Michael Weiß

Dissertation

System Architectures to Improve Trust, Integrity and Resilience of Embedded Systems

Technische Universität München

FAKULTÄT FÜR INFORMATIK
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Lehrstuhl für Sicherheit in der Informatik

System Architectures to Improve Trust, Integrity and Resilience of Embedded Systems

Michael Weiß

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. Uwe Baumgarten
Prüfer der Dissertation:
1. Univ.-Prof. Dr. Claudia Eckert
2. Univ.-Prof. Dr. Georg Sigl

Die Dissertation wurde am 15.03.2016 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 16.06.2016 angenommen.

Abstract

Virtualization has become one of the most important security enhancing techniques for embedded systems during the last years, both for mobile devices and cyber-physical systems (CPSs). In this work, we elaborate a security architecture based on a light-weight virtualization technique, the so called Trusted Execution Environment (TEE). Since currently available TEEs are either proprietary or highly hardware dependent, we provide a hardware independent approach by utilizing a microkernel as separation layer. To achieve a flexible and generic system architecture, we use trusted computing approaches based on the ideas of the Trusted Computing Group (TCG), as well as mobile security concepts of the GlobalPlatform, as building blocks for our own architecture. Thus, we could operate use-cases for mobile devices as well as use-cases based on trusted computing.

While most microkernel-based systems implement non-essential software components as user space tasks and strictly separate those tasks during runtime, they often rely on a static configuration and composition of their software components to ensure safety and security. Throughout this work, we extend our mikrokernel-based system architecture with a Trusted Platform Module (TPM) and propose a verification mechanism for our microkernel runtime environment, which calculates integrity measurements before allowing to load (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. We discuss the security of our proposed system architecture and protocols as well as TCB reduction by means of prototype implementations on ARM-based embedded platforms utilizing L4-based microkernel frameworks.

Even though our security architecture is conceptually sound, we identify a major security threat in this context. This threat is posed by cache-based side channel attacks. We show that the isolation characteristic of the proposed microkernel architecture as well as TEEs in general can be bypassed by the use of a cache timing attack. We elaborate a so called time-driven attack by which an adversary is able to extract sensitive keying material from an isolated trusted execution domain. To still provide a resilient system architecture against timing leakage, we discuss and propose several countermeasures. As this work aims platform independence, we just use multi-core or scheduler features for that purpose. We demonstrate the attack and provide evidence about the effectiveness of our countermeasures in our prototype implementations on ARM-based platforms.

Zusammenfassung

Virtualisierung ist in den letzten Jahren zunehmend zu einer der wichtigsten Sicherheitstechnologien für eingebettete Systeme geworden, sowohl für mobile Geräte als auch *Cyber-physical systems (CPSs)*. Diese Arbeit befasst sich mit der Herleitung einer Sicherheitsarchitektur basierend auf einer leicht gewichtigen Virtualisierungstechnik, welche als *Trusted Execution Environment (TEE)* bezeichnet wird. Aktuell verfügbare TEEs sind entweder proprietär oder stark Hardware-abhängig, deshalb wählen wir einen Hardware-unabhängigen Ansatz mittels eines Mikrokerns als Separationsschicht. Um eine flexible und generische Systemarchitektur zu erhalten nutzen wir sowohl Ansätze basierend auf den Konzepten der *Trusted Computing Group (TCG)* als auch der GlobalPlatform als Bausteine für unsere Architektur. Durch dieses Vorgehen können Anwendungsfälle für mobile Geräte und *Trusted Computing* gleichermaßen bedient werden.

Obwohl die meisten Mikrokern-basierten Systeme nicht essentielle Software-Komponenten als unprivilegierte Anwendungen im Benutzerkontext implementieren und diese während der Laufzeit streng voneinander separieren, werden oft statische Konfigurationen der Software-Komponenten vorausgesetzt um die IT-Sicherheit und Betriebssicherheit gewährleisten zu können. Im Verlauf dieser Arbeit wird daher unsere Mikrokern-basierte Systemarchitektur um ein *Trusted Platform Module (TPM)* erweitert und darauf aufbauend Verifikationsmechanismen vorgeschlagen. Diese Mechanismen berechnen Integritätsmessungen bevor ein (entferntes) Binärprogramm geladen wird. Demzufolge ist unser Ansatz der erste welcher die Hauptkonzepte von der *Integrity Measurement Architecture (IMA)*, ursprünglich für Linux-basierte Systeme gedacht, mit einem Mikrokernsystem verbindet. Im Vergleich jedoch reduziert unser Ansatz die *Trusted Computing Base (TCB)* signifikant und erlaubt eine strikte Separierung der Integritätsschutzkomponenten von einem parallel laufendem umfangreichen Betriebssystem, wie zum Beispiel Linux oder Android. Wir diskutieren die Sicherheit unserer System Architektur und Protokolle sowie die Reduzierung der TCB anhand von Prototypimplementierungen auf ARM-basierten eingebetteten Plattformen unter Nutzung von L4-basierten Mikrokernel-Frameworks.

Obgleich unsere Sicherheitsarchitektur konzeptionell stichhaltig ist, können wir eine gravierende Sicherheitsbedrohung in diesem Kontext ausmachen. Die Bedrohung besteht in Form eines sogenannten Cache-Seitenkanalangriffes. Wir zeigen auf, dass die Isolationseigenschaft der vorgeschlagenen Mikrokern-basierten Architektur und generell von TEEs durch eine zeitbasierte Attacke auf den Cache-Speicher umgangen werden kann. Es wird eine sogenannte *time-driven* Attacke hergeleitet, mit welcher ein Angreifer sensitives Schlüsselmaterial aus der vertrauenswürdigen Domäne extrahieren kann. Um trotzdem eine gegen zeitbasierte Cache-Angriffe widerstandsfähige Systemarchitektur zu ermöglichen, werden verschiedene Gegenmaßnahmen vorgeschlagen und diskutiert. Da diese Arbeit unter anderem Plattformunabhängigkeit zum Ziel hat, werden nur Multi-core oder generische Scheduler-Eigenschaften genutzt. Wir demonstrieren den Angriff und belegen die Effektivität der Gegenmaßnahmen mit unseren Prototypimplementierungen auf ARM-basierten Plattformen.

Contents

1 Introduction

The aerospace industry has been utilizing microkernel-based systems in their airplanes for decades. Currently, not only the aerospace but also the automotive industry is looking for secure systems based on microkernels or virtualization bringing together security and safety critical components with e.g., user controlled infotainment components on a single hardware platform. During the last years, embedded systems have evolved to highly integrated and interconnected powerful virtualized platforms so called cyber-physical systems (CPSs) being able to provide the hardware platform for those demands. However, currently deployed systems in vehicles and airplanes are still separated on different hardware platforms since severe security concerns especially on airplanes exist on bringing together critical systems like steering control and infotainment on one physical hardware platform.

Currently deployed microkernel 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. The ability to dynamically load remote binaries can be a desirable property with legitimate benefits, e.g., for updating defective software components remotely. A system which provides that ability must be able to verify the authenticity and integrity of the binary to preserve its trustworthiness.

On the other side there exist flexible security approaches on desktop and server computers in form of trusted computing providing flexible security designs. One approach to measure and verify the integrity of software binaries relies on a hardware security module (HSM), such as a Trusted Platform Module (TPM) [Tru11]. As specified by the Trusted Computing Group (TCG), a TPM can be used to securely store integrity measurements of software binaries, which are calculated during authenticated boot, where the current component in the boot chain hashes the next one before executing it.

Authenticated boot has the drawback that it only protects the bootchain and does not prevent execution of (potentially malicious) binaries after boot. To overcome this limitation, the Integrity Measurement Architecture (IMA) [SZJvD04] 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 does not prevent loading of remote binaries from an unknown source. Further do we see the need to reduce the trusted computing base (TCB) for the integrity components themselves. Since IMA as part of the large monolithic Linux kernel and other adaption of IMA to monolithic hypervisor kernels like Xen, also rely on the correctness of the corresponding kernel which results in a rather large TCB.

When we started with the research for this work, GlobalPlatform was in the process of specifying a Trusted Execution Environment (TEE) [Glo10] as light-weight virtualization environment for embedded systems. Nowadays these kind of light-wight virtualization concepts find its acceptance in embedded devices such as mobile phones. For instance, Qualcomm provides a TEE realization based on ARM TrustZone (TZ) [ARM09] called Qualcomm Secure Execution Environment (QSEE) which is used to provide on-the-fly-encryption for the persistent storage in Android-based smartphones. However, TZ implementations are highly vendor specific, proprietary and not applicable to every embedded system. In case of QSEE, also vulnerabilities in the proprietary TZ implementation were already published [Ros14].

In this thesis, we address the question on how these three different concepts, the microkernel for separation, the trusted computing approaches for integrity and attestation, and the trusted execution environment in combination with hardware security chips like the TPM can be combined in a generic security architecture for embedded systems. Further, our architecture approach should not only be applicable to special CPSs but also to mobile and home network equipment.

We will elaborate a trusted execution environment based on a microkernel operating system framework. With the use of the microkernel we address the goal to reduce the TCB in contrast to approaches based on monolithic kernels or hypervisors such as IMA.

Further, we will elaborate a TPM-based integrity verification service for this microkernel-based TEE that allows to securely load remote binaries. 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 party. With this approach, we enable more flexible system designs for future microkernel-based systems not only in the aerospace and automotive domain but also for mobile devices.

It was generally believed long time that virtualization characteristics such as the TEE provide an isolated execution environment where sensitive code can be executed isolated from untrustworthy applications. However, we will show in this work that this isolation characteristic can be bypassed by the use of cache timing attacks. Even though it has already been stated that cache timing attacks may circumvent the virtualization barriers in [GSS+07], we provide additional practical evidence to that matter.

Especially, we show how this side channel can also be exploited on embedded ARM-based architectures as evolving CPU architectures also opened this new attack vector to embedded systems. We were able to derive an attack against ARM-based systems from a time-driven approach against x86-based PC platforms introduced by Bernstein [Ber05b]. With our corresponding work [WHS12], we were the first to examine this kind of attacks on an embedded light-weight virtualization setup with a realistic attacker model for real world attack scenarios. During the last years several countermeasures were proposed and are already in place like the Intel AES-NI Instruction Set Architecture (ISA) extension for their x86 processors. However, for embedded systems there still are some open issues we address within this work, e.g., we want to answer the question: How could multi-core CPUs effect time driven cache attacks and how could they be used to mitigate these kind of attacks in a TEE without relying on special hardware features? Finally, we aim to provide a timing leakage resilient architecture by proposing hardware independent countermeasures.

1.1 Contributions

In the following, we discuss our contributions in more detail.

Trusted Execution Environment based on Microkernel Framework
We provide a system architecture which realizes a TEE using a microkernel framework for embedded systems. Our work can be seen as a merge of the concepts the Nizza [HHF+05] architecture is providing and the ones provided by the GlobalPlatform (GP) TEE architecture. Further, our architecture comprises a TPM adaption as trust anchor for the trusted part of the system. By using a microkernel and para-virtualization, we achieve a vendor independent realization in contrast to ARM TZ or Intel Trusted Execution Technology (TXT).

Combining GlobalPlatform and Trusted Computing Approaches With our overall architecture, we combine the trusted execution approach of the GlobalPlatform with the concepts for trusted computing of the TCG. Both, the GlobalPlatform as well as the TCG, provide concepts which are based on hardware and software layers. We describe how these different concepts can be mapped to our microkernel-based system architecture. We show how the TEE can be realized with our microkernel-based architecture. Furthermore, we provide software abstractions realizing the TCG Software Stack (TSS). This contributes to the goal of a generic architecture which also directly applies to mobile devices. We specify a subset of the TPM command set for our embedded use cases. With this we also address our goal of an overall reduced TCB.

Integrity and Attestation Protocols for Trusted Runtime To provide trust in our system, especially the trusted runtime environment, we make use of the IMA approach proposed by Sailer et al. [SZJvD04] for Linux-based PC-Systems. We introduce new entities in user-space which take over the functions realized in privileged kernel space of their approach. We show that our approach reduces the TCB compared to the original approach significantly.

Secure Remote Loading Procedure for Trusted Applications In addition to the integrity and attestation protocols, we want to allow more flexible binary execution on microkernel systems. To achieve that, we provide a secure loading service which is able to assure that an encrypted remote binary is executed in a trusted environment on a specific device. Our approach relies on trusted computing featuring a TPM without relying on application processor-specific security features, such as ARM TrustZone [ARM09]. We show that this indeed contributes to the goal of a more flexible microkernel system design compared to currently deployed static systems by providing an informal security analyses of the procedure.

Cache-based Side Channel Attack in embedded Virtualization Context We show that our architecture as well as TEE-based or virtualization-based architectures in general are vulnerable against cache-timing attacks. We show this by means of an adapted version of Bernstein's time-driven cache attack. In contrast to previous work which mainly dealt with either x86-based hardware architectures or dedicated micro controllers, we provide results for ARM-based embedded/mobile systems. For this purpose, two different Testbeds are considered in this work, one using the Cortex-A8 and the other using the Cortex-A9. We also consider different realizations of our TEE, namely a PikeOS and a Fiasco.OC/L4Re based setup.

Elaboration of Different Multi-core Scheduler Configurations In addition to the single core execution, we elaborate different multi-core scheduler configurations and evaluate their vulnerability against the herein proposed time-driven cache attack. By comparing the attack results between single- and multi-core configurations, we are able to show that dedicated cores for crypto services leak the most information about the key.

Hardware independent Countermeasures against Cache Timing Attacks We provide two hardware independent countermeasures to mitigate cache-based side channel attacks. Our first approach is a randomized multi-core software implementation of AES which we call RMC-AES. In contrast to other approaches like bit-slicing which needs single instruction, multiple data (SIMD) extensions, e.g., Intel MMX and ARM NEON, RMC-AES can be implemented on any 32bit platform. Although, we do not gain good performance, we step forward to achieve a leakage resilient system implementation with this approach. Further, we propose a countermeasure which is based on a real-time scheduler configuration to provide a *discrete-time* execution for the component which is executing the cipher. This countermeasure can be used as a drop-in update to existing systems. We show that by an evaluation on an ARM-based quad-core processor.

1.2 Publications

Parts of this thesis have been published. In the following, we provide a brief overview about these scientific, peer-reviewed articles and which chapters cover them.

In [WHS12], we propose our cache-based side-channel attack on a virtualization-based security architecture for embedded systems. This is mainly covered by Chapter 7. Further, in this context, we provide an evaluation of the same attack in a different multi-core-based embedded system and propose the *discrete-time* countermeasure in [WWAS15], which is also covered by Chapter 7. However, the fundamental architecture of both of them, namely the microkernel-based trusted execution environment is the fundamental part of Chapter 5. In [WWAS15], we propose a security architecture and protocols for remote binary loading, integrity and attestation inside a trusted runtime based on microkernel systems. This work is covered by Chapter 6 while the high-level system architecture using trusted computing is part of Chapter 5. Another work we contributed to, is the joint work with Proskurin et al. [PWS15] which provides seTPM, a Java Card design and evaluation of a TPM 1.2 and TPM 2.0 compliant TPM emulator. The TPM 1.2 architecture and design of that work is also part of this thesis and discussed in Chapter 5.

Furthermore, some auxiliary contributions were made in the context of mobile embedded devices which are not part of this work. We contributed to the joint work of Horsch et al. [HBW$^+$14] in which a secure identity derivation protocol for mobile devices was proposed and evaluated. In contrast to the topics covered by this thesis, no trusted execution environment is necessary. The trustworthiness needed for that protocol relies only on the used Java Card Secure Element (SE). Concurrently, to the microkernel-based research, we contributed to the *trust-me* project. This is a follow-up development project to the work presented in [WHSE15]. In *trust-me*, we implemented a virtualization concept based on so called operating system (OS)-level virtualization for Android-based mobile devices. The separation is here provided by the OS kernel for several independent user space Android stacks with the consequence of a larger TCB. However, the performance of this approach is almost native. The resulting research with the underlying security concept is joint work with Huber et al. [HHV$^+$15].

References

[HBW$^+$14] Julian Horsch, Konstantin Böttinger, Michael Weiß, Sascha Wessel, and Frederic Stumpf. Trustid: Trustworthy identities for untrusted mobile devices. In *Proceedings of the 4th ACM Conference on Data and Application Security and Privacy*, CODASPY '14, pages 281–288, New York, NY, USA, 2014. ACM.

[HHV$^+$15] Manuel Huber, Julian Horsch, Michael Velten, Michael Weiss, and Sascha Wessel. A secure architecture for operating system-level virtualization on mobile devices. In *Proceedings of the 11th International Conference on Information Security and Cryptology (Inscrypt 2015)*, 2015.

[PWS15] Sergej Proskurin, Michael Weiß, and Georg Sigl. seTPM: Towards Flexible Trusted Computing on Mobile Devices based on GlobalPlatform Secure Elements. In *Smart Card Research and Advanced Application, 14th International Conference, CARDIS 2015, Bochum, Germany, November 4-6, 2015. Proceedings*, 2015.

[WHS12] Michael Weiß, Benedikt Heinz, and Frederic Stumpf. A cache timing attack on aes in virtualization environments. In Angelos D. Keromytis, editor, *Financial Cryptography and Data Security*, volume 7397 of *Lecture Notes in Computer Science*, pages 314–328. Springer Berlin Heidelberg, 2012.

[WWAS15] Michael Weiß, Benjamin Weggenmann, Moritz August, and Georg Sigl. *Trusted Systems: 6th International Conference, INTRUST 2014, Beijing, China, December 16-17, 2014, Revised Selected Papers*, chapter On Cache Timing Attacks Considering Multi-core Aspects in Virtualized Embedded Systems, pages 151–167. Springer International Publishing, Cham, 2015.

[WWHW14] Michael Weiss, Steffen Wagner, Roland Hellmann, and Sascha Wessel. Integrity Verification and Secure Loading of Remote Binaries for Microkernel-based Runtime Environments. In *13th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*. IEEE Computer Society, 2014.

1.3 Outline

In the next chapter we will provide fundamental background information. As this thesis combines several different research aspects, that chapter will introduce trusted computing, GlobalPlatform trusted execution environments and secure elements, microkernel in general and the L4 microkernel family in particular and finally some basics about cache-based side channel attacks including the AES cipher as sample target.

In Chapter 3, we provide related work on integrity and secure loading to motivate our system architecture. This chapter provides an insight into why we choose the microkernel-based approach.

We introduce some scenarios and a coarse attacker model which are used throughout the whole thesis in Chapter 4. The provided scenarios either further motivate our architecture or provide a better understanding for the application of TEEs in general.

We specify and discuss the system architecture in Chapter 5. This chapter shows how the concepts of trusted computing and GlobalPlatform can be combined in an overall microkernel-based system architecture. We describe a secure boot process based on the TPM to establish trust into our system. Further, we show how a GlobalPlatform complaint secure element can be integrated as TPM replacement.

In Chapter 6, we discuss trusted computing based protocols for integrity and attestation as well as our concept to securely load remote binaries in the trusted compartment. We provide a substantial discussion about a prototype implementation of the provided concept. Further, an informal security evaluation as well as evaluations of performance and code size of our approach is to be found in Chapter 6.

Chapter 7 will discuss the whole aspect of cache timing vulnerabilities of TEEs. We start with a section about related work on cache-based side channels. Afterwards, we discuss the time-driven cache attack on TEEs and several countermeasures including the *discrete-time* scheduling approach and RMC-AES. We provide thorough attack and countermeasure evaluation in different testbeds under several configurations.

Finally, Chapter 8 concludes this thesis and provides an outlook to further research directions.

2 Background

Since this thesis covers several research areas, we will give the reader the corresponding background of each of them. The covered topics are trusted computing, GlobalPlatform, microkernel systems and cache-based side channel attacks. As throughout this work, we elaborate a system architecture which provides adoptions and extensions to trusted computing approaches, we first introduce some essential background on this topic. This includes the Trusted Platform Module (TPM) chip as well as the software stack and trusted boot.

We later also combine approaches from the GlobalPlatform into our architecture, such as the Trusted Execution Environment (TEE) and the Secure Element (SE). That is why we also provide the basic background on that. Instead of the original approaches of the TCG which bases their concept on monolithic operating system architectures, we provide concepts and implementations on microkernel-based operating systems. We denote the difference of those systems in general. Background on the L4 microkernel family in particular is provided as we use two – one academic open source and one commercial kernel – out of this microkernel family to demonstrate the feasibility of our design and protocols. Finally, we denote brief background on cache timing side channels as a main part of this thesis deals with the evaluation and countermeasures of those kind of attacks in our architecture. This also includes background on the cryptographic algorithm AES, which we use as representative cipher algorithm for this purpose.

2.1 Trusted Computing

Trusted computing is the term for protocols, soft and hardware designs provided by the TCG. The TCG is a consortium of several industry and research institutions founded in 2003. It supersedes the former Trusted Computing Alliance (TCPA). Their goal is to establish trust into platforms by a hardware-based root of trust as vendor independent standards.

Since the TCG's main targets are PC and server platforms, the provided standards are targeting systems based on monolithic operating system kernels, such as Linux and Windows. Even though some efforts are made to support mobile platforms, microkernel-based systems are not targeted by the TCG. With this work we do transfer some of those concepts to a microkernel-based system. That is why we provide some trusted computing basics in the following. Note that this is information derived mainly from [Eck14] and [CYC$^+$08] with some additions from the TPM Main Specification [Tru11]. Any deeper knowledge about trusted computing concepts can be found there.

2.1.1 Trusted Platform Module

Centerpiece of the TCG specifications is the TPM which is a cryptographic co-processor with special hardening against physical and side channel attacks. Internally, the TPM comprises a true Random Number Generator (RNG), a set of cryptographic functions, such as a SHA1, HMAC hashing engine and an RSA crypto engine, limited internal volatile and non-volatile secure storage for keys and integrity measurements. Furthermore, a TPM is protected by several active sensors (shield) against physical attacks. A simplified Architecture is shown in Figure 2.1.

Root of Trust

The Roots of Trust are the start point for the process to establish trust in a platform. It is immutable to trust the roots of trust without any further validation. A misbehaviour of a root of trust cannot be revealed. A TCG conform system contains three such roots of trust, the Root of Trust for Measurement (RTM), Root of Trust for Storage (RTS) and Root of Trust for Reporting (RTR).

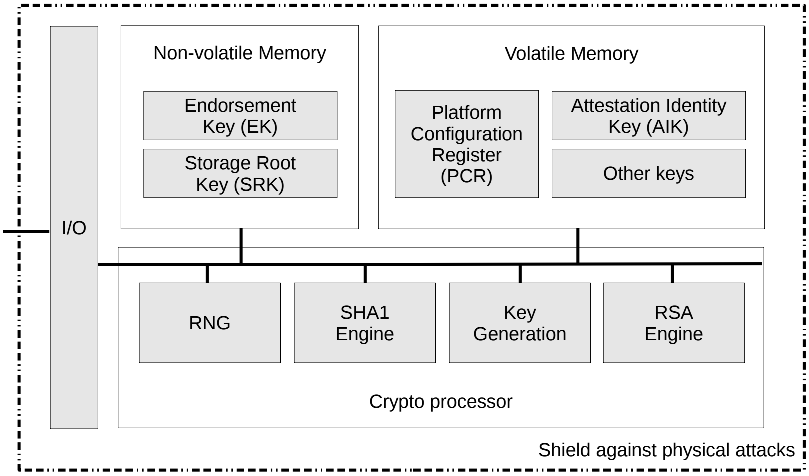


Fig. 2.1: Simplified block diagram of the TPM architecture

Root of Trust for Measurement The RTM is used to determine the integrity of a platform's configuration during boot. The corresponding computations start with Core Root of Trust for Measurement (CRTM) and are also called measurements. In the x86-based desktop and server environment the CRTM is part of the Basic Input Output System (BIOS). The rest of the RTM's functionality is implemented inside the TPM. This functionality comprises the used hash algorithm for measurements and a secure storage for them. The specification demands a SHA1 engine and so called platform configuration register (PCR) registers for volatile storage.

Root of Trust for Storage The RTS is used to protect keys and trusted data objects. It is represented by an RSA keypair called Storage Root Key (SRK). The SRK is generated and stored permanently in non-volatile memory when the TPM is owned by an user. There exist the special command *TPM_ TakeOwnership* for this. Afterwards the SRK cannot be changed anymore because it is used as root for all keys including non-migratable ones. However, a non-migratable key is defined to be bound to a specific TPM and therefore the key used to encrypt the non-migratable keys must be unique to that TPM during its usage period after deployment.

Root of Trust for Reporting The RTR is used to provide trustworthy reports of integrity protected data. The RTR is realized as an RSA key pair which is called Endorsement Key (EK). The EK is generated during production and stored inside persistent memory of the TPM. It is the only key which cannot be changed during a TPM's lifetime. However, the EK is never used directly to attest an identity. This is due to privacy concerns. For this purpose the TCG specifies identity keys (see below). The TCG specification demands that the EK should only participate during TPM Ownership insertion and Attestation Identity Key (AIK) creation and verification.

Platform Configuration Register

A PCR is volatile secure storage for measurement values. Since Version 1.2, a TPM has at least 24 of such PCR registers. The size of each PCR is 160bit so that it can store a SHA1 hash. Reading of a PCR is possible in an easy single operation. Nevertheless, a PCR_i can only be written by extending the new value x to the old value already stored inside of PCR_i as shown in Equation 2.1; $||$ denotes a concatenation.

$$extend(PCR_i, x) \equiv PCR_i \leftarrow SHA1(PCR_i||x) \tag{2.1}$$

Clearing a PCR is not possible without a reset of the whole system. This way, secure hash chains can be established to represent a trusted platform state afterwards.

2.1.2 Type of Keys

There are different kind of keys used in the TCG architecture. The specification distinguishes between migratable and non-migratable keys. Migratable keys can be migrated from one to another TPM while non-migratable keys are bound to the specific TPM. Due to limited space for keys inside the TPM, keys are usually stored externally on hard disk or flash memory. Only for usage, they are loaded into a free key slot in the TPM's volatile memory. Those keys have to be encrypted to maintain their security.

We already introduced two special non-migratable keys, the EK and the SRK which form the particular root of trust RTR and RTS as described above. For this work, also the AIKs and binding keys are of special interest.

Signature Keys

Signature Keys are asymmetric variable length RSA keys. However, the specification demands a maximum size of 2048 bit to be available for signature keys. These keys can be used to sign arbitrary application data out side of the TPM. They could be either created as non-migratable or migratable key pairs.

Storage Keys

Storage keys are used to securely store data. Usually, other keys like binding and signature keys are encrypted with storage keys. This is also called key wrapping. However, storage keys are not used to encrypt symmetric keys. Storage keys are always 2048 bit RSA private keys which could either be migratable or non-migratable. The SRK is an example for a non-migratable storage key.

Binding Keys

The primary use for binding keys lies in the encryption of symmetric keys. Even if binding keys could be used to encrypt arbitrary data, they can only encrypt a small amount of data as the TPM only does basic RSA operations with binding keys. This means the payload size is restricted to the RSA block size As also binding keys are 2048 bit RSA keys this is 256 byte. Therefore, it is not reasonable to encrypt other data than symmetric keys.

Identity Keys

Identity keys also called AIKs are special signature keys which are only allowed to sign data which originates inside the TPM such as PCRs. An AIK can be seen as an alias for the EK as the EK cannot directly be used to perform signatures. It is possible to create unlimited amounts of AIKs to generate several pseudonyms for different applicants. AIKs must be 2048 bit RSA key pairs. They are non-migratable keys which are always encrypted with a non-migratable storage key or as such directly with the SRK. Thus, in case the owner is cleared, it is assured that also all associated identities are destroyed.

Authentication Keys

Authentication keys are symmetric keys used to authenticate the communication between the host system and the TPM. This so called transport sessions are used to assure the confidential and integrity protected data exchange between the application or the operating system and the TPM. Thus, a physical attacker on the bus cannot eavesdrop the communication between the TPM chip and application processor. The corresponding authorization protocols are the Object Specific Authorization Protocol (OSAP) and the Object Independent Authorization Protocol (OIAP).

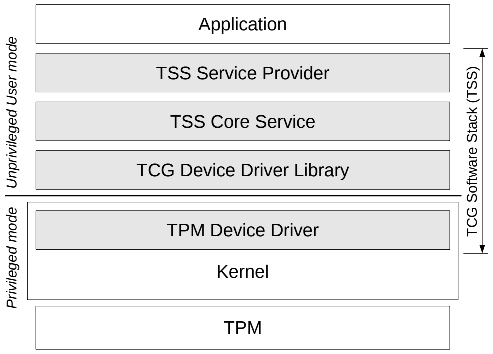


Fig. 2.2: TCG architecture

Legacy Keys

As the name indicates, those keys are legacy and should not be used. For this thesis they are therefore not relevant.

2.1.3 TCG Software Architecture

To maintain a trusted platform, the TCG software architecture provides a modular design that distributes tasks across layers in user- and kernel space as shown in Figure 2.2). Every layer of the TCG architecture provides an abstract interface towards its respective upper layer. The user space components comprise the TCG Device Driver Library (TDDL), TCG Software Stack Core Services (TCS), TCG Service Provider (TSP) and an application making use of the above layers. In short, the TPM device driver establishes a communication with the TPM, providing an interface towards user space layers. As can be seen by this definition, the TCG software architecture is targeting monolithic operating system kernels such as Linux and Microsoft Windows as common PC platforms.

2.1.4 Integrity and Attestation

As specified by the TCG, a TPM can be used to securely store integrity measurements of software binaries, which are calculated during authenticated boot,

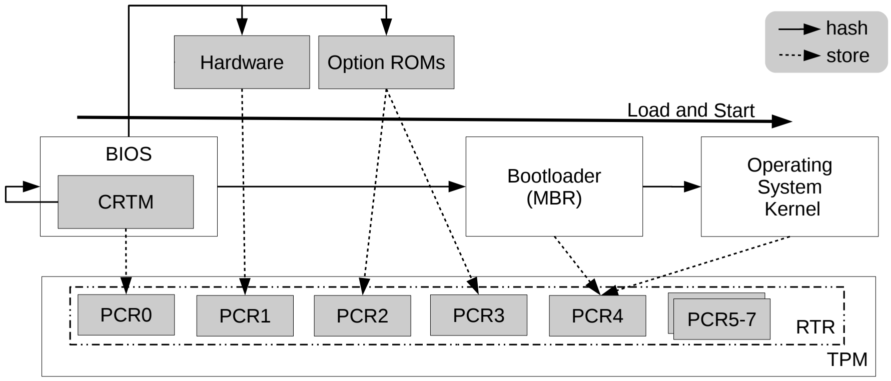


Fig. 2.3: TCG authenticated boot procedure

where the current component in the boot chain hashes the next one before executing it. Starting from a root of trust for measurement (usually a very small piece of code in read-only memory), those integrity measurement values are calculated and securely stored inside the TPM using the RTR.

Figure 2.3 shows the authenticated boot procedure as defined by the TCG for PC platforms. There, the CRTM is part of the BIOS and measures (calculates a hash over) itself and stores the result in *PCR* 0. Further, the BIOS measures some parts of the hardware (register, memory states) to *PCR* 1 and Option ROMs to *PCR* 2 and *PCR* 3. Before the actual bootloader code from the Master Boot Record (MBR)[1] is started, the BIOS measures the MBR and stores the result to *PCR* 4. In *PCR* 5-7 additional information such as transactions of Option Rom execution, configuration and image files are stored during boot. Afterwards the bootloader measures the operating system kernel and extends the value also to *PCR* 4. Finally, the OS kernel is started.

These integrity measurement values can then be used in a remote attestation to prove the integrity of the system to a remote verifier. For a remote attestation, the hash values representing the current platform state are digitally signed by the TPM using an AIK. This signed platform state also called "Quote" is sent to the remote party that can verify the signature and check the integrity measurements by comparing them to known trustworthy reference values. For a detailed protocol description we refer to [Eck14].

[1] The MBR is the first sector of the bootable hard disk in common PC platforms

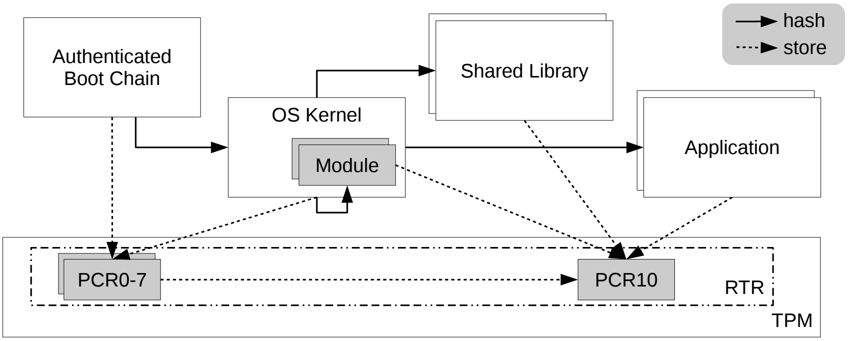


Fig. 2.4: Integrity measurement architecture

Integrity Measurement Architecture

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 compromise the system in the meantime. To overcome this limitation, the IMA [SZJvD04] has been proposed for Linux-based systems, where integrity values are calculated during run-time whenever a binary is loaded. IMA is included in Linux mainline kernel since version 2.6.30.

This extension to the authenticated boot chain is depicted in Figure 2.4. After the OS kernel is loaded, IMA as part of the Kernel measures each binary before it is executed and stores the result in a measurement list. Each entry of the measurement list is extended to the *PCR* 10. As start of the hash chain in *PCR* 10 the so called boot aggregate is stored in *PCR* 10 which is computed over *PCR* 0-7 which are specified by the TCG for usage during boot.

IMA is proposed as a Linux Security Module (LSM) module for Linux. The Linux kernel provides security hooks distributed in the whole kernel code, allowing an LSM instance to instrument several parts of kernel code execution. The measurements are made inside the kernel using such a security hook (`file_mmap`) which is called whenever a new file is mapped into memory. Thus, also when the loader code maps the new binary into memory before the `execve` system call is invoked, the IMA module enforces the integrity measurement for loaded binaries. This is sufficient if the binary has no library dependencies or is statically linked. However, in a Linux system, binaries usually depend upon shared libraries. Those are mapped by the linker flagged executable into virtual memory with the `mmap` system call. This again invokes the `file_mmap` hook which is called whenever a file is mapped into memory and therefore also for shared libraries the measurement is enforced. Further, in monolithic kernels not all kernel code is loaded at boot time. Some device drivers are realized as so called kernel modules which can be loaded into the kernel during runtime by invoking a `sys_init_module` system call. Afterwards the module code starts execution inside the kernel. Thus, Sailer et al. [SZJvD04] propose to implement a measure call inside the load module routine, because there is no security hook which could be utilized to instrument the module loading procedure. As IMA maintains an aggregate of all measured application binaries, libraries and kernel modules in secure storage of the TPM (*PCR* 10), this aggregate could be used during attestation and therefore, provide a much more recent state of the running system, instead of just reporting the state after booting the operating system. Unfortunately, IMA focuses on Linux-based systems and does not prevent loading of remote binaries from an unknown source. This presents a major threat to the system's integrity. Thus, loading such binaries is usually not acceptable for systems with safety- or security-critical applications, such as secure payments or online banking on a smartphone.

2.2 GlobalPlatform

GlobalPlatform is an industry consortium which is similar to the TCG a non-profit, member driven association. Their focus is on mobile, embedded devices to incorporate secure chip technology for several secure applications like payment and Digital Rights Management (DRM)[2]. In contrast to the TCG which has members who are strictly supporting the *opt-in* approach which gives the user the control of the TPM and let him decide for what purpose he wants to use it, the GlobalPlatform officially supports DRM use-cases where vendors control the privileged hard- and software components. The two main topics of the GlobalPlatform are TEE and SE, which both provide separate compartments to execute critical code, however with different level of security. While a TEE can be realized as software-only solution, the SE provides a separate physical hardened security chip similar to the TPM.

[2] A statement of the author: *We highly disagree with the ideas behind DRM. However, the industry provides the hardware and some concepts which we can use for "good" security, namely to provide trust into software systems. This allows to defend against malware and provide integrity protected systems which no intelligence apparatus easily can manipulate to spy on high value data.*

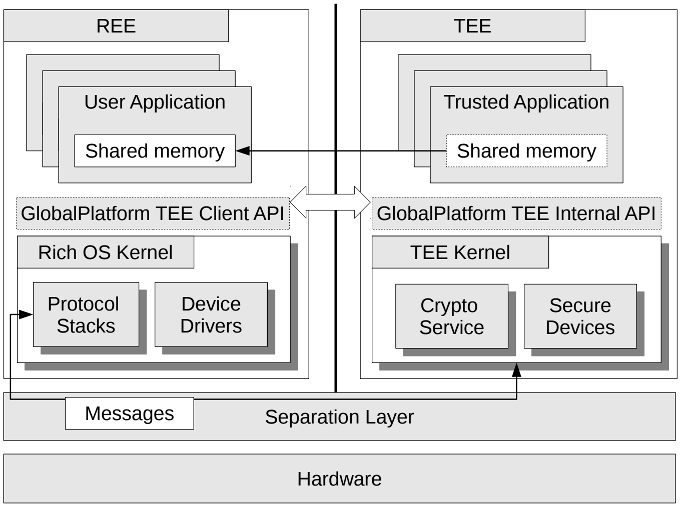


Fig. 2.5: GlobalPlatform high level security architecture of an embedded device based on virtualization

2.2.1 Trusted Execution Environment

GlobalPlatform has specified a high level system architecture of a trusted execution environment (TEE) [BFG+11], as shown in Figure 2.5. This is an abstract version of security architecture provided in the TEE Client API Specification [Glo10]. The system architecture consists of two execution domains, the trusted execution environment for the trusted applications and the rich environment for the user controlled rich operating system[3]. It is much more likely that the rich execution environment (REE) is infected by malware due to the greater software complexity. The trusted applications are either executed in their own virtual machine or are separated in different address spaces and do not share any memory to allow the deployment of trusted application by different vendors which may not trust each other. However, each trusted application depends on the security of the underlying isolation layer, which is a hardware-based dedicated split of resources. The ARM TZ [ARM09] provides an implementation of this separation layer for instance. In this thesis we provide a TEE realization on a software-only solution without the need of hardware dependent features, see Chapter 5.

[3] A rich operation system is a full operating system with drivers, userland and user interfaces, e. g., Android

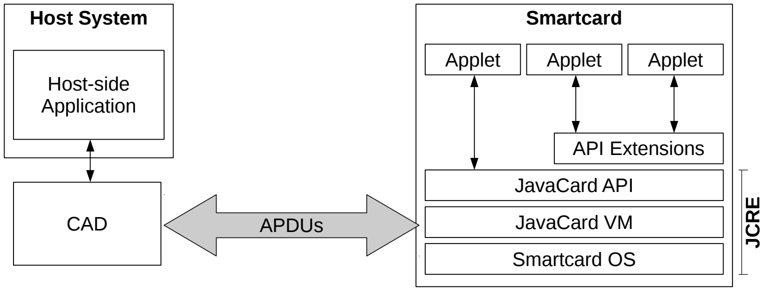


Fig. 2.6: Overview of the Java Card technology platform

2.2.2 Secure Element

The GlobalPlatform defined SE is the denotation for a security chip residing in any form factor, such as Universal Integrated Circuit Card (UICC), embedded SE, secure microSD or just an ordinary smart card. The main difference to standard smart cards is the running firmware on this chip. GlobalPlatform specifies System Architecture and communication APIs in their Card Specification [Glo11]. For external communication the specification demands to use corresponding ISO protocols for smart card communication such as ISO 7816 contact based and ISO 14443 contactless smart cards. There are only two concrete Implementations for this specification namely the Java Card and the MULTOS framework. We concentrate on the Java Card framework in this work, as we believe that it is the more flexible and more popular framework. The corresponding Java Card specification is 2.2.x.

Java Card Technology The Java Card technology specification comprises the Java Card Virtual Machine (JCVM), Java Card Runtime Environment (JCRE) and the Java Card API specification.

1. JCVM specification defines the programming language which is a subset of the Java language and a light VM for smart cards
2. JCRE specification defines the runtime behavior for Java Cards
3. Java Card API specification defines the programming framework Java packages and classes for so called Java Card Applets

The JCVM and its underlying vendor specific Smartcard OS together with the Java Card API abstraction layer form the actual Java Card platform, as shown in Figure 2.6.

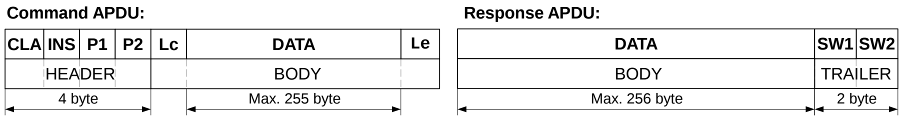


Fig. 2.7: ISO 7816-4 command and response APDUs

The advantage of this design approach is that the corresponding smart card applications (Applets) can be developed in a hardware independent manner. This is similar to the original idea of Java of an platform independent programming language. Thus, in total, the JCVM and JCRE provide hardware abstraction and an universal programming API for smartcard developers. As a consequence, the system setup does not rely on certain smartcard platform or manufacturer, but rather on limitations of the actual smartcard hardware such as RAM size and cryptographic co-processors.

Furthermore, the design allows to use one smartcard for several different purposes, by just deploying the appropriate Applet during runtime. For instance on an mobile device, the mobile network operator can install a payment applet beside its already deployed SIM applet on the UICC. As the JCVM provides a firewall between those Applets, it is assured that the code and data are strictly separated similar to major operating systems on desktop or mobile systems.

To distinguish and deploy proper firewall rules each Applet needs a so called Application Identifier (AID). This AID is registered inside the JCRE as last step of deployment. In each JCRE exists a privileged applet the card manager which has control over the firewall and allows for installation of new Applets. The AID of the card manager is protected and cannot be impersonated by an ordinary applet.

Communication As the GlobalPlatform specification demands, the communication between host and Java Card is established by the means of a message passing protocol using Application Protocol Data Units (APDUs). The corresponding case-4 command APDUs and respective response APDUs as defined in [ISO05] are depicted in Figure 2.7.

A command APDU consists of 4 byte header and body with a maximum size of 257 byte. The header is compounded of the instruction class (CLA) and type (INS), as well as the instructions parameters (P1 and P2). The body comprises a byte (Lc) which specifies the length of the data to follow, the actual data and a byte (Le) which defines the expected response size. The response APDU consist of a variable length body containing the payload data and a 2 byte trailer which contains the so called status word (SW1 and SW2).

The maximum payload of a single command APDU is 255 due to the included Lc byte is interpreted as size. The payload of a response APDU however can be 256 byte of length due to Le = $0x00$ is interpreted as the maximal length of 256 byte.

2.3 Microkernel Systems

One major goal of this work is reducing the TCB for trusted computing based applications. A main contributor to achieve that goal provides the use of a microkernel-based operating system as base for our concepts. Therefore, we discuss the difference between the microkernel design approach versus the monolithic kernel design usually applied in TCG based systems. Further, we give a short introduction to the L4 microkernel family because throughout this work, we demonstrate our concepts on the Fiasco.OC/L4Re as well as the PikeOS operating system frameworks which both are using kernels of the L4 family. The following subsections are summarizing information of the survey given by Elphinstone and Heiser [EH13] and the lecture by Weinhold [Wei14].

2.3.1 Microkernel versus Monolithic Kernel Design

Usually, the purpose of an operating system is to manage resources, which could either be hardware or software. Hardware resources are CPU, memory, low-level buses and devices such as a TPM. Software resources also called system services are for instance file systems and network stacks which also make use of device drivers. Further, it provides abstraction interfaces to access those resources, e.g., the read and write API through so called sockets instead of having to deal directly with TCP/IP packets in the application. Further, an essential job of the operating system is to provide isolation and controlled communications between applications in form of message passing.

Generic multi purpose operating systems perform all these tasks in privileged operation mode (kernel mode or supervisor mode) of the system. This is the monolithic kernel design approach which is used for instance in Microsoft Windows and Linux-based operating systems. This approach has some major issues concerning security. As all components are running in privileged mode, each subsystem of the kernel has access to all kernel-level data. This allows for instance to easily abuse the module/driver loading to instantiate a rootkit inside the operating system. Also resilience is a problem in the monolithic approach because a faulty driver can cause the whole system to crash. In total, the whole software complexity of this approach is hard to manage.

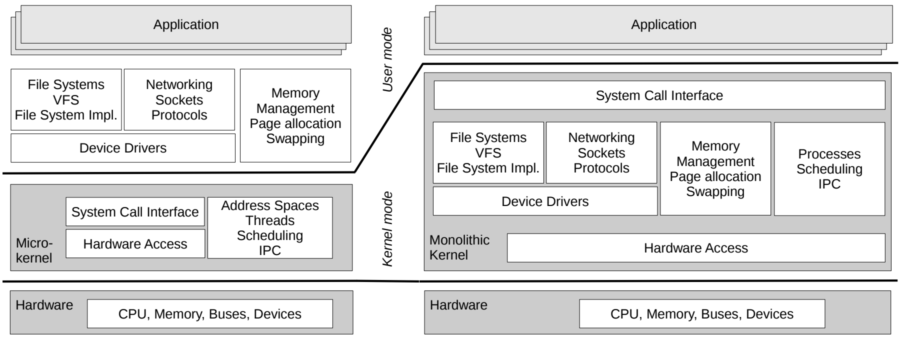


Fig. 2.8: Microkernel versus monolithic kernel design approach

The microkernel approach moves all the components of the operating system which do not need privileged instructions out of the kernel to non-privileged user mode. The software complexity of the kernel is reduced significantly by just running the essential parts inside the microkernel. However, as a result of moving all system services to user space, the amount of inter-process communication (IPC) raises significantly. Thus, the efficient implementation of IPC is of high importance. That is why early microkernel-based operating systems lacked performance in comparison to the monolithic approach.However, novel so called third generation microkernels like current kernels of the l4 family have fast IPC implementations and performance issues are of no concern anymore.

One of the most important responsibilities of a microkernel (besides handling IPC and scheduling Threads) is the separation of user space tasks. By strictly separating tasks with different levels of criticality, a microkernel-based system is able to provide a trusted runtime environment on the same hardware platform, where untrusted software is also executed at the same time. Similarly other virtualization-based approaches using a hypervisor provide task separation. However, conventional hypervisors need device drivers in privileged kernel mode with the consequence of a larger TCB. Figure 2.8 shows a comparison between the microkernel and monolithic operating system design approach.

The microkernel runs in privileged mode directly on the hardware. All non-essential system services, such as memory management (*paging*) or devices drivers, are executed in user space and are usually called *servers*, as they provide services or device access to other tasks (*clients*) through IPC. The memory management servers, however, are usually referred to as *pagers*. Similarly, all other services that are usually part of a monolithic kernel, for instance, file system drivers, are implemented as servers in user space. In addition to those basic mechanisms and components, the L4 kernel interface implementation Fiasco.OC [LW09] includes a capability system, which is used for securing the access to kernel objects. In our secure loader concept in Chapter 6, we make use of capabilities, for instance, when we establish communication channels between certain tasks. Besides the native microkernel applications, a modified rich operating system (rich OS) kernel, e.g., a para-virtualized Linux like L4Linux [HHL$^+$97], can also run as application in user space on top of the microkernel.

2.3.2 History of the L4 Family

Since we use microkernel systems based on L4 microkernels throughout this work and L4 is an expansive notation for several different microkernel application binary interfaces (ABIs) and implementations, we want to clarify the notation L4 in the following excursus.

Currently, there are 5 actively developed microkernels which have their root in the L4 microkernel design by Liedtke [Lie95] and thus are part of the L4 microkernel family. These are the seL4, OKL4, Nova, Fiasco.OC and PikeOS/P4.

As NOVA only supports x86 and the source of seL4 was not publicly available until 2014, at the beginning of our work on embedded systems, the latest available open source microkernel with modern design and ARM support was the Fiasco.OC. Further, as a commercial implementation of the "v2" ABI the P4 microkernel as part of PikeOS were available for this work due to research cooperations with the Sysgo AG. PikeOS is widely used in safety-critical avionics systems and therefore of high interest for us. The corresponding user-space runtime environments L4Re for the Fiasco.OC and PikeOS are described later when we discuss our architecture in Chapter 5.

Liedtke's first L4 microkernel was written in assembler for Intel i486 systems and later also Pentium at GMD. This version is referenced as the original L4/x86 kernel in the L4 community. The ABI version of the original implementation is "v2". There is no ABI v1 because Liedtke was experimenting with L4 concepts already in earlier work [Lie93a], [Lie93b] were he called the kernel L3. This lead to the "v2" ABI for which he finally introduced the wording L4. Form this ABI version "v2" three groups started research and development. An overview on the kernel ABIs and their implementations is provided in Figure 2.9.

System Architecture Group at the University of Karlsruhe The group around Liedtke in Karlsruhe continued his work and came up with the experimental ABI version X.0 and its implementation L4Ka:Hazelnut. This was the

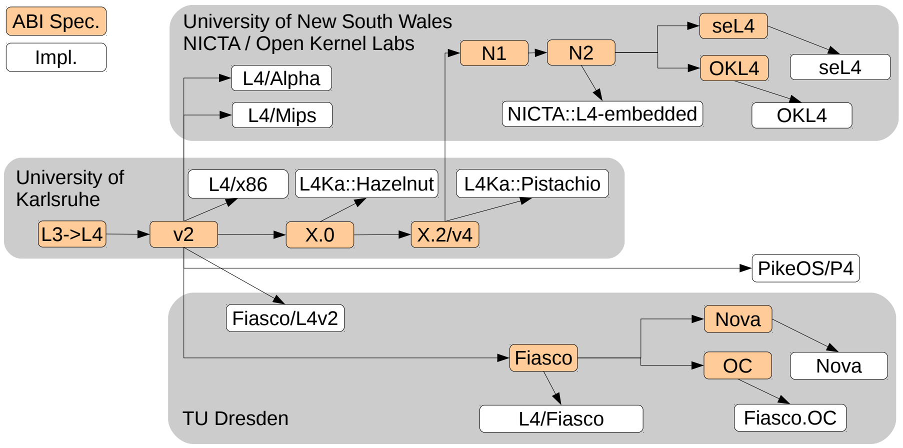


Fig. 2.9: Development on the L4 ABI specification and their implementations

their first portable implementation for different processor architectures (Pentium and ARM). Further research work of the group on improving portability and multi-processor support resulted in the major ABI Revision "v4" which is also referred to as "X.2". Their implementation called L4Ka::Pistachio closes their research work on this topic and further development at Karslruhe was discontinued.

University of New South Wales and NICTA The second research group around Gernot Heiser is located at the UNSW and NICTA and the spinout Open Kernel Labs. At UNSW/NICTA the "v2" ABI version was reimplemented for Alpha and Mips processors called L4/Alpha and L4/Mips. However, those implementations were discontinued and the group at UNSW/NICTA provided ports of L4Ka::Pistachio on Mips, Alpha and 64-bit Power-PC. Later they also re-targeted the L4Ka::Pistachio implementation for use in resource-constrained embedded systems, resulting in the "N1" ABI and the "N2" ABI. The corresponding implementation is called NICTA::Pistachio-embedded or L4-embedded. This version was used in commercial products of Qualcomm and the development continued in the "OKL4" ABI and implementation in the spinout Open Kernel Labs. The OKL4 Kernel in version 2.1 introduces capabilities for access control. With introduction of the OKL4 v3.0, Open Kernel Labs started with a proprietary licensing scheme. At UNSW/NICTA the research continues on the "seL4" ABI [KEH$^+$09] and implementation aiming for formal verification on security and safety. The source code, models and proves of seL4 are now publicly available under GPL.

Operating Systems Group at TU Dresden The third group in the L4 community is located at TU Dresden. The Operating Systems group around Herman Härtig also started with an own implementation of the "v2" ABI called Fiasco/L4v2. However, they soon started to continuously develop their own ABI called Fiasco resulting in the implementation L4/Fiasco. Later with the introduction of capabilities [LW09] similar to the OKL4 kernel Fiasco was renamed to Fiasco.OC. Further, the design NOVA [SK10] influenced by Fiasco for hardware-assisted virtualization on x86 platforms was started from scratch.

2.4 Cache-based Side Channels

In this thesis, as a main part we deal with cached-based side channels. We later demonstrate that our architecture as well as TEEs in general are prone to cache timing side channel attacks. To better understand the remarks in Chapter 7, we give now the corresponding background.

A cache timing attack exploits the cache architecture of modern CPUs in the following way. The cache architecture has influence on the timing behavior of each memory access. The timing depends on whether the addressed data is already loaded into the cache (cache-hit) or it is accessed for the first time (cache-miss). In case of a cache-miss, the CPU has to fetch the data from the main memory which causes a higher delay compared to a cache-hit where the data can be used directly from the much faster cache. Modern symmetric as well as asymmetric cipher algorithms usually make use of lookup tables. These tables hold precomputed data which is retrieved during cipher execution by key dependent lookups. This results in individual timing patterns for different keys for the same plaintext input due to specific collision during cipher execution.

2.4.1 Classification of Attacks

Current research work about Cache-based attacks can be divided into three different categories, each having a different attacker model:

1. *Time-driven* attacks
2. *Trace-driven* attacks
3. *Access-driven* attacks

Time-driven

Time-driven attacks [Ber05b, OST05, NSW06, ASK06, BM06] make use of the cache model in a very general way as they only require timing data of entire runs of a cryptographic algorithm, e.g., an encryption using AES. This corresponds to an attacker who has only very limited or coarse information about the cache.

Trace-driven

Trace-driven attacks [AK06, GKT11] additionally require detailed information about the cache activity during single runs of the encryption, in particular the sequence of cache hits and misses caused by the memory accesses performed by the encryption algorithm. A trace can for instance be captured by profiling the power consumption while the encryption routine is running. This translates to an attacker, who has gained a substantial level of knowledge about the runtime cache behavior which in case of a power profile also requires physical access to the device.

Access-driven

Finally, *access-driven* attacks [OST05, GBK11] assume to have knowledge about the cache-sets accessed by the algorithm. The underlying assumption is therefore that the attacker can control the cache runtime behavior. In the *Prime+Probe* attack [OST05], for example, those areas of the cache that also hold the lookup tables of the attacked algorithm are filled by a spy process with own data before the encryption is triggered (*Prime*). After the encryption, the spy process measures the access time to its own data to see which parts have been evicted from the cache by the encryption algorithm (*Probe*). Now the attacker can deduce which parts of the lookup tables were accessed by the encryption and from this infer some or all bits of the secret key.

For an in-depth background on cache-based side channels, we refer to [TOS10] which also introduces CPU cache architectures in general.

2.4.2 Applicability of Time-driven Attacks in Virtualization Context

As can be seen from the above explanations, time-driven attacks are the most widely applicable class of attacks since they do not require a strong attacker with fine grained access to the cache. Furthermore, for a time-driven attack, it is sufficient to see the attacked system as a black box which makes the attack portable to different CPU architectures.

Bernstein [Ber05b] for instance used this characteristic for a known plaintext attack to recover the secret key of an AES encryption on a remote server. However, Bernstein had to measure the timing on the attacked system to get rid of the noisy network channel between the attacked server and the attacking client. While this is a rather unrealistic scenario since the server needs to be modified, it is very relevant in the context of virtualization. In the context of virtualization, the noise is negligible since local communication channels are used for controlled inter-domain data exchange. These communication channels are based on shared memory mechanisms which introduce only a small and almost constant timing overhead.

2.4.3 Cryptographic Implementations

Exemplarily, we use the AES cipher to demonstrate vulnerabilities and countermeasures against cache-based side channel attacks. Nevertheless, also other symmetric ciphers such as DES and Clefia [RM11, ZW10] are vulnerable to this kind of attack as they also use lookup table-based implementations. We provide information about the original proposed SBox implementation as well as T-Tables implementation to better understand the demonstrated attack and countermeasures discussed in Chapter 7. For an in-depth understanding of AES, we refer to [DR02] as well as for an introduction to the mathematical basis namely the Galois-Field $GF(2^8)$, we refer to [Eck14].

AES SBox Implementation

As original proposed as Rijndael algorithm by Daemen et al. [DR02] and accepted by NIST as the AES standard [NIS01] in 2001, the AES cipher algorithm is a symmetric block cipher consisting of a key schedule and repeated round transformations. The NIST specifies the key length n = 16, 24 or 32 Byte, by a fixed block length $Nb = 128$ (bit). In this work, we stick to n = 16 Byte key length.

An input plaintext is then denoted as $p = \{p_0, ..., p_{15}\}$ and the AES key as $k = \{k_0, ..., k_{15}\}$ while p_i and k_i addresses the respective i^{th} Byte. The resulting ciphertext of an AES encryption is respectively denoted as $c = \{c_0, ..., c_{15}\}$. An AES encryption produces the ciphertext c and uses p and k as input as denoted in Equation 2.2

$$c = enc_{AES}(p, k) \tag{2.2}$$

After an initial key schedule the KeyExpansion, a set of round keys is generated as denoted in Equation 2.3,

$$K^r = \{K_0^r, ..., K_{15}^r\}, \text{for } r \in [0, 10] \tag{2.3}$$

while for the first round $r = 0$, it holds that

$$K^0 = k$$

This is important as we use this fact in the heuristic for the correlation in our timing attack.

Further, for 16 byte keys it is noteworthy to say that the KeyExpansion is reversible. This means an attacker who guesses any round key K^r correctly can compute any other round key and therefore also k.

The actual round operations are called *SubBytes, ShiftRows, MixColumns* and *AddRoundKey* and successfully applied to a state s which is denoted as $s^r = \{s_0^r, ..., s_{15}^r\}$ during a round r. After an initial *AddRoundKey*, for $n = 16$, 10 rounds of the just mentioned 4 operations are performed in exact this order, except in the last round the *MixColumns* step is omitted, resulting in the following high-level encryption algorithm 1.

Algorithm 1 AES high-level encryption

```
1: procedure $enc_{AES}(p, k)$
2:     K = KeyExpansion(k)
3:     $s^0$ = AddRoundKey(p, $K^0$)
4:     for $r$ in 1 to 10 do
5:         $s^r$ = SubBytes($s^{r-1}$)
6:         $s^r$ = ShiftRows($s^r$)
7:         if $r$ != 10 then
8:             $s^r$ = MixColumns($s^r$)
9:         $s^r$ = AddRoundKey($s^r$, $K^r$)
10:    $c = s^r$
11:    return $c$
```

For us it is not interesting what happens in detail in the operations themselves, except the *AddRoundKey* operation, which is nothing more than the bit-wise XOR of the corresponding input parameters.

$$AddRoundKey(s^r, K^r) \equiv s^r \oplus K^r \tag{2.4}$$

However, in *SubBytes* a non linear operation is performed which makes use of the AES Sbox S_{RD}. This SBox is realized as memory lookup table, which makes it vulnerable to cache-based timing attacks.

The decryption process is not relevant for this thesis, however for the sake of completeness it is described here. A decryption happens the same way while c and p are exchanged as input and output parameters denoted as

$$p = dec_{AES}(c, k) \tag{2.5}$$

Further, all operations are inversed in their order as well as the operations themselves, resulting in the straightforward decryption algorithm 2.

Algorithm 2 AES straightforward decryption

```
 1: procedure $dec_{AES}(c, k)$
 2:     K = KeyExpansion(k)
 3:     $s^{10} = c$
 4:     for $r$ in 10 downto 1 do
 5:         $s^r$ = AddRoundKey($s^r$, $K^r$)
 6:         if $r$ != 10 then
 7:             $s^r$ = InvMixColumns($s^r$)
 8:         $s^r$ = InvShiftRows($s^r$)
 9:         $s^{r-1}$ = InvSubBytes($s^r$)
10:     $p = s^0$
11:     return $p$
```

Especially, this mean that the SBox in *InvSubBytes* is exchanged with the inverse SBox S_{RD}^{-1}. Thus, attacks on the decryption process can be performed analogous to the encryption. However, it is not possible to attack the AES key k directly, but the round key K_{10}. Since the key schedule is reversible for 16 byte keys, it is feasible to compute k out of K_{10}.

AES T-Table Implementation

The T-Tables implementation is an efficient way to pre-compute the round operations in 4 lookup tables T_0, T_1, T_2, T_3 which consist of 256 32bit words, resulting in 1KB of size each. With those tables the state could be computed by 32bit word operations in form of lookups and bit-wise XOR operations. We denote the word representation of state s as

$$\mathbf{s}_i^r = \{s_{4i}^r, s_{4i+1}^r, s_{4i+2}^r, s_{4i+3}^r\}$$

and the word representation of the round key

$$\mathbf{K}_i^r = \{K_{4i}^r, K_{4i+1}^r, K_{4i+2}^r, K_{4i+3}^r\} \text{ for } i \in [0,3]$$

The resulting state transformation during a round r with $r \in [1,10]$ using 4 state words is then carried out as depicted in Equation 2.6.

$$\begin{aligned}
\mathbf{s}_0^r &= T_0[s_0^{r-1}] \oplus T_1[s_5^{r-1}] \oplus T_2[s_{10}^{r-1}] \oplus T_3[s_{15}^{r-1}] \oplus \mathbf{K}_0^r \\
\mathbf{s}_1^r &= T_0[s_4^{r-1}] \oplus T_1[s_9^{r-1}] \oplus T_2[s_{14}^{r-1}] \oplus T_3[s_3^{r-1}] \oplus \mathbf{K}_1^r \\
\mathbf{s}_2^r &= T_0[s_8^{r-1}] \oplus T_1[s_{13}^{r-1}] \oplus T_2[s_2^{r-1}] \oplus T_3[s_7^{r-1}] \oplus \mathbf{K}_2^r \\
\mathbf{s}_3^r &= T_0[s_{12}^{r-1}] \oplus T_1[s_1^{r-1}] \oplus T_2[s_6^{r-1}] \oplus T_3[s_{11}^{r-1}] \oplus \mathbf{K}_3^r
\end{aligned} \tag{2.6}$$

The initial state s^0 is just the bit-wise XOR of the plaintext with the first round key.

$$s^0 = p \oplus K^0 \Rightarrow s^0 = p \oplus k \tag{2.7}$$

For the final round a special Table T_4 is used which does not include the *MixColumn* operation.

The T-Tables are generated out of the original SBox $S = S_{RD}$ and its inverse $S^{-1} = S_{RD}^{-1}$ by the following concatenations:

$$\begin{aligned}
T_0 &= \{S^{-1}, S, S, (S \oplus S^{-1})\} \\
T_1 &= \{(S \oplus S^{-1}), S^{-1}, S, S\} \\
T_2 &= \{S, (S \oplus S^{-1}), S^{-1}, S\} \\
T_3 &= \{S, S, (S \oplus S^{-1}), S^{-1}\} \\
T_4 &= \{S, S, S, S\}
\end{aligned} \tag{2.8}$$

We do not describe the decryption with T-Tables as this is not necessary to understand the topics discussed in this theses. For further readings we refer to the book of Daemen et al. [DR02] and the AES standard [NIS01].

3

Related Work on Integrity and Secure Loading

In this chapter, we discuss related work on integrity verification approaches, hardware-based attestation mechanisms to prove the integrity to a remote verifier, and hypervisor-based techniques for protecting virtualized systems. We also denote how those work differ from our following microkernel-based approach.

Remember related work on cache-based side channels is provided separately in Chapter 7. This chapter covers architecture and trusted computing related work concerning the following Chapters 5 and 6.

3.1 Hardware-based Attestation Mechanisms

In the field of integrity verification, authenticated boot in combination with a TPM as specified by the TCG was one of the first approaches to establish a cryptographic set of integrity measurements for all components involved in the boot process as described in Chapter 2.1.4. Unfortunately, this approach does not include all components of the operating system and the applications, which are loaded during run-time. To overcome this limitation, Sailer et al. [SZJvD04] proposed the Integrity Measurement Architecture (IMA) for the Linux kernel (see Chapter 2.1.4), which measures the integrity of every binary that is executed. Jaeger et al. [JSS06] then extended IMA with a mechanism based on *SELinux*, which also verifies information flows at runtime. However, both approaches must be realized as part of the monolithic Linux kernel, whereas our integrity measurement components are implemented as unprivileged user space servers, which are strictly separated by a very small microkernel.

Based on the techniques for collecting integrity values, the concept of a remote attestation enables a system to prove its integrity to a remote verifier. In a traditional hash-based remote attestation as specified by the TCG [Tru11], the prover sends a set of aggregated integrity measurements signed by the TPM together with a Storage Measurement Log (SML) to a remote verifier, which can evaluate the SML and check the signed integrity measurements. Other schemes generalize this hash-based approach by proposing a property-based [SS04], semantic [HCF04], or logic-based attestation [SdBR$^+$11].

In total, research about TPM-based integrity protection has diverged into different directions. Parno et al. [PMP10] have conducted a comprehensive survey on this. For the interested reader, we refer to their work to get an complete overview on TPM-based integrity approaches.

3.2 Mobile Devices

For mobile devices, recent attestation concepts [BDH$^+$11, MMSS13, NKZS10] focus on providing verifiable proof for the integrity of mobile operating systems, such as Android, because of their increasing popularity.

Nauman et al. [NKZS10], for instance, propose a remote attestation for Android systems. To collect measurements, they extended the ClassLoader of the Dalvik VM, Android's process virtual machine [SN05, p. 38], which allows to execute Java binaries. However, for native applications that do not run in the Dalvik VM and the Dalvik binary itself, they still rely on IMA.

In contrast to our approach, those concepts cannot prevent loading of unknown binaries, since they only measure the integrity of binaries and attest the trustworthiness of the system afterwards. In terms of separation, those approaches mainly depend on the weak separation mechanisms within the Android kernel.

3.3 Virtualization Approaches

To improve isolation, virtualization-based approaches propose to separate the measurement code from the operating system kernel by implementing it in a privileged virtualization environment. Härtig et al. [HHF$^+$05] proposed the *Nizza* security architecture for microkernel-based systems. However, the focus of this theoretical concept lies in the separation of applications, the reduction of the TCB, and only briefly mentions a need for a TPM-based attestation.

Focus on mobile platform security by running Android on top of a microkernel is provided by Lange et al. [LLL$^+$11].

Other existing mechanisms [SVJ12, MLQ$^+$10, MPP$^+$08, BCG$^+$06, SPvD05] are either based on KVM or Xen for x86 architectures, a hypervisor, or a trusted execution environment utilizing hardware features of the x86 architecture. Schiffman et al. [SVJ12] proposed an attestation mechanism for virtualized systems, which includes a local representative (proxy) of the backend system. Similar to our local attestation approach which we will discuss in Chapter 6, this reduces the gap between time of measurement and the attestation. However, they do not focus on providing a minimal TCB for their proxy as they rely on KVM as their hypervisor utilizing IMA. Furthermore, none of the concepts deal with a TCB-reduced microkernel application scenario, such as in the following discussed secure offline payment use case, on an embedded ARM platform with a real TPM chip for measurements combined by a runtime loading mechanism for external binaries.

4

Scenarios and Attacker Model

In this chapter, we want to state the main scenario denoted as *Secure Offline Payment* which motivates a microkernel-based security architecture for loading and verifying integrity of a remote binary into a trusted runtime as we will propose in Chapter 6. Further, we will introduce standard scenarios for TEEs as specified by the GlobalPlatform (see Chapter 2.2.1) in general, which also directly apply to our herein proposed architecture. We identify a major threat to the separation by means of those scenarios, namely a cache-based side channel which may allow to extract, e.g., symmetric session or master keys. Based on these scenarios, we specify our coarse attacker model.

4.1 Scenario: Secure Offline Payment

For our secure loading mechanism, which assures the authenticity of a remote binary and measures the integrity to be able to provide verifiable proof for a remote attestation, we assume a scenario we refer to as *Secure Offline Payment*.

In our scenario, we have two parties, a payment service provider (e.g., a bank) and a customer who is using, for example, a smartphone with our microkernel-based system architecture. We presume that the customer's microkernel-based system is equipped with a hardware-based security module, such as a smart card or a TPM, which can be used to provide secure storage. The payment provider on the other hand is a remote party, which provides secure payment services by sending a trusted application that handles the (offline) payment transactions, to its customers' systems. The trusted application is protected with a device-specific key stored inside the SE and transferred to the corresponding device, which the customer has registered to the service provider in an initial setup procedure. After a verification of its authenticity and integrity (via remote attestation), the trusted application handles the offline payments and securely stores the transaction data in an application-specific secure storage. As soon as the device is connected to the Internet, the trusted application synchronizes all offline payments with the backend after providing proof via attestation that the system including the trusted application is still trustworthy.

The trusted payment application may also include a verification service component (also referred to as *challenger*), which can verify attestation results locally. The challenger then effectively represents the remote verifier on the local system as proposed in [SVJ12]. As indicated, that allows for a local verification of the attestation results and only requires one traditional remote attestation (for the system with the challenger). After a successful remote attestation, the challenger can verify attestation results on behalf of the remote verifier. As a result, the effects of the time of check to time of use (TOCTOU) problem can be reduced significantly.

TOCTOU is the generic problem definition for a race condition between the result of a condition check in our case the attestation result and the later use of the result, the prove against the valid system states on the remote system. An attacker could use this time interval to alter the system state and perform malicious operations undetected. Hence, it is desirable to keep this time window as small as possible.

4.2 Generic TEE Usage Scenarios

We now state general valid usage scenarios of TEEs, which assumes an attacker is restricted to the rich environment. Assets inside the trusted environment are protected by separation throught the virtualization layer. These scenarios should also enable a deeper understanding why it is reasonable to make use of a TEE to improve security in comparison to directly running all services inside one operating system. Attacking these kinds of usage scenarios, we address mainly in Chapter 7.

4.2.1 Scenario: TEE Secure VPN

We assume a TEE which separates two compartments, a trusted environment which provides crypto services and an untrusted environment which runs user applications. The secret keys used for encryption are highly critical values and

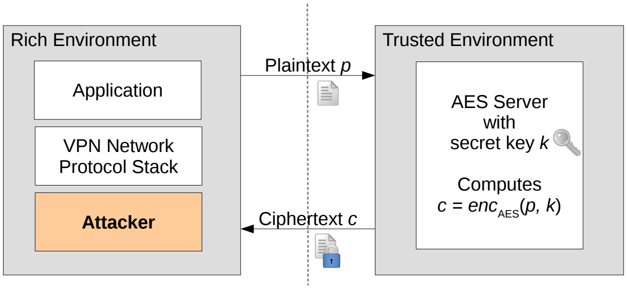

Rich Environment
Application
VPN Network Protocol Stack
Attacker
Plaintext p
Ciphertext c
Trusted Environment
AES Server with secret key k
Computes c = enc_AES(p, k)

Fig. 4.1: Secure VPN using trusted AES server to protect the key k from an attacker in the rich environment

thus are only accessible in the trusted environment. A viable usage scenario is, e.g., to establish a VPN tunnel in the following way. The network protocol stacks of a rich operating system kernel are used in the untrusted environment while the payload is encrypted by a driver using an encryption service inside the trusted environment. Hence, secret session keys as well as globally used master keys cannot be compromised by an attacker in the rich OS. For instance if the VPN key is shared between all company devices and only those devices are allowed to enter the company network, one compromised device will allow to connect any device to the VPN afterwards.

A concrete attack scenario could then look as follows. An AES encryption server runs in the trusted environment. To launch an encryption, a user application or directly a virtual driver inside the rich environments OS kernel simply stores the plaintext in shared memory and calls the AES server through IPC. The ciphertext is then written back to the shared memory. This scenario is visualized in Figure 4.1. Usually, the TEE is secure against such an attack. We derive an attacker model which shows the contrary.

4.2.2 Other Generic Scenarios Using Crypto Services

The AES server can also be used for several encryption services like on-the-fly storage encryption. Similar to the VPN setup, the symmetric master key for the actual file system partition is then only available inside the trusted environment. Another scenario is transparent voice encryption which can make use of the protected crypto server. The actual audio stream de- and encoding is done in the rich environment while the resulting audio packages can be en- and decrypted by the AES server inside the trusted environment.

4.3 Attacker Model

A general attacker model based on the TEE scenarios (Secure VPN and generic Crypto Services) as well as our *Secure Offline Payment* denotes as follows. We assume a remote attacker, who is able to read, drop, and manipulate messages if they are unprotected and sent via an unsecured communication channel. The attacker can also initiate communication, create new messages and try to replay old ones. However, we assume that the attacker is not capable of breaking cryptographic security functions, such as a state-of-the-art encryption, message authentication codes, or cryptographic hash functions. That means the attacker is not able to decrypt an encrypted message or forge a message authentication code for a modified message without knowing the correct key. In addition, the attacker might be able to compromise the rich OS by exploiting its software components. However, the attacker is not able to modify hardware components, which would require physical access to the device. As a consequence, we can assume that removing hardware components, such as the TPM or a SD memory card, is not possible.

Furthermore, in the VPN scenario, an attacker wants to determine the key used by the AES server. In the secure offline payment scenario, he wants to determine the authentication key or a symmetric session key. Both of them are assets inside a trusted component inside the trusted environment. As he has full access to the rich OS in the untrusted environment, he is able to launch as many encryptions as he likes with chosen plaintexts. This he could do either by hijacking running processes or deploying own code that directly uses the kernel of the rich OS. The attacker is therefore able to launch a time-driven cache attack, which we will discuss in detail in Chapter 7. Note that we provide a more detailed attacker model for this kind of attacks within that chapter.

5 Microkernel-based System Architecture

Before we get down to our proposal for integrity and secure loading for generic embedded systems, we introduce our microkernel-based system architecture in a general way. Inside this chapter, we mainly address our goal of a flexible and generically applicable security architecture for both mobile and embedded systems in general. We will discuss how we can use the security approaches of the mobile device focused GlobalPlatform and the trusted computing approaches which are mainly adopted in x86-based servers or personal computers, as building blocks for our architecture to achieve that goal. Further, we provide architectural measures to reduce the TCB on kernel level as well as for the integrity components themselves. These measures will then be quantified by an overall evaluation in Chapter 6. After introducing and identifying those building blocks, we contemplate in more detail how each of them contribute to our virtualization-based system architecture using a TPM as trust anchor.

Note that parts of the herein proposed system architecture were published in [WHS12, WWHW14, WWAS15, PWS15].

5.1 Combining GlobalPlatform with Trusted Computing Approaches

We combine both the hardware and software components of the GlobalPlatform and trusted computing into our architecture as shown in Figure 5.1. We identify the following four major components as building blocks for our architecture.

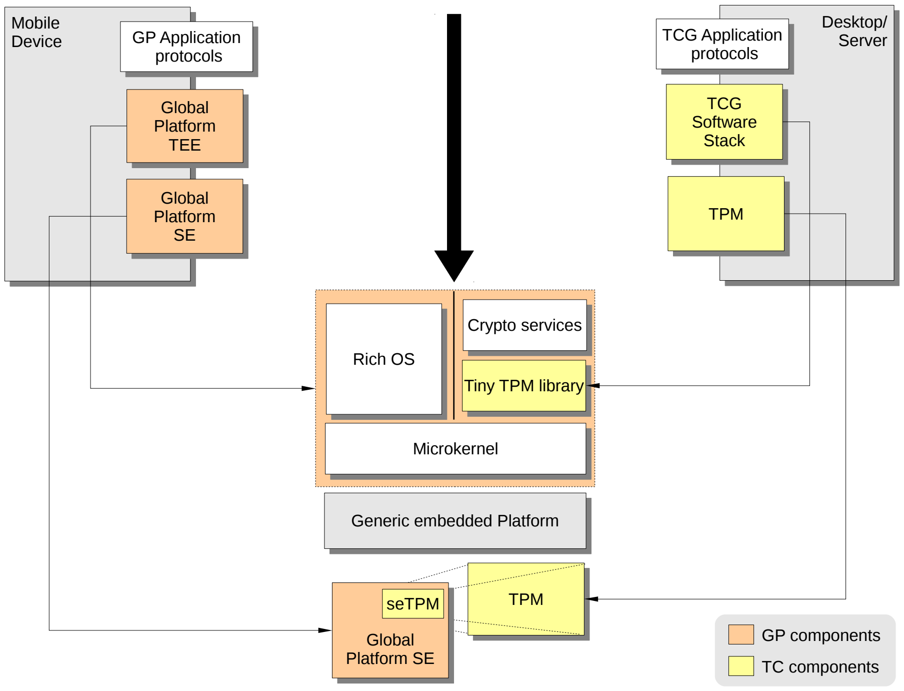


Fig. 5.1: Combining building blocks of GlobalPlatform and trusted computing to a system architecture for generic embedded platforms

1. Trusted Execution Environment (TEE)
2. Secure Element (SE)
3. TCG Software Stack (TSS)
4. Trusted Platform Module (TPM)

Hardware Components The GlobalPlatform defines hardware assisted generic application protocols using a so called SE. This is usually a Java Card based smart card as described in Chapter 2.2 and is programmable for specific protocols. The TCG however defines the TPM very specific for several application protocols, e.g., remote attestation. Thus, our architecture in a first step provides a TPM as well as a secure element side by side on an embedded platform. In a second step as the GlobalPlatform SE provides the ability to program the firmware of the SE for specific application protocols, we can move the hardware TPM inside the SE by instantiating an SE application *seTPM* which implements the TPM behaviour inside the GP SE. Thus, we are able

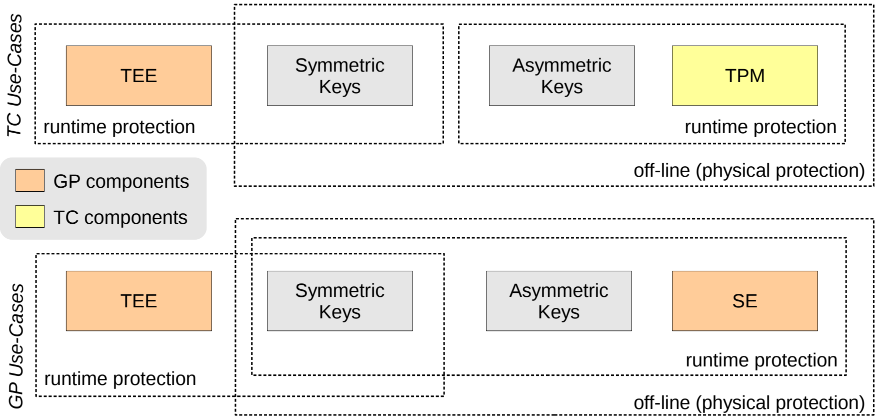


Fig. 5.2: Key protection for different use-cases

to run trusted computing protocols on mobile phone devices, as those already have the SE either integrated or can be upgraded by an additional SE. In contrast to a generic embedded device, to which also a physical TPM chip could be added as we show with our prototype implementation in Chapter 6.

Software Components The above described components are only the hardware parts of the particular group. However, the software parts are different. The GlobalPlatform describes a so called Trusted Execution environment which is kind of a virtualization-based separation architecture to divide security critical parts of applications from a rich operating system. A detailed description about the TEE was given in Chapter 2.2.1. We realize this idea with a microkernel-based operating system framework. The TCG defines the TSS, which is in the x86 world rather large software stack and contradicts the idea to keep the TCB small. Thus, we compress the TSS to a small TPM library which is running directly on top of the microkernel separated from the rich operating system. Thus, we are able to run trusted computing based applications inside the trusted execution environment, see Chapter 6.

5.2 Virtualization-based Security Architecture

Virtualization techniques can be used to provide strong isolation of execution environments and thus enable the construction of compartments. One compartment can then be used to execute sensitive transactions while the other compartment is used for transactions with a lower trust level. This design process is already partly employed by smartphone architectures. The former Dalvik VM on Android Version less then 5.0, or the Art VM on current Android versions provides some sort of process virtualization [SN05, p. 83], however, without providing the same level of isolation achieved by system virtualization [SN05, p. 369]. Due to the insecurity of current smartphones' and other embedded systems' architectures, virtualization solutions based on either full system virtualization or light-weight solutions of separate execution environments such as TEE implementations are more and more used to increase security and reliability. Furthermore, the TPM does not export any symmetric cryptographic functionality to the host system. This means symmetric session keys are always accessible in RAM of the operating system. Compared to our attacker model in Chapter 4, even if we use the secure storage of the TPM to protect a symmetric key, this key is vulnerable during runtime of the system. Thus, we also decided to use a TEE approach as fundamental building block of our system architecture. This way we can assure off-line (physical) protection of asymmetric and symmetric keys provided by the TPM as well as runtime protection for asymmetric keys. The missing runtime protection for symmetric keys is then provided by the isolation mechanism of the TEE. Further, for GlobalPlatform based applications also the SE can be used to protect symmetric and asymmetric keys during runtime. This is visualized in Figure 5.2.

5.2.1 Trusted Execution Environment

In contrast to the suggestion of GlobalPlatform to use a separate TEE kernel inside the trusted environment, we propose an architecture based on only one microkernel running as separation layer between execution domains as well as the trusted applications. In other words, we do not have an explicit TEE kernel, but merge the separation layer with it. Instead of running the crypto services and secure devices drivers inside the TEE kernel, we make use of the microkernel feature and run those as separated user space applications directly on top of the microkernel. The rich environment is encapsulated inside a user space para-virtualized rich OS kernel. This is visualized in Figure 5.3.

In our architecture, we also 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, or UDP, inside the trusted runtime environment. We design a component called communication server (Com Server) in the trusted environment and a communication client (Com Client) as virtual device driver inside the rich OS to provide network

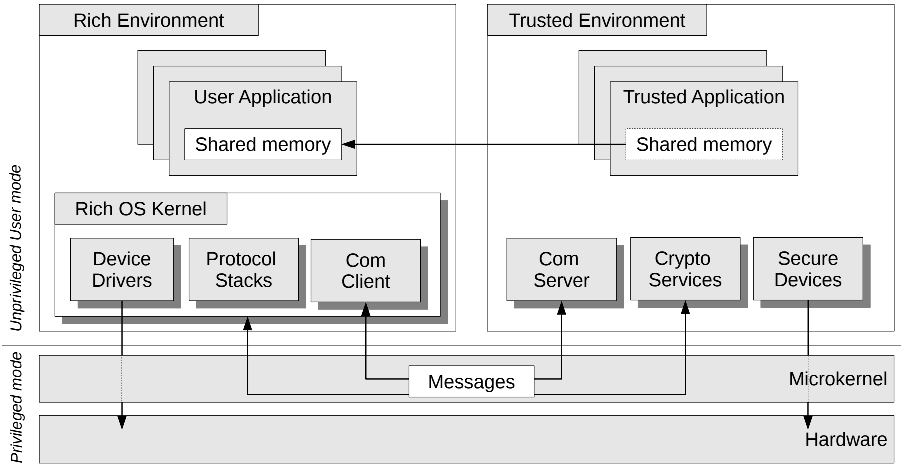


Fig. 5.3: Microkernel-based trusted execution environment

connectivity for trusted applications inside the trusted runtime as shown in Figure 5.3. Remember compared to our attacker model in Chapter 4, the communication is untrusted and the trusted application needs to run protocols which ensure authenticity and confidentiality themselves.

Throughout this work, we show the realization of our system architecture, in two concrete microkernel frameworks, PikeOS and Fiasco.OC/L4Re. Their particular underlaying kernels are part of the L4 family as described in Chapter 2.3.2.

L4Re Components as Trusted Runtime

The runtime environment for the Fiasco.OC microkernel, which is known as L4Re [LW09], mainly consists of a root task (*MOE*), a root pager (*Sigma0*), and an init process (*Ned*). Those main tasks establish a trusted runtime environment, allow for executing native microkernel tasks, and also provide the basis for running a rich operating system.

The root task, MOE, which is the first task started by the kernel, inherits all resources that are not occupied by the microkernel (Fiasco.OC) at startup. The memory management, however, is delegated to Sigma0, which handles all memory that is not used by the microkernel and acts as the root pager for the entire system. The *init* process, Ned, is able to start other applications by loading a runtime binary (*l4re_kernel* in Figure 5.4 denoted as *l4re*) in a new application context. After it is switched to the newly created context, the loader code of l4re_kernel loads the *actual* application binary in its context. With this approach, each new application has the L4Re runtime code for communication and memory abstractions, denoted as dataspaces, mapped into its virtual memory. This aspect is also very important for our secure loading approach described later on in Chapter 6. Ned also provides the ability to configure IPC between applications, including the capability settings for the corresponding access rights inside the Fiasco.OC microkernel. Additionally, to these L4Re standard components we need to realize the Com Driver inside L4Linux and the Com Server as well as the crypto services inside the trusted environment as L4Servers and L4Clients the high-level user space abstraction for IPC of L4Re. We discuss this in more detail when we describe our prototype implementation in Chapter 6.2. The signaling of new data is provided by this mechanism while the payload data are delivered over shared memory (*smem*). Similarly the communication between crypto services such as an AES server and the L4Linux kernel is handled. Remember the TEE usage scenario secure VPN. An AES driver inside the L4Linux kernel uses the AES server inside the trusted environment. The signaling is done over synchronous IPC while the payload – plain and cipher text of network packets – are written and read from shared memory.

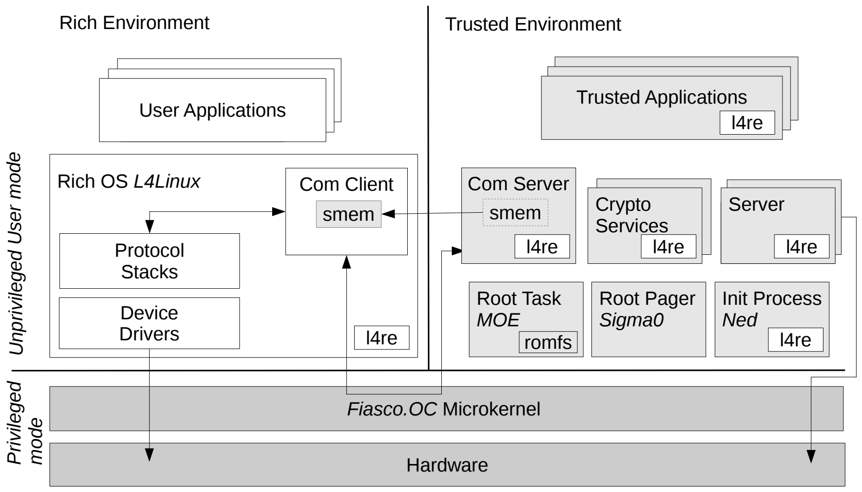


Fig. 5.4: Microkernel-based TEE realized with Fiasco.OC/L4Re

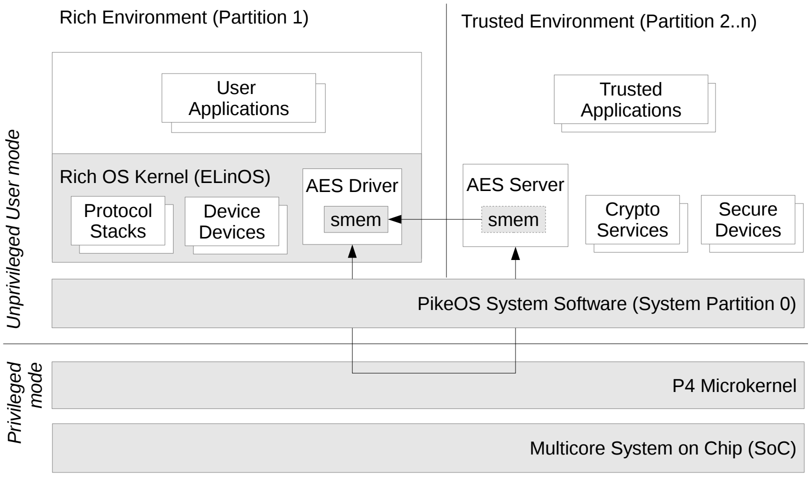


Fig. 5.5: TEE adapted to PikeOS

Using PikeOS to realize a TEE

PikeOS distinguishes between resource and time partitions. A resource partition in PikeOS denotes a separate address space protected by the microkernel, while time partitions are used to assign computation time to threads. The P4 microkernel itself only implements the basic mechanism for IPC, scheduling, and separation of address spaces in privileged processor mode. Device drivers, higher level abstraction for inter-partition communication as well as virtual memory management are implemented in the user-space abstraction layer, called PikeOS System Software (PSSW). Native device drivers for secure devices can be implemented in their own partition also in user-space. Figure 5.5 illustrates this architecture. In our context, the architecture comprises a rich environment which runs the untrusted user applications in one partition as well as a trusted environment that hosts the security and safety relevant trusted applications each in their own partition. Both environments are allowed to communicate with each other using protocol messages transmitted via the virtualization layer, which in that case is the P4 microkernel and its user-space abstraction layer PSSW. To exchange data between the trusted and untrusted applications, shared memory is used. The user applications may use the trusted applications via special device drivers integrated into the rich OS kernel. This is visualized in Figure 5.5 by the exemplary trusted crypto service AES server and the AES driver, its corresponding counter part in the rich OS Kernel.

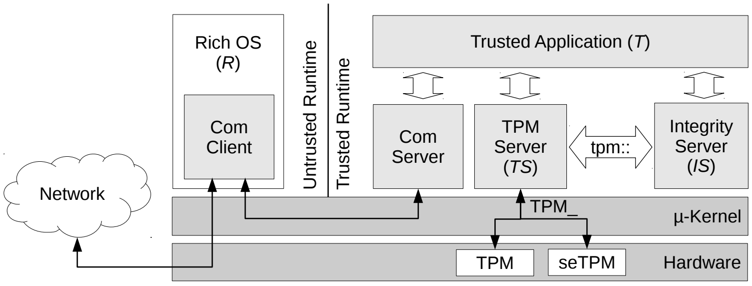


Fig. 5.6: High-level architecture containing trusted computing components

5.3 Trusted Computing

Since we just have constructed a trusted execution environment which provides separate execution domains by the use of a microkernel, we also have to assure to establish trust into one of these domains by further means. Hence, we integrate the trusted computing building blocks, TPM and TSS as described in Section 5.1 into this architecture. To be able to run trusted computing based protocols inside the trusted environment, as a first step, we have to add support for a TPM or the already mentioned *seTPM*. As both are API compatible from a software and architecture perspective our architecture allows to use either of which. The only point in which they differ are the low-level device driver. A visualization of the TPM integration in our architecture is provided in Figure 5.6. We use a user-space component, the TPM Server (TS) which provides an TPM abstraction (tpm::) for other client components. This includes our tiny TPM library as well as the low-level device drivers. As a client application making use of the TPM abstraction we design a separate component the Integrity Server (IS). This is later used to implement our integrity and attestation protocols in Chapter 6.

5.3.1 Tiny TPM Library

Figure 5.7 shows the mapping from the TSS layers to our user-space TPM server. The critical kernel level driver is moved into user mode directly inside the TPM server. This makes the separate TDDL obsolete.

Further for the TPM server outer interface, which can be used for instance by the *IS*, we define a very small API which consists of only the following commands: *Pcr_read*, *Extend*, *Quote*, *CreateWrapKey*, *MakeIdentity*, *LoadKey2*, *Seal* and *Unseal*.

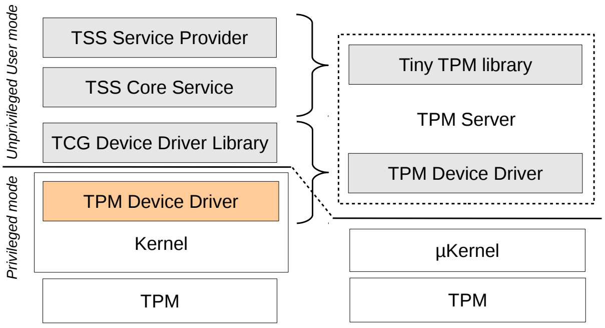


Fig. 5.7: TSS mapping to our microkernel-based framework

We specify that all of these commands are directly mapped to the corresponding TPM commands in part 3 of the specification. Complex session handling like OSAP should be completely hidden inside the TPM server.

The lower API between TPM server and the TPM is specified by the TPM command specification. However, we only need a subset to allow our concepts for secure boot, integrity and attestation to work. We specify the following subset of commands as sufficient for our embedded platform, in contrast to the TCG specified concepts for several use cases which are applicable for PC and server platforms. For instance, we do not need to migrate keys as we provide a special provisioning phase for our integrity and attestation protocols which we will discuss in Chapter 6.1.2. Therefore we can strip down those kind of commands and reduce the TCB even further. Due to this provisioning phase, we are also able to use the result of MakeIdentity to extract and sign the AIK's public key directly and do not need the `TPM_ActivateIdentity` command for this purpose. Our reduced TPM command specification and mapping to the high-level TPM server API (tpm::) is shown in Table 5.1. The first column provides the TPM command ordinal as defined in the TPM command specification. The second column shows the mapping to the high-level API which has to be provided by the TPM server if its prepended with "tpm::". Otherwise the second column shows the general purpose for which it is needed; *bootloader* means that those commands are only needed during boot, which is described in the next section. *initial setup* describes commands which only are needed once when the platform is personalized. Those commands could be either executed by the bootloader or during the already mentioned provisioning phase. *internally used* denotes commands which are usually not used directly, but are needed to provide, e.g., secure session initialization to instantiate preconditions for other commands of which the execution would be

Table 5.1: Reduced TPM command specification

TPM Command	**TPM Server (tpm::) / Purpose**
TPM_ContinueSelfTest	bootloader
TPM_CreateWrapKey	tpm::CreateWrapKey
TPM_Extend	tpm::Extend
TPM_GetCapability	bootloader
TPM_GetRandom	internally used
TPM_LoadKey2	tpm::LoadKey2
TPM_MakeIdentity	tpm::MakeIdentity
TPM_NV_DefineSpace	bootloader
TPM_NV_ReadValue	bootloader
TPM_NV_ReadValueAuth	bootloader
TPM_NV_WriteValue	bootloader
TPM_NV_WriteValueAuth	bootloader
TPM_OIAP	internally used
TPM_OSAP	internally used
TPM_OwnerClear	initial setup
TPM_PCRRead	tpm::Pcr_read
TPM_PhysicalEnable	initial setup
TPM_PhysicalSetDeactivate	bootloader
TPM_ReadPubek	initial setup
TPM_ResetLockValue	initial setup
TPM_Seal	tpm::Seal
TPM_SelftestFull	bootloader
TPM_Startup	bootloader
TPM_TakeOwnership	initial setup
TPM_Quote	tpm::Quote
TPM_Unseal	tpm::Unseal
TSC_PhysicalPresence	initial setup

denied by the TPM firmware otherwise. We do not provide further details on the TPM commands now as this is done in Chapter 6. There, we also show the feasibility of the API mapping and that the reduced command set is indeed sufficient by a prototype implementation based on L4Re.

As a consequence, the TCB for trusted computing based applications is reduced significantly in comparison to a full blown TSS like The open-source TCG Software Stack (TrouSerS) which is used in server or desktop environments, see the code size evaluation of our prototype implementation in Chapter 6.3. Concerns about the security are handled by an informal security analyses as part of our evaluation in that Chapter, too.

5.3.2 Establishing Trust

Further, to establish trust, we need a secure or trusted boot. On embedded devices, usually a multi stage boot loader mechanism is deployed. Usually there is only a small amount of SRAM which is set as starting point in the SoC's ROM code which initiates the boot. This SRAM retains only a very small loader which contains just a few rudimentary device drivers to access and execute the second stage bootloader which resides in flash or external storage. The second stage bootloader then initializes the rest of the hardware needed to boot the Operating system. We propose a TPM based secure boot, which extends the ROM as CRTM. A TPM-based trusted boot as proposed by the TCG usually looks as follows. Each component in the boot process measures the next component before it executes that component. The measured SHA1 hash is extended into the next free PCR register of the TPM (see Chapter 2.1.4). Ideally, the first stage bootloader is measured by the boot ROM. The problem is that it is not possible to alter the SoC's ROM code after production. Thus, if the vendor does not provide a fuse based secure boot, it is not possible to sign the bootloader code with an own key. However, if the boot order of the device can be configured to start from another read only memory, one could use this to replace the SoC's internal ROM as CRTM. For instance, the beagle board provides hardware means to configure the boot order to start from SD card instead of internal flash. We propose to place the first bootloader to a write once read many WROM-SD card in that case. Thus, the SD card as an external ROM containing the first stage bootloader forms the CRTM. Under the assumption that the attacker has no physical access to the hardware as we stated in our attacker model (Chapter 4), this in fact provides a CRTM as the attacker cannot alter the boot chain. Each stage of the bootloader has to include the TPM library which we specified above. Sharing this code with the TPM server the TCB can be reduced even more.

During an initialization phase, the first stage bootloader takes ownership (*TPM_ TakeOwnership*) of the TPM to insert the authentication data for owner authorized commands and to create the SRK. Further, it initializes space in the non-volatile memory using `TPM_NV_DefineSpace` and places a random value as secret inside that area. The size of the secret is the same as the block size of the used hashing algorithm which in case of SHA1 is 20 byte. We later use this secret to protect the reference values (hashes) of the bootloader binaries. The TPM allows to specify the data area to be writable only once, thus we are able to protect the secret of being overwritten by anyone after initialization. Further, to make sure an attacker may not be able to update the reference values after boot, the last stage bootloader has to lock the area inside non-volatile memory containing the secret.

Algorithm 3 Secure boot algorithm – normal boot

```
 1: procedure secure_boot
 2:     $PCR_0 = \texttt{TPM\_Extend}(PCR_0, bin\_current'_1)$
 3:     for $i$ in 1 to $n$ do
 4:         $bin\_current'_{i+1} = h(h(bin\_current_{i+1})||secret)$
 5:         $PCR_i = \texttt{TPM\_Extend}(PCR_{i+1}, bin\_current'_{i+1})$
 6:         $(error, bin'_{i+1}) = \texttt{TPM\_Unseal}((PCR_0, ..., PCR_i), K_{dec}, blob(bin'_{i+1}))$
 7:         if $error \neq \texttt{TPM\_SUCCESS}$ then
 8:             halt
 9:         if $bin'_{i+1} \neq bin\_current'_{i+1}$ then
10:             halt
11:         if $i = n$ then
12:             lock NV_area
13:         $exec(bin_{i+1})$;
```

The reference values bin' are sealed to the corresponding platform state during each stage i of the boot process as depicted in Equation 5.1.

$$\begin{aligned} &bin'_{i+1} = h(h(bin_{i+1})||secret); \\ &PCR_i = \texttt{TPM\_Extend}(PCR_i, bin'_{i+1})); \qquad \forall i \in [1..n] \\ &blob_{i+1} = \texttt{TPM\_Seal}((PCR_0, ..., PCR_i), K_{enc}, bin'_{i+1}) \end{aligned} \tag{5.1}$$

The reference values can be sealed by any storage key K_{enc} loaded into the TPM. However, due to limited space in first stage bootloaders, we recommend to use the SRK which is directly available after the TPM is owned with `TPM_TakeOwbership`. As the first stage bootloader is part of the CRTM in our case, it has to measure itself (bin_1) and the next stage bootloader binary. Therefore, it additionally performs stage zero (Equation 5.2):

$$\begin{aligned} &bin'_1 = h(h(bin_1)||secret); \\ &PCR_0 = \texttt{TPM\_Extend}(PCR_0, bin'_1); \\ &blob_1 = \texttt{TPM\_Seal}((PCR_0), K_{enc}, bin'_1) \end{aligned} \tag{5.2}$$

During normal boot operation, Algorithm 3 then checks the integrity of the current binaries in each boot stage and stops execution if an integrity violation due to either compromised sealed blobs or exchanged binaries is detected. Again note that stage zero is performed in advance to stage 1 inside the first stage bootloader.

In each stage, the current platform state is extended by the hash of the next stage binary ($bin\_current'_{i+1}$). Then the actual stage tries to unseal the reference value bin'_{i+1}. This only works if the platform state represents the same state as used for the generation of the sealed blob. However, this is not enough to guarantee a trusted platform state, we still have to compare the hash values contained in the sealed blob with the currently calculated ones.

As mentioned before, we protect the sealed reference values with the secret. This is concatenated to the corresponding binary's hash value and sealed to the corresponding platform state. Thus, an attacker may not be able to generate a correct hash chain as the secret is locked after secure boot procedure during runtime. Thus, in case the sealed blob was exchanged during runtime, at the next boot the check in the corresponding boot stage will fail due to mismatch of the corresponding hash chains. This is necessary as an attacker might be able to use the SRK or any storage key which resides in persistent flash memory to seal new hash values. An overview of the TPM-based secure boot approach for an embedded SoC is given by Figure 5.8. A concrete realization of this procedure for a two stage bootloader including a prototype implementation and security analyses using the ARM Cortex-A8-based BeagleBoard (see also Chapter 6.2) is provided by Lorenz [Lor12].

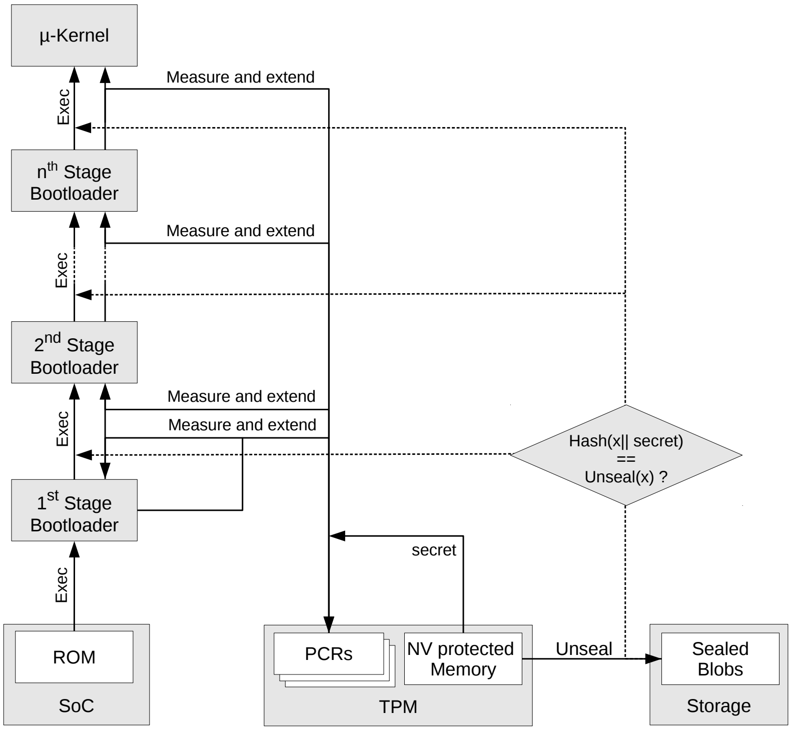


Fig. 5.8: TPM-based secure boot approach for embedded SoCs

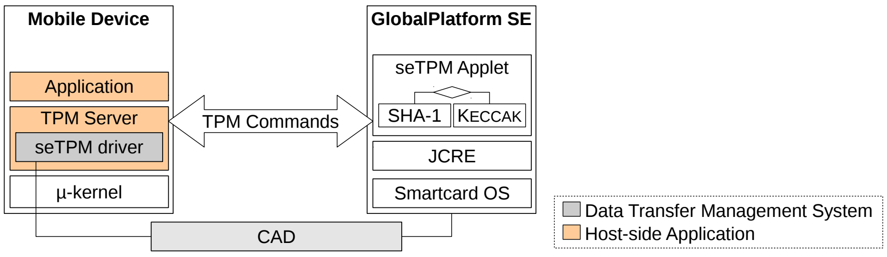


Fig. 5.9: Architecture of seTPM

5.4 GP Secure Element as TPM

In the following, we sketch the idea of replacing the TPM by a Java Card secure element. This allows us to transparently apply our herein proposed architecture directly to mobile devices on which we cannot alter hardware, meaning adding an additional TPM chip. Note an evolved version of seTPM with TPM 2.0 abilities is published in [PWS15]. Due to the obtained flexibility by the Java Card design, it is also feasible to switch crypto and hashing algorithms. For instance, we are able to replace the outdated SHA1 hash engine by a Java Card implementation of the KECCAK (SHA3) algorithm.

The resulting architecture adaption to a mobile device is shown in Figure 5.9. Our architecture comprises a mobile device or generic embedded device which runs a trusted execution environment as described above.

Further, the architecture comprises the data transfer management system which includes the Card Acceptance Device (CAD) and the corresponding seTPM driver. The seTPM device driver is replacing the TPM device driver in our TPM server. The CAD connects the host system with the Java Card secure element. This could be either a conventional Smart Card/RFID reader which is connected over serial or USB to the host system or if the secure element is embedded in a microSD card a simple microSD card slot.

Finally, the main part of the architecture is inside the GlobalPlatform SE represented by the seTPM Applet. This Applet needs to implement a TPM command interpreter and emulate the TPM commands as specified by the TCG. For our concepts, it is sufficient that the commands from Table 5.1 are emulated inside the seTPM Applet. This reduces code size of the Applet in contrast to a full TPM specification. Since current smart card hardware have limited space for executable code due to expensive EEPROM, it is advisable to save code size.

5.4.1 Communication

A further aspect of the seTPM architecture is the communication between TPM server and the seTPM Applet. At an abstract communication layer the TPM server exchanges TPM commands with the seTPM Applet. However, the Java Card framework uses APDU communication between host and the smartcard. Thus, at first we provide a mapping or an encapsulation of the TPM specified communication to the ISO 7816-4 APDU message protocol. In order to facilitate the interpretation of the APDU commands inside the seTPM main processing loop, we map individual TPM request meta data to the header of the command APDU as shown in Table 5.2. We could not just pack each TPM command and response as payload in a command or response APDUs due to size limitations. A command APDU can only handle 255 byte of payload (`MAX_APDU_PAYLOAD`) because the command APDU entry `Lc` which represents the APDU size is only one byte. On the other side, the TPM specification includes TPM command sizes up to 4KByte. As a result, we use chaining of APDUs to deliver a full TPM command or response in several chained command or response APDUs.

In case a TPM request exceeds this size, the APDU entry `CLA` signalizes data to follow with a set *chaining bit*. The chaining bit is the least significant bit of the `CLA` header entry. Chained TPM requests reuse the header of the first transmitted command APDU. The end of a TPM request chain is identified by a clear chaining bit. Besides, only two bytes (MSB and LSB) of ordinals within TPM requests carry significant information. Because of that and restricted space in the APDU header, we only map those two bytes and discard the remaining bytes. Note that the full TPM command header is included in the payload anyway.

As response APDUs do not include a header we just pack the response into (if necessary chained) response APDUs. The status word is used to signal the data management system about additional data. The corresponding driver of the CAD is then responsible to reconstruct the full message.

Table 5.2: seTPM APDU header definition

APDU Header	TPM Request Header Parameter
CLA	`0xB0 | CLA xor 0x01`
INS	`TAG[0]`
P1	`ORDINAL[3]`
P2	`ORDINAL[0]`
Lc	`PARAMSIZE[0] | MAX_APDU_PAYLOAD`

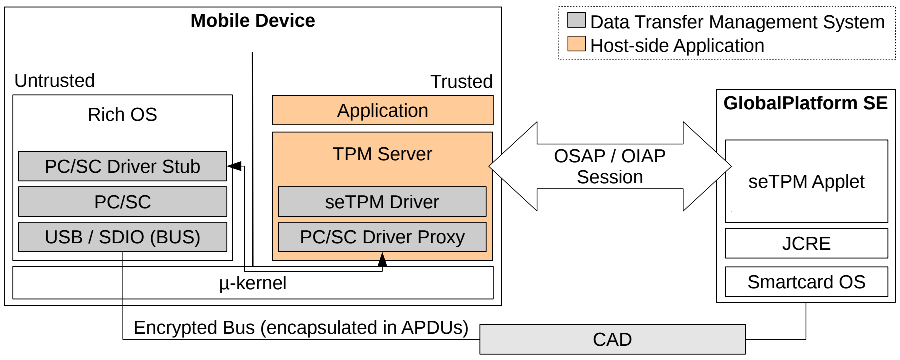


Fig. 5.10: Transparent data transfer management system using PC/SC

5.4.2 Transparent Data Transfer Management System

As we already mentioned the data transfer management system is responsible to deliver the payload of the APDU to the application or in our case the abstraction layer (tiny TPM library) inside the TPM server. As a Java card can be of any format we could reuse the corresponding PC/SC user level framework and the corresponding low-level USB, SDIO and other bus drivers of the rich OS. This is possible due to the end-to-end BUS encryption between the corresponding TPM application and the TPM itself. Thