$	The policy $Pol_{K_{\mathsf{p}}}$ 144
↳ $Pol_{K_{\mathsf{SK}}}$	The load policy $Pol_{K_{\mathsf{SK}}}$ for the storage key K_{SK} 142
↳ Pol_{Load}	The load policy Pol_{Load} 145
↳ $Pol_{\mathsf{Load\_NVs}}$	The policy for loading a key depending on certain NV values 144
↳ $Pol_{\mathsf{NV\_Access}}$	The policy $Pol_{\mathsf{NV\_Access}}$ 145
↳ $Pol_{\mathsf{NV\_Read}}$	The policy $Pol_{\mathsf{NV\_Read}}$ 143
↳ Pol_{NVs}	The policy Pol_{NVs} 144
↳ Pol_{PCR}	The policy Pol_{PCR} 146
↳ $Pol_{\mathsf{PCR\_HMAC}}$	The policy $Pol_{\mathsf{PCR\_HMAC}}$ 146
↳ Pol_{Sign}	The policy Pol_{Sign} 142
R	The Rich OS R 94
$RAND$	The random number $RAND$ of the AKA protocol 83
rc	The return code rc 124
req_U	An request req_U for update U 126

req	An attestation request *req*	122
RES	The result value *RES* of the AKA protocol	85
res	An attestation response *res*	124
sig	The signature (*sig*) of a baseband update	79
Sigma0	The L4Re root pager *Sigma0*	38
SQN	The sequence number *SQN* of the AKA protocol	84
t	A threshold value *t*	144
$\hookrightarrow t_c$	The threshold t_c for a *context c*	121
TA	The Trusted Application *TA*	95
TS	The TPM Server *TS*	95
U	The code update U	126
$\mathcal{V}$	The verifier $\mathcal{V}$	60
WL	The whitelist *WL*	99
$\{\cdot\}^{P}_{pk}$	The generic notation of the result of wrapping (or sealing) data with a public key *pk* to a *platform configuration P*	78
XATT	The expected attestation value *XATT*	84
$\oplus$	eXclusive OR	78
XRES	The expected result value *XRES* of the AKA protocol	84

Acronyms

AD	anomaly detection	50
ADR	anomaly detection record	119
AES	Advanced Encryption Standard *see also Glossary:* AES	14
AIK	Attestation Identity Key	18
AKA	Authentication and Key Agreement	75
API	Application Programming Interface *see also Glossary:* API	37
AR	access requestor	30
ASID	Address Space Identifier	43
ASME	Access Security Management Entity	84
AuC	Authentication Center	75
AV	authentication vector	83
BEE	Baseband Execution Environment	66
BIOS	Basic Input/Output System	16
BMBF	Bundesministerium für Bildung und Forschung (Federal Ministry of Education and Research)	1
CA	certificate authority	54
CFI	control flow integrity	50
CN	Core Network	75
CPU	central processing unit	13
CRTM	Core Root of Trust for Measurement	16
DAA	Direct Anonymous Attestation	54
DDoS	distributed denial-of-service	73

DRM	digital rights management	12
DSP	digital signal processor	71
EA	Enhanced Authorization	21
ECC	elliptic curve cryptography	15
ECDH	Elliptic Curve Diffie-Hellman	171
ECU	electronic control unit	1
EEPROM	electrically erasable programmable read-only memory	13
EFI	Extensible Firmware Interface	25
EH	Endorsement Hierarchy	19
EK	Endorsement Key	16
EPS	Endorsement Primary Seed	19
EVM	Extended Verification Module	27
FPGA	Field Programmable Gate Array	31
GIC	Generic Interrupt Controller	43
GUID	Globally Unique Identifier	25
HCR	Hyp Configuration Register	43
HLR	Home Location Register	75
HMAC	hash-based message authentication code *see also Glossary:* HMAC	5
HN	Home Network	75
HSM	hardware security module	3
HSS	Home Subscriber Server	75
I/O	input/output	13
I2C	Inter-Integrated Circuit	100
IEC	International Electrotechnical Commission	13
IMA	Integrity Measurement Architecture	3
IMC	Integrity Measurement Collector	30
IMEI	International Mobile Equipment Identity	71
IMSI	International Mobile Subscriber Identity	71
IMV	Integrity Measurement Verifier	30
IoT	Internet of Things	1

IPC inter-process communication 34
IRQ interrupt request 102
ISO International Organization for Standardization 13

JCOP Java Card OpenPlatform 33
JVM Java Virtual Machine 41

KDF key derivation function 84
KVM Kernel-based Virtual Machine 44

L4Re L4 Runtime Environment 38
LIM Linux Integrity Module 26
LKM Linux Kernel Module 27
LOC lines of code 108
LPC Low Pin Count 128
LSM Linux Security Module 26

MAC message authentication code 63
see also Glossary: MAC
MAP metadata access point 30
MAPC MAP client 30
MBR Master Boot Record 25
ME Mobile Equipment 75
MEE Microkernel Execution Environment 66
MME Mobility Management Entity 75
MMU Memory Management Unit 34
MSC Mobile Switching Center 75
MTM Mobile Trusted Module 73

NAR network access authority 30
NAR network access requestor 30
NIST National Institute of Standards and Technology 171
NS non-secure 47
NV non-volatile 14

OC object-capability model 37

OIAP	Object Independent Authorization Protocol	18
OS	operating system	2
OSAP	Object-Specific Authorization Protocol	18
PC	personal computer	1
PCR	Platform Configuration Register	9
PDP	policy decision point	30
PEP	policy enforcement point	30
PH	Platform Hierarchy	19
POSIX	Portable Operating System Interface	118
PPS	Platform Primary Seed	19
PRF	pseudo-random function	171
PS	Primary Seed	19
PTS	platform trust service	30
RAM	random access memory	13
RAN	Radio Access Network	75
REE	Rich Execution Environment	66
RF	radio frequency	71
RNC	Radio Network Controller	75
RNG	random number generator	14
ro	read-only	105
ROM	read-only memory	13
ROP	return-oriented programming	86
RPC	remote procedure calls	39
RSA	Rivest-Shamir-Adleman cryptosystem	14
RTM	Root of Trust for Measurement	16
RTR	Root of Trust for Reporting	16
RTS	Root of Trust for Storage	16
rw	read-write	105
SCR	Secure Configuration Register	47
SD	Secure Digital	63
SE	secure element	4
SELinux	Security-Enhanced Linux	34
SGX	Software Guard Extensions	46

SH	Storage Hierarchy	19
SHA-1	Secure Hash Algorithm 1	14
	see also Glossary: SHA-1	
SHA-2	Secure Hash Algorithm 2	15
	see also Glossary: SHA-2	
SIM	subscriber identity module	13
SLAT	second-stage address translation	44
SMC	Secure Monitor Call	46
SML	stored measurement log	29
SMS	short message service	76
SN	Service Network	75
SoC	system on chip	43
SPS	Storage Primary Seed	19
SRK	Storage Root Key	17
TC	Trusted Computing	4
TCB	trusted computing base	4
TCG	Trusted Computing Group	4
TCP	Transmission Control Protocol	91
TCPA	Trusted Computing Platform Alliance	12
TEE	Trusted Execution Environment	4
TI	Texas Instruments	100
TIMA	TrustZone-based Integrity Measurement Architecture	50
TLB	translation lookaside buffer	43
TMSI	Temporary Mobile Subscriber Identity	71
TNC	Trusted Network Connect	29
TNCC	TNC client	30
TNCS	TNC server	30
TOCTOU	time of check to time of use	29
TPM	Trusted Platform Module	viii
TRNG	true random number generator	119
TSS	TCG Software Stack	12
TTBR	Translation Table Base Register	43
UDP	User Datagram Protocol	91
UE	User Equipment	75

UEFI	Unified Extensible Firmware Interface	16
USB	Universal Serial Bus	70
USIM	Universal Subscriber Identity Module	65
VLR	Visitor Location Register	75
VM	virtual machine	4
VMI	virtual machine introspection	50
VMM	virtual machine monitor	41
VPN	virtual private network	61
VT	Virtualization Technology	41
VTCR	Virtualization Translation Control Register	43
vTPM	virtual TPM	56
VTTBR	Virtualization Translation Table Base Register	43

Glossary

AES AES, the Advanced Encryption Standard, is a specification for the cryptographic encryption of data established by the National Institute of Standards and Technology (NIST) in 2001. 165

API An Application Programming Interface is a specified set of functions that a software program can use to access services and resources provided by another software that implements that API 37

Fiasco.OC *Fiasco.OC* is a 3rd-generation μ-kernel (microkernel) that implements the object-capability (OC) model 31

Genode *Genode* is a novel OS architecture that is able to master complexity by applying a strict organizational structure to all software components including device drivers, system services, and applications. The *Genode* OS *framework* is the effort to advance the *Genode* OS architecture as a community-driven open-source project [Gen17] 37

HYP The additional privileged CPU mode provided by ARM's Virtualization Extensions. 43

KDFa The TPM 2.0 uses a hash-based function, *KDFa()*, to generate keys for multiple purposes. With the exception of Elliptic Curve Diffie-Hellman (ECDH), *KDFa()* is used in all cases where a KDF is required. *KDFa()* uses Counter mode from *SP800-108* [Che09], with HMAC as the pseudo-random function (PRF) [Tru16, Part 1, Section 11.4.9]. 19

KLOC Unit denoting *thousands of lines of code* – A metric used to measure the size of a computer program by counting the number of lines in the source code of a software program 107

L4Android The (para-)virtualized Android for L4/Fiasco.OC. 107

L4Linux The (para-)virtualized Linux for L4/Fiasco.OC. 42

MLOC Unit denoting *millions of lines of code* – A metric used to measure the size of a computer program by counting the number of lines in the source code of a software program 107

ProVerif *ProVerif* is an automatic cryptographic protocol verifier, in the formal model (so called Dolev-Yao model) v

SHA-1 SHA-1 is a cryptographic hash function published in 1993 by the NIST under the title *Secure Hash Standard* (FIPS PUB 180) 14

SHA-2 SHA-2 is a cryptographic hash function published in 2001 by the NIST under the title *Secure Hash Standard* (FIPS PUB 180-4) 15

Bibliography

[3rd12a] 3rd Generation Partnership Project (3GPP): *TS 23.002, Network Architecture. Technical Specification*. 2012 (cit. on p. 75).

[3rd12b] 3rd Generation Partnership Project (3GPP): *TS 33.102, 3G security; Security architecture. Technical Specification*. 2012 (cit. on p. 84).

[3rd12c] 3rd Generation Partnership Project (3GPP): *TS 33.401, System Architecture Evolution (SAE); Security architecture. Technical Specification*. 2012 (cit. on pp. 75, 76, 84).

[Aba05] Abadi, Martín, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti: "Control-flow integrity." *Proceedings of the 12th ACM Conference on Computer and Communications Security, CCS 2005, Alexandria, VA, USA, November 7-11, 2005*. Ed. by Atluri, Vijay, Catherine A. Meadows, and Ari Juels. ACM, 2005: pp. 340–353 (cit. on p. 52).

[Als10] Alsouri, Sami, Özgür Dagdelen, and Stefan Katzenbeisser: "Group-based Attestation: Enhancing Privacy and Management in Remote Attestation." *Proceedings of the 3rd International Conference on Trust and Trustworthy Computing*. TRUST'10. Berlin, Germany: Springer-Verlag, 2010: pp. 63–77 (cit. on p. 54).

[And16] Andersen, Erik, Rob Landley, Denys Vlasenko, et al.: *BusyBox*. 2016. url: `https://busybox.net` (visited on 11/01/2017) (cit. on p. 151).

[App07] Apple Inc.: *iOS*. 2007. url: `https://www.apple.com/ios/ios-11/` (visited on 11/01/2017) (cit. on p. 40).

[App96] Apple Inc.: *XNU*. 1996. url: `https://opensource.apple.com/source/xnu/` (visited on 11/01/2017) (cit. on p. 40).

[ARM12] ARM Ltd.: *ARM Architecture Reference Manual. ARMv7-A and ARMv7-R edition. ARM DDI 0406C.b*. July 2012 (cit. on pp. 11, 41, 43, 136).

[ARM11] ARM Ltd.: *ARM Cortex-A15 Technical Reference Manual. ARM DDI 0438C*. Sept. 2011 (cit. on p. 150).

[ARM09] ARM Ltd.: *ARM Security Technology - Building a Secure System using TrustZone Technology*. PRD29-GENC-009492C. ARM Ltd. Apr. 2009 (cit. on p. 11).

[ARM17] ARM Ltd.: *mbed TLS*. Formerly known as PolarSSL. 2017. URL: `https://tls.mbed.org` (visited on 11/01/2017) (cit. on p. 102).

[ARM10] ARM Ltd.: *Virtualization Extensions Architecture Specification*. `http://infocenter.arm.com`. 2010 (cit. on pp. 11, 41, 43, 136).

[Art15] Arthur, Will and David Challener: *A Practical Guide to TPM 2.0: Using the Trusted Platform Module in the New Age of Security*. 1st. Berkely, CA, USA: Apress, 2015 (cit. on pp. 17, 19, 24, 25).

[Bai11] Bailey, Samuel A., Don Felton, Virginie Galindo, Franz Hauswirth, Janne Hirvimies, Milas Fokle, Fredric Morenius, Christophe Colas, and Jean-Philippe Galvan: *The Trusted Execution Environment: Delivering Enhanced Security at a Lower Cost to the Mobile Market*. Tech. rep. GlobalPlatform Inc., 2011 (cit. on pp. 46, 92).

[Bar06] Barr, M. and A. Massa: *Programming Embedded Systems: With C and GNU Development Tools*. O'Reilly Media, 2006 (cit. on p. 2).

[Bel96] Bellare, Mihir, Ran Canetti, and Hugo Krawczyk: "Message Authentication using Hash Functions—The HMAC Construction." *CryptoBytes* (1996), vol. 2 (cit. on p. 78).

[Ben11] Bente, Ingo, Gabi Dreo, Bastian Hellmann, Stephan Heuser, Joerg Vieweg, Josef Helden, and Johannes Westhuis: "Towards Permission-Based Attestation for the Android Platform." *Trust and Trustworthy Computing*. Ed. by McCune, Jonathan M., Boris Balacheff, Adrian Perrig, Ahmad-Reza Sadeghi, Angela Sasse, and Yolanta Beres. Vol. 6740. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2011: pp. 108–115 (cit. on p. 55).

[Ber06] Berger, Stefan, Ramón Cáceres, Kenneth A. Goldman, Ronald Perez, Reiner Sailer, and Leendert van Doorn: "vTPM: Virtualizing the Trusted Platform Module." *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15*. USENIX-SS'06. Vancouver, B.C., Canada: USENIX Association, 2006 (cit. on pp. 56, 116).

[Bla01] Blanchet, Bruno: "An Efficient Cryptographic Protocol Verifier Based on Prolog Rules." *Proceedings of the 14th IEEE workshop on Computer Security Foundations*. CSFW '01. Washington, DC, USA: IEEE Computer Society, 2001 (cit. on p. 130).

[Bri04] Brickell, Ernie, Jan Camenisch, and Liqun Chen: "Direct Anonymous Attestation." *Proceedings of the 11th ACM Conference on Computer and Communications Security*. CCS '04. Washington DC, USA: ACM, 2004: pp. 132–145 (cit. on p. 54).

[Bul13] Bulygin, Yuriy: *Evil Maid Just Got Angrier: Why Full-Disk Encryption With TPM is Insecure on Many Systems*. Mar. 2013 (cit. on pp. 24, 25).

[Bun06] Bundesministerium für Bildung und Forschung (BMBF, Federal Ministry of Education and Research): *The High-Tech Strategy for Germany*. 2006 (cit. on p. 1).

[Che09] Chen, Lidong: *SP 800-108. Recommendation for Key Derivation Using Pseudorandom Functions (Revised)*. Tech. rep. Gaithersburg, MD, United States: National Institute of Standards & Technology, 2009 (cit. on p. 171).

[Che06] Chen, Liqun, Rainer Landfermann, Hans Löhr, Markus Rohe, Ahmad-Reza Sadeghi, and Christian Stüble: "A Protocol for Property-Based Attestation." *Proceedings of the First ACM Workshop on Scalable Trusted Computing*. STC '06. ACM, 2006: pp. 7–16 (cit. on p. 54).

[Che08] Chen, Xiaoxin, Tal Garfinkel, E. Christopher Lewis, Pratap Subrahmanyam, Carl A. Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan R.K. Ports: "Overshadow: A Virtualization-based Approach to Retrofitting Protection in Commodity Operating Systems." *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*. 2008 (cit. on p. 56).

[Com15] Computer History Museum: *Timeline of Computer History*. 2015. URL: `http://www.computerhistory.org/timeline/computers/` (visited on 11/01/2017) (cit. on p. 1).

[Cuz15] Cuzzocrea, Alfredo, Enzo Mumolo, and Riccardo Cecolin: "Runtime Anomaly Detection in Embedded Systems by Binary Tracing and Hidden Markov Models." *39th IEEE Annual Computer Software and Applications Conference, COMPSAC 2015, Taichung, Taiwan, July 1-5, 2015. Volume 2*. Ed. by Ahamed, Sheikh Iqbal, Carl K. Chang, William C. Chu, Ivica Crnkovic, Pao-Ann Hsiung, Gang Huang, and Jingwei Yang. IEEE Computer Society, 2015: pp. 15–22 (cit. on p. 52).

[Dan16] Danial, Al: *CLOC – Count Lines of Code*. 2016. URL: `https://github.com/AlDanial/cloc` (visited on 11/01/2017) (cit. on pp. 107, 151).

[Def85] Defense, Department of: *Trusted Computer System Evaluation Criteria (Orange Book), DoD 5200.28-STD*. In the glossary under entry Trusted Computing Base (TCB). 1985 (cit. on p. 4).

[Doc13] Docker, Inc.: *Docker*. 2013. URL: `https://www.docker.com` (visited on 11/01/2017) (cit. on p. 34).

[Dol83] Dolev, D. and A. Yao: "On the security of public key protocols." *IEEE Transactions on Information Theory* (Mar. 1983), vol. 29(2): pp. 198–208 (cit. on p. 63).

[Eck14] Eckert, Claudia: *IT-Sicherheit: Konzepte - Verfahren - Protokolle*. De Gruyter Oldenbourg. De Gruyter, 2014 (cit. on pp. 63, 78).

[Eng08] England, Paul and Jork Löser: "Para-Virtualized TPM Sharing." *Trusted Computing - Challenges and Applications, Trust 2008*. Ed. by Lipp, Peter, Ahmad-Reza Sadeghi, and Klaus-Michael Koch. Vol. 4968. Lecture Notes in Computer Science. Villach, Austria: Springer, Mar. 2008: pp. 119–132 (cit. on p. 116).

[Fel11] Feller, Thomas, Sunil Malipatlolla, Michael Kasper, and Sorin A. Huss: "dcTPM: A Generic Architecture for Dynamic Context Management." *2011 International Conference on Reconfigurable Computing and FPGAs, ReConFig 2011*. Ed. by Athanas, Peter M., Jürgen Becker, and René Cumplido. Cancun, Mexico: IEEE Computer Society, Nov. 2011: pp. 211–216 (cit. on p. 116).

[Fes06] Feske, Norman: *Introduction into TUD:OS*. 2006. URL: `http://demo.tudos.org/intro_tutorial.html` (visited on 11/01/2017) (cit. on p. 35).

[For10] Forsberg, Dan, Günther Horn, Wolf-Dietrich Moeller, and Valtteri Niemi: *LTE Security*. Wiley, 2010 (cit. on pp. 75, 84).

[Gan03] Ganssle, J.G. and M. Barr: *Embedded Systems Dictionary*. R and D Developer Series. Taylor & Francis, 2003 (cit. on pp. 1, 2).

[Gar03] Garfinkel, Tal and Mendel Rosenblum: "A Virtual Machine Introspection Based Architecture for Intrusion Detection." *Proceedings of the Network and Distributed System Security Symposium, NDSS 2003, San Diego, California, USA*. The Internet Society, 2003 (cit. on p. 51).

[Gar17] Gartner, Inc.: *Gartner Says 8.4 Billion Connected "Things" Will Be in Use in 2017, Up 31 Percent From 2016*. Press Release. 2017. URL: `http://www.gartner.com/newsroom/id/3598917` (visited on 11/01/2017) (cit. on p. 1).

[Gen17] Genode Labs: *Genode*. 2017. URL: `http://www.genode.org` (visited on 11/01/2017) (cit. on pp. 40, 150, 171).

[Glo11] GlobalPlatform Inc.: *TEE System Architecture – Public Release v1.0*. GlobalPlatform Inc. Dec. 2011 (cit. on pp. 46, 92).

[gru10] grugq: *Base Jumping: Attacking the GSM baseband and base station*. COSEINC, 2010 (cit. on p. 76).

[Gur16] Guri, Mordechai, Yisroel Mirsky, and Yuval Elovici: *9-1-1 DDoS: Threat, Analysis and Mitigation*. 2016. URL: `http://arxiv.org/abs/1609.02353` (visited on 11/01/2017) (cit. on p. 73).

[Hal09] Halderman, J. Alex, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten: "Lest we remember: cold-boot attacks on encryption keys." *Commun. ACM* (May 2009), vol. 52(5): pp. 91–98 (cit. on p. 128).

[Här05] Härtig, Hermann, Michael Hohmuth, Norman Feske, Christian Helmuth, Adam Lackorzynski, Frank Mehnert, and Michael Peter: "The Nizza Secure-System Architecture." *International Conference on Collaborative Computing: Networking, Applications and Worksharing* (2005), vol. (cit. on pp. 56, 67).

[Här97] Härtig, Hermann, Michael Hohmuth, Jochen Liedtke, Jean Wolter, and Sebastian Schönberg: "The Performance of μ-kernel-based Systems." *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*. SOSP '97. Saint Malo, France: ACM, 1997: pp. 66–77 (cit. on pp. 42, 91).

[Hea02] Heath, S.: *Embedded Systems Design*. Elsevier Science, 2002 (cit. on p. 2).

[Her64] Herstein, Israel Nathan: *Topics in algebra*. A Blaisdell book in the pure and applied sciences. New York, Toronto, London: Blaisdell, 1964 (cit. on p. 77).

[Hof13] Hofmann, Owen S., Sangman Kim, Alan M. Dunn, Michael Z. Lee, and Emmett Witchel: "InkTag: Secure Applications on an Untrusted Operating System." *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*. 2013 (cit. on p. 56).

[Hor15] Horsch, Julian and Sascha Wessel: "Transparent Page-Based Kernel and User Space Execution Tracing from a Custom Minimal ARM Hypervisor." *2015 IEEE TrustCom/BigDataSE/ISPA, Helsinki, Finland, August 20-22, 2015, Volume 1*. IEEE, 2015: pp. 408–417 (cit. on p. 52).

[Hor14] Horsch, Julian, Sascha Wessel, Frederic Stumpf, and Claudia Eckert: "SobrTrA: a software-based trust anchor for ARM cortex application processors." *Fourth ACM Conference on Data and Application Security and Privacy, CODASPY'14, San Antonio, TX, USA - March 03 - 05, 2014*. Ed. by Bertino, Elisa, Ravi S. Sandhu, and Jaehong Park. ACM, 2014: pp. 273–280 (cit. on p. 53).

[Hub16] Huber, Manuel, Julian Horsch, Michael Velten, Michael Weiss, and Sascha Wessel: "A Secure Architecture for Operating System-Level Virtualization on Mobile Devices." *Information Security and Cryptology: 11th International Conference, Inscrypt 2015, Beijing, China, November 1-3, 2015, Revised Selected Papers*. Ed. by Lin, Dongdai, XiaoFeng Wang, and Moti Yung. Cham: Springer International Publishing, 2016: pp. 430–450 (cit. on p. 57).

[ISO09] ISO/IEC JTC 1: *Information technology – Trusted Platform Module*. ISO/IEC 11889:2009. ISO/IEC 11889:2009. International Organization for Standardization. Geneva, Switzerland, 2009 (cit. on p. 13).

[ISO15] ISO/IEC JTC 1: *Information Technology – Trusted Platform Module Library*. ISO/IEC 11889:2015. ISO/IEC 11889:2015. International Organization for Standardization. Geneva, Switzerland, 2015 (cit. on p. 13).

[Jae06] Jaeger, Trent, Reiner Sailer, and Umesh Shankar: "PRIMA: policy-reduced integrity measurement architecture." *Proceedings of the eleventh ACM symposium on Access control models and technologies*. SACMAT '06. Lake Tahoe, California, USA: ACM, 2006: pp. 19–28 (cit. on p. 50).

[Ker83] Kerckhoffs, Auguste: "La cryptographie militaire." *Journal des sciences militaires* (Jan. 1883), vol. IX: pp. 5–83 (cit. on p. 63).

[Kra07] Krauss, Christoph, Frederic Stumpf, and Claudia Eckert: "Detecting Node Compromise in Hybrid Wireless Sensor Networks Using Attestation Techniques." *Security and Privacy in Ad-hoc and Sensor Networks, 4th European Workshop, ESAS 2007*. Ed. by Stajano, Frank, Catherine Meadows, Srdjan Capkun, and Tyler Moore. Vol. 4572. Lecture Notes in Computer Science. Heidelberg: Springer, 2007: pp. 203–217 (cit. on p. 81).

[Kra98] Krawczyk, Hugo and Tal Rabin: *Chameleon Hashing and Signatures*. 1998. url: `https://eprint.iacr.org/1998/010.ps.gz` (visited on 11/01/2017) (cit. on p. 54).

[Küh07] Kühn, Ulrich, Marcel Selhorst, and Christian Stüble: "Realizing property-based attestation and sealing with commonly available hard- and software." *Proceedings of the 2nd ACM Workshop on Scalable Trusted Computing, STC 2007, Alexandria, VA, USA, November 2, 2007*. Ed. by Ning, Peng, Vijay Atluri, Shouhuai Xu, and Moti Yung. ACM, 2007: pp. 50–57 (cit. on p. 54).

[Kun06] Kuntze, Nicolai and Andreas U. Schmidt: *Trusted Computing in Mobile Action*. 2006. url: `https://arxiv.org/pdf/cs/0606045.pdf` (visited on 11/01/2017) (cit. on p. 55).

[Lac09] Lackorzynski, Adam and Alexander Warg: "Taming Subsystems: Capabilities As Universal Resource Access Control in L4." *Proceedings of the Second Workshop on Isolation and Integration in Embedded Systems*. IIES '09. Nuremburg, Germany: ACM, 2009: pp. 25–30 (cit. on pp. 91, 92).

[Lam91] Lampson, Butler, Martín Abadi, Michael Burrows, and Edward Wobber: "Authentication in Distributed Systems: Theory and Practice." *SIGOPS Oper. Syst. Rev.* (Sept. 1991), vol. 25(5): pp. 165–182 (cit. on p. 4).

[Lan11] Lange, Matthias, Steffen Liebergeld, Adam Lackorzynski, Alexander Warg, and Michael Peter: "L4Android: A Generic Operating System Framework for Secure Smartphones." *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*. SPSM '11. Chicago, Illinois, USA: ACM, 2011: pp. 39–50 (cit. on p. 56).

[Len14] Lengyel, Tamas K., Thomas Kittel, Jonas Pfoh, and Claudia Eckert: "Multitiered Security Architecture for ARM via the Virtualization and Security Extensions." *25th International Workshop on Database and Expert Systems Applications, DEXA 2014, Munich, Germany, September 1-5, 2014*. IEEE, 2014: pp. 308–312 (cit. on p. 65).

[Len15] Lengyel, Tamas, Thomas Kittel, and Claudia Eckert: "Virtual Machine Introspection with Xen on ARM." *2nd Workshop on Security in highly connected IT systems (SHCIS)*. Sept. 2015 (cit. on p. 51).

[Li10] Li, Yanlin, Jonathan M. McCune, and Adrian Perrig: "SBAP: Software-Based Attestation for Peripherals." *Trust and Trustworthy Computing, Third International Conference, TRUST 2010, Berlin, Germany, June 21-23, 2010. Proceedings*. Vol. 6101. Lecture Notes in Computer Science. Springer, 2010: pp. 16–29 (cit. on p. 53).

[Lie95] Liedtke, J.: "On Micro-kernel Construction." *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*. SOSP '95. Copper Mountain, Colorado, USA: ACM, 1995: pp. 237–250 (cit. on pp. 36, 40).

[Lie93] Liedtke, Jochen: "Improving IPC by Kernel Design." *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*. SOSP '93. Asheville, North Carolina, USA: ACM, 1993: pp. 175–188 (cit. on p. 40).

[Lie96] Liedtke, Jochen: "Microkernels Must And Can Be Small." *Proceedings of the 5th IEEE International Workshop on Object-Orientation in Operating Systems (IWOOOS)*. Oct. 1996 (cit. on pp. 116, 136).

[Lor12] Lorenz, Markus: "TPM-based Secure Boot for embedded Hypervisors." Diploma Thesis. Technische Universität München, June 2012 (cit. on p. 102).

[LXC08] LXC: *LXC (Linux Containers)*. 2008. URL: `https://linuxcontainers.org` (visited on 11/01/2017) (cit. on p. 34).

[Mar13] Martinelli, Fabio, Ilaria Matteucci, Andrea Saracino, and Daniele Sgandurra: "Remote Policy Enforcement for Trusted Application Execution in Mobile Environments." *Trusted Systems*. Ed. by Bloem, Roderick and Peter Lipp. Vol. 8292. Lecture Notes in Computer Science. Springer International Publishing, 2013: pp. 70–84 (cit. on p. 55).

[Mar17] Marwedel, P.: *Embedded System Design: Embedded Systems, Foundations of Cyber-Physical Systems, and the Internet of Things*. Embedded Systems. Springer International Publishing, 2017 (cit. on p. 2).

[Max02] Maxion, Roy A. and Kymie M. C. Tan: "Anomaly Detection in Embedded Systems." *IEEE Trans. Computers* (2002), vol. 51(2): pp. 108–120 (cit. on p. 52).

[McC10] McCune, J.M., Yanlin Li, Ning Qu, Zongwei Zhou, A. Datta, V. Gligor, and A. Perrig: "TrustVisor: Efficient TCB Reduction and Attestation." *Security and Privacy (SP), 2010 IEEE Symposium on*. 2010: pp. 143–158 (cit. on p. 56).

[McC08] McCune, Jonathan M., Bryan J. Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki: "Flicker: An Execution Infrastructure for TCB Minimization." *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*. Eurosys '08. Glasgow, Scotland UK: ACM, 2008: pp. 315–328 (cit. on p. 56).

[Mer17] Merriam-Webster, Inc.: *Autotelic | Definition of Autotelic by Merriam-Webster*. 2017. url: `https://www.merriam-webster.com/dictionary/autotelic` (visited on 11/01/2017) (cit. on p. 6).

[Mit05] Mitchell, C. and Institution of Electrical Engineers: *Trusted Computing*. Computing and Networks Series. Institution of Engineering and Technology, 2005 (cit. on p. 12).

[Mul11] Mulliner, Collin, Nico Golde, and Jean-Pierre Seifert: "SMS of death: from analyzing to attacking mobile phones on a large scale." *Proceedings of the 20th USENIX conference on Security*. SEC'11. San Francisco, CA: USENIX Association, 2011: pp. 24–24 (cit. on pp. 55, 73, 76).

[Mut08] Muthukumaran, Divya, Anuj Sawani, Joshua Schiffman, Brian M. Jung, and Trent Jaeger: "Measuring integrity on mobile phone systems." *Proceedings of the 13th ACM symposium on Access control models and technologies*. SACMAT '08. Estes Park, CO, USA: ACM, 2008: pp. 155–164 (cit. on p. 55).

[Nau10] Nauman, Mohammad, Sohail Khan, Xinwen Zhang, and Jean-Pierre Seifert: "Beyond Kernel-Level Integrity Measurement: Enabling Remote Attestation for the Android Platform." *Trust and Trustworthy Computing, Third International Conference,*

TRUST 2010, Berlin, Germany, June 21-23, 2010. Proceedings. Vol. 6101. Lecture Notes in Computer Science. Berlin: Springer, 2010: pp. 1–15 (cit. on p. 55).

[Nei06] Neiger, G., A. Santoni, F. Leung, D. Rodgers, and R. Uhlig: "Intel virtualization technology: Hardware support for efficient processor virtualization." *Intel Technology Journal* (Aug. 2006), vol. 10(3): pp. 167–177 (cit. on pp. 41, 43, 136).

[Nie02] Niemi, A., J. Arkko, and V. Torvinen: *Hypertext Transfer Protocol (HTTP) Digest Authentication Using Authentication and Key Agreement (AKA)*. RFC 3310. Internet Engineering Task Force, Sept. 2002 (cit. on p. 75).

[Ope06] OpenSSL Project: *OpenSSL*. 2006. url: `http://www.openssl.org/` (visited on 11/01/2017) (cit. on p. 112).

[Osh13] Oshana, R.: *Software Engineering for Embedded Systems: Methods, Practical Techniques, and Applications*. EngineeringPro collection. Elsevier Science, 2013 (cit. on p. 2).

[Pan10] Pandaboard.org: *PandyBoard*. Oct. 2010. url: `http://pandaboard.org` (visited on 03/19/2017) (cit. on p. 101).

[Pay11] Payne, Bryan D.: "Virtual Machine Introspection." *Encyclopedia of Cryptography and Security*. Ed. by Tilborg, Henk C. A. van and Sushil Jajodia. Boston, MA: Springer US, 2011: pp. 1360–1362 (cit. on p. 51).

[Pfo10] Pfoh, Jonas, Christian A. Schneider, and Claudia Eckert: "Exploiting the x86 Architecture to Derive Virtual Machine State Information." *Fourth International Conference on Emerging Security Information Systems and Technologies, SECURWARE 2010, Venice, Italy, July 18-25, 2010*. Ed. by Savola, Reijo, Masaru Takesue, Rainer Falk, and Manuela Popescu. IEEE Computer Society, 2010: pp. 166–175 (cit. on p. 51).

[Pfo13] Pfoh, Jonas, Christian A. Schneider, and Claudia Eckert: "Leveraging String Kernels for Malware Detection." *Network and System Security - 7th International Conference, NSS 2013, Madrid, Spain, June 3-4, 2013. Proceedings*. Ed. by Lopez, Javier, Xinyi Huang, and Ravi Sandhu. Vol. 7873. Lecture Notes in Computer Science. Springer, 2013: pp. 206–219 (cit. on p. 52).

[Pop74] Popek, Gerald J. and Robert P. Goldberg: "Formal Requirements for Virtualizable Third Generation Architectures." *Commun. ACM* (July 1974), vol. 17(7): pp. 412–421 (cit. on p. 45).

[Pre13] Preschern, Christopher, Andreas Johann Hormer, Nermin Kajtazovic, and Christian Kreiner: "Software-Based Remote Attestation for Safety-Critical Systems." *Sixth IEEE International Conference on Software Testing, Verification and Validation,*

ICST 2013 Workshops Proceedings, Luxembourg, Luxembourg, March 18-22, 2013. IEEE Computer Society, 2013: pp. 8–12 (cit. on p. 53).

[Rac12] Raciti, Massimiliano and Simin Nadjm-Tehrani: "Embedded Cyber-Physical Anomaly Detection in Smart Meters." *Critical Information Infrastructures Security - 7th International Workshop, CRITIS 2012, Lillehammer, Norway, September 17-18, 2012, Revised Selected Papers*. Ed. by Hämmerli, Bernhard M., Nils Kalstad Svendsen, and Javier Lopez. Vol. 7722. Lecture Notes in Computer Science. Springer, 2012: pp. 34–45 (cit. on p. 52).

[Raj16] Raj, Himanshu, Stefan Saroiu, Alec Wolman, Ronald Aigner, Jeremiah Cox, Paul England, Chris Fenner, Kinshuman Kinshumann, Jork Loeser, Dennis Mattoon, Magnus Nystrom, David Robinson, Rob Spiger, Stefan Thom, and David Wooten: "fTPM: A Software-Only Implementation of a TPM Chip." *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX: USENIX Association, 2016: pp. 841–856 (cit. on p. 151).

[Rus81] Rushby, J. M.: "Design and Verification of Secure Systems." *Proceedings of the Eighth ACM Symposium on Operating Systems Principles*. SOSP '81. Pacific Grove, California, USA: ACM, 1981: pp. 12–21 (cit. on p. 4).

[Sad04] Sadeghi, Ahmad-Reza and Christian Stüble: "Property-based attestation for computing platforms: caring about properties, not mechanisms." *Proceedings of the 2004 workshop on New security paradigms*. NSPW '04. Nova Scotia, Canada: ACM, 2004: pp. 67–77 (cit. on p. 54).

[Sai04] Sailer, Reiner, Xiaolan Zhang, Trent Jaeger, and Leendert van Doorn: "Design and Implementation of a TCG-based Integrity Measurement Architecture." *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*. SSYM'04. San Diego, CA: USENIX Association, 2004: p. 16 (cit. on pp. 26, 27, 50, 86, 90, 98, 99, 128, 136).

[Sam13] Samsung: *White Paper : An Overview of Samsung KNOX™*. Apr. 2013. url: `http://www.samsung.com/global/business/business-images/resource/white-paper/2013/05/Samsung_KNOX_whitepaper_April2013_v1.1-0.pdf` (visited on 11/01/2017) (cit. on p. 50).

[Sch09] Schiffman, J., T. Moyer, C. Shal, T. Jaeger, and P. McDaniel: "Justifying Integrity Using a Virtual Machine Verifier." *Computer Security Applications Conference, 2009. ACSAC '09. Annual*. 2009: pp. 83–92 (cit. on p. 56).

[Sch12] Schiffman, Joshua, Hayawardh Vijayakumar, and Trent Jaeger: "Verifying System Integrity by Proxy." *Proceedings of the 5th international conference on Trust and Trustworthy Computing*. TRUST'12. Vienna, Austria: Springer-Verlag, 2012: pp. 179–200 (cit. on pp. 56, 96, 109).

[Sch11] Schneider, Christian A., Jonas Pfoh, and Claudia Eckert: "A Universal Semantic Bridge for Virtual Machine Introspection." *Information Systems Security - 7th International Conference, ICISS 2011, Kolkata, India, December 15-19, 2011, Procedings*. Ed. by Jajodia, Sushil and Chandan Mazumdar. Vol. 7093. Lecture Notes in Computer Science. Springer, 2011: pp. 370–373 (cit. on p. 51).

[Ses04] Seshadri, A, A Perrig, L van Doorn, and P Khosla: "SWATT: SoftWare-based ATTestation for Embedded Devices." *IEEE Symposium on Security and Privacy, 2004. Proceedings. 2004*. 2004: pp. 272–282 (cit. on p. 53).

[Sha05] Shaneck, Mark, Karthikeyan Mahadevan, Vishal Kher, and Yongdae Kim: "Remote Software-Based Attestation for Wireless Sensors." *Security and Privacy in Adhoc and Sensor Networks, Second European Workshop, ESAS 2005, Visegrad, Hungary, July 13-14, 2005, Revised Selected Papers*. Ed. by Molva, Refik, Gene Tsudik, and Dirk Westhoff. Vol. 3813. Lecture Notes in Computer Science. Springer, 2005: pp. 27–41 (cit. on p. 53).

[Sir11] Sirer, Emin Gün, Willem de Bruijn, Patrick Reynolds, Alan Shieh, Kevin Walsh, Dan Williams, and Fred B. Schneider: "Logical attestation: an authorization architecture for trustworthy computing." *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. SOSP '11. Cascais, Portugal: ACM, 2011: pp. 249–264 (cit. on p. 54).

[Smi05] Smith, Jim and Ravi Nair: *Virtual Machines: Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and Design)*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2005 (cit. on p. 55).

[Sri10] Srinivasan, Raghunathan, Partha Dasgupta, Tushar Gohad, and Amiya Bhattacharya: "Determining the Integrity of Application Binaries on Unsecure Legacy Machines Using Software Based Remote Attestation." *Information Systems Security - 6th International Conference, ICISS 2010, Gandhinagar, India, December 17-19, 2010. Proceedings*. Ed. by Jha, Somesh and Anish Mathuria. Vol. 6503. Lecture Notes in Computer Science. Springer, 2010: pp. 66–80 (cit. on p. 53).

[Stu08] Stumpf, F. and C. Eckert: "Enhancing Trusted Platform Modules with Hardware-Based Virtualization Techniques." *Emerging Security Information, Systems and Technologies*. 2008: pp. 1–9 (cit. on p. 116).

[SYS91] SYSGO AG: *PikeOS*. 1991. URL: http://www.sysgo.com/ (visited on 11/01/2017) (cit. on pp. 40, 118).

[Tel13] Telekom Innovation Laboratories: *Highest security with the SiMKo 3 smartphone*. 2013. URL: http://www.laboratories.telekom.com/public/English/Newsroom/news/Pages/simko3-smartphone.aspx (visited on 11/01/2017) (cit. on p. 40).

[Tra09] Traynor, Patrick et al.: "On cellular botnets: measuring the impact of malicious devices on a cellular network core." *Proceedings of the 16th ACM Conference on Computer and Communications Security*. CCS '09. Chicago, Illinois, USA: ACM, 2009 (cit. on pp. 55, 76).

[Tro05] TrouSerS: *TrouSerS*. 2005. URL: http://trousers.sourceforge.net (visited on 11/01/2017) (cit. on pp. 102, 108).

[Tru10] Trusted Computing Group (TCG): *Mobile Trusted Module Specification*. Version 1.0, Revision 7.02. Apr. 2010 (cit. on p. 74).

[Tru12] Trusted Computing Group (TCG): *TNC Architecture for Interoperability, Version 1.5, Revision 4*. http://www.trustedcomputinggroup.org/wp-content/uploads/TNC_Architecture_v1_5_r4.pdf. May 2012 (cit. on p. 30).

[Tru11] Trusted Computing Group (TCG): *TPM Main Specification. Level 2, Version 1.2, Revision 116*. http://www.trustedcomputinggroup.org/resources/tpm_main_specification. Mar. 2011 (cit. on pp. 14, 16, 18, 54, 83, 90, 102, 122, 123, 126, 127).

[Tru14] Trusted Computing Group (TCG): *Trusted Platform Module Library Specification. Family "2.0". Level 00, Revision 01.16*. http://www.trustedcomputinggroup.org/resources/tpm_library_specification. Oct. 2014 (cit. on pp. 7, 137).

[Tru16] Trusted Computing Group (TCG): *Trusted Platform Module Library Specification. Family "2.0". Level 00, Revision 01.38*. http://www.trustedcomputinggroup.org/resources/tpm_library_specification. Sept. 2016 (cit. on pp. 7, 12, 15, 17, 19, 21, 22, 54, 137, 171).

[TUD11a] TU Dresden Operating Systems Group: *L4Re – The L4 Runtime Environment*. 2011. URL: http://os.inf.tu-dresden.de/L4Re/ (visited on 11/01/2017) (cit. on pp. 38, 92, 95).

[TUD11b] TU Dresden OS Group: *L4/Fiasco.OC*. Jan. 2011. URL: http://os.inf.tu-dresden.de/fiasco/ (visited on 11/01/2017) (cit. on pp. 38, 40, 116, 136).

[UEF12] UEFI Forum: *UEFI Specification Version 2.3.1 (Errata Revision "C")*. `http://www.uefi.org/sites/default/files/resources/UEFI_2_3_1_C.pdf`. June 2012 (cit. on p. 50).

[Vas12] Vasudevan, Amit, Emmanuel Owusu, Zongwei Zhou, James Newsome, and Jonathan M. McCune: "Trustworthy Execution on Mobile Devices: What Security Properties Can My Mobile Platform Give Me?" *Trust and Trustworthy Computing: 5th International Conference, TRUST 2012, Vienna, Austria, June 13-15, 2012. Proceedings*. Ed. by Katzenbeisser, Stefan, Edgar Weippl, L. Jean Camp, Melanie Volkamer, Mike Reiter, and Xinwen Zhang. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012: pp. 159–178 (cit. on p. 46).

[Vel17] Velten, Michael: "Hardware-based Integrity Protection Combined With Continuous User Verification in Virtualized Systems." Dissertation. Technische Universität München, 2017 (cit. on p. 58).

[Vel13] Velten, Michael and Frederic Stumpf: "Secure and Privacy-Aware Multiplexing of Hardware-Protected TPM Integrity Measurements among Virtual Machines." *Information Security and Cryptology – ICISC 2012: 15th International Conference, Seoul, Korea, November 28-30, 2012, Revised Selected Papers*. Ed. by Kwon, Taekyoung, Mun-Kyu Lee, and Daesung Kwon. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013: pp. 324–336 (cit. on p. 56).

[Vie17] Viegas, Eduardo, Altair Olivo Santin, Andre Luiz Pereira de Franca, Ricardo P. Jasinski, Volnei A. Pedroni, and Luiz S. Oliveira: "Towards an Energy-Efficient Anomaly-Based Intrusion Detection Engine for Embedded Systems." *IEEE Trans. Computers* (2017), vol. 66(1): pp. 163–177 (cit. on p. 52).

[Vog14] Vogl, Sebastian, Robert Gawlik, Behrad Garmany, Thomas Kittel, Jonas Pfoh, Claudia Eckert, and Thorsten Holz: "Dynamic Hooks: Hiding Control Flow Changes within Non-Control Data." *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014*. Ed. by Fu, Kevin and Jaeyeon Jung. USENIX Association, 2014: pp. 813–828 (cit. on p. 52).

[Wag16a] Wagner, Steffen and Claudia Eckert: "Policy-Based Implicit Attestation for Microkernel-Based Virtualized Systems." *Information Security: 19th International Conference, ISC 2016, Honolulu, HI, USA, September 3-6, 2016. Proceedings*. Ed. by Bishop, Matt and Anderson C A Nascimento. Cham: Springer International Publishing, 2016: pp. 305–322 (cit. on pp. 7, 65, 135).

[Wag15] Wagner, Steffen, Christoph Krauss, and Claudia Eckert: "Lightweight Attestation and Secure Code Update for Multiple Separated Microkernel Tasks." *Information Security: 16th International Conference, ISC 2013, Dallas, Texas, November 13-15, 2013, Proceedings*. Ed. by Desmedt, Yvo. Vol. 7807. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2015: pp. 20–36 (cit. on pp. 7, 65, 115).

[Wag12a] Wagner, Steffen, Christoph Krauss, and Claudia Eckert: "T-CUP: A TPM-Based Code Update Protocol Enabling Attestations for Sensor Networks." *Security and Privacy in Communication Networks: 7th International ICST Conference, SecureComm 2011, London, UK, September 7-9, 2011, Revised Selected Papers*. Ed. by Rajarajan, Muttukrishnan, Fred Piper, Haining Wang, and George Kesidis. Vol. 96. LNICST. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012: pp. 511–521 (cit. on p. 7).

[Wag16b] Wagner, Steffen, Sergej Proskurin, and Tamas Bakos: *TPM 2.0 Simulator Extraction Script*. `https://github.com/stwagnr/tpm2simulator`. Jan. 2016 (cit. on pp. 7, 137, 151).

[Wag12b] Wagner, Steffen, Sascha Wessel, and Frederic Stumpf: "Attestation of Mobile Baseband Stacks." *Network and System Security: 6th International Conference, NSS 2012, Wuyishan, Fujian, China, November 21-23, 2012. Proceedings*. Ed. by Xu, Li, Elisa Bertino, and Yi Mu. Vol. 7645. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012: pp. 29–43 (cit. on pp. 7, 65, 74).

[Wan10] Wang, Zhi and Xuxian Jiang: "HyperSafe: A Lightweight Approach to Provide Lifetime Hypervisor Control-Flow Integrity." *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*. IEEE Computer Society, 2010: pp. 380–395 (cit. on p. 52).

[Wei17] Weinhold, Carsten: *Microkernel-based Operating Systems – Introduction*. `http://os.inf.tu-dresden.de/Studium/KMB/WS2017/01-Introduction.pdf`. Oct. 2017 (cit. on p. 36).

[Wei16] Weiss, Michael: "System Architectures to Improve Trust, Integrity and Resilience of Embedded Systems." Dissertation. Technische Universität München, 2016 (cit. on pp. 58, 92).

[Wei14] Weiss, Michael, Steffen Wagner, Roland Hellman, and Sascha Wessel: "Integrity Verification and Secure Loading of Remote Binaries for Microkernel-Based Runtime Environments." *2014 IEEE 13th International Conference on Trust, Security and Privacy in Computing and Communications*. Sept. 2014: pp. 544–551 (cit. on pp. 6, 7, 58, 65, 89, 112).

[Win12] Winter, Johannes and Kurt Dietrich: "A Hijacker's Guide to the LPC Bus." *Public Key Infrastructures, Services and Applications*. Ed. by Petkova-Nikova, Svetla, Andreas Pashalidis, and Günther Pernul. Vol. 7163. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012: pp. 176–193 (cit. on p. 128).

[Wol07] Wolf, Marko, André Weimerskirch, and Thomas Wollinger: "State of the Art: Embedding Security in Vehicles." *EURASIP Journal on Embedded Systems* (2007), vol. 2007(1): p. 074706 (cit. on p. 1).

[Xen17] Xen Project: *Virtual Machine Introspection*. 2017. url: `https://wiki.xenproject.org/wiki/Virtual_Machine_Introspection` (visited on 11/01/2017) (cit. on p. 51).

[Xia12a] Xia, Yubin, Yutao Liu, Haibo Chen, and Binyu Zang: "CFIMon: Detecting violation of control flow integrity using performance counters." *IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2012, Boston, MA, USA, June 25-28, 2012*. Ed. by Swarz, Robert S., Philip Koopman, and Michel Cukier. IEEE Computer Society, 2012: pp. 1–12 (cit. on p. 52).

[Xia13a] Xiao, Han and Claudia Eckert: "Efficient Online Sequence Prediction with Side Information." *2013 IEEE 13th International Conference on Data Mining, Dallas, TX, USA, December 7-10, 2013*. Ed. by Xiong, Hui, George Karypis, Bhavani M. Thuraisingham, Diane J. Cook, and Xindong Wu. IEEE Computer Society, 2013: pp. 1235–1240 (cit. on p. 52).

[Xia13b] Xiao, Han and Claudia Eckert: "Lazy Gaussian Process Committee for Real-Time Online Regression." *Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence, July 14-18, 2013, Bellevue, Washington, USA*. Ed. by desJardins, Marie and Michael L. Littman. AAAI Press, 2013 (cit. on p. 52).

[Xia13c] Xiao, Han and Claudia Eckert: "Lazy Gaussian Process Committee for Real-Time Online Regression." *27th AAAI Conference on Artificial Intelligence (AAAI '13)*. Washington, USA: AAAI Press, July 2013 (cit. on p. 121).

[Xia12b] Xiao, Han, Huang Xiao, and Claudia Eckert: "Adversarial Label Flips Attack on Support Vector Machines." *ECAI 2012 - 20th European Conference on Artificial Intelligence. Including Prestigious Applications of Artificial Intelligence (PAIS-2012) System Demonstrations Track, Montpellier, France, August 27-31 , 2012*. Ed. by Raedt, Luc De, Christian Bessière, Didier Dubois, Patrick Doherty, Paolo Frasconi, Fredrik Heintz, and Peter J. F. Lucas. Vol. 242. Frontiers in Artificial Intelligence and Applications. IOS Press, 2012: pp. 870–875 (cit. on p. 52).

[Xia13d] Xiao, Han, Huang Xiao, and Claudia Eckert: "Learning from Multiple Observers with Unknown Expertise." *Proceedings of 17th Pacific-Asia Conference on Knowledge Discovery and Data Mining*. Gold Coast, Australia: Springer, Apr. 2013 (cit. on p. 121).

[Xia13e] Xiao, Han, Huang Xiao, and Claudia Eckert: "Learning from Multiple Observers with Unknown Expertise." *Advances in Knowledge Discovery and Data Mining, 17th Pacific-Asia Conference, PAKDD 2013, Gold Coast, Australia, April 14-17, 2013, Proceedings, Part I*. Ed. by Pei, Jian, Vincent S. Tseng, Longbing Cao, Hiroshi Motoda, and Guandong Xu. Vol. 7818. Lecture Notes in Computer Science. Springer, 2013: pp. 595–606 (cit. on p. 52).

[Xia15] Xiao, Huang, Battista Biggio, Blaine Nelson, Han Xiao, Claudia Eckert, and Fabio Roli: "Support vector machines under adversarial label contamination." *Neurocomputing* (2015), vol. 160: pp. 53–62 (cit. on p. 52).

[Yin10] Yingyou, Wen et al.: "A Secure Access Approach of UMTS Terminal Based on Trusted Computing." *Proceedings of the Int. Conf. on Networks Security, Wireless Comm. and Trusted Computing*. NSWCTC '10. USA, 2010 (cit. on p. 55).

[Yoo17] Yoon, Man-Ki, Sibin Mohan, Jaesik Choi, Mihai Christodorescu, and Lui Sha: "Learning Execution Contexts from System Call Distribution for Anomaly Detection in Smart Embedded System." *Proceedings of the Second International Conference on Internet-of-Things Design and Implementation, IoTDI 2017, Pittsburgh, PA, USA, April 18-21, 2017*. Ed. by Abdelzaher, Tarek F., P. R. Kumar, Alejandro P. Buchmann, and Chenyang Lu. ACM, 2017: pp. 191–196 (cit. on p. 52).

[Yoo15] Yoon, Man-Ki, Lui Sha, Sibin Mohan, and Jaesik Choi: "Memory heat map: anomaly detection in real-time embedded systems using memory behavior." *Proceedings of the 52nd Annual Design Automation Conference, San Francisco, CA, USA, June 7-11, 2015*. ACM, 2015: 35:1–35:6 (cit. on p. 52).

[Zha13] Zhang, Mingwei and R. Sekar: "Control Flow Integrity for COTS Binaries." *Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14-16, 2013*. Ed. by King, Samuel T. USENIX Association, 2013: pp. 337–352 (cit. on p. 52).

[Zha09] Zhang, Xinwen, Onur Aciiçmez, and Jean-Pierre Seifert: "Building Efficient Integrity Measurement and Attestation for Mobile Phone Platforms." *Security and Privacy in Mobile Information and Communication Systems*. Ed. by Schmidt, AndreasU. and Shiguo Lian. Vol. 17. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering. Springer Berlin Heidelberg, 2009: pp. 71–82 (cit. on p. 55).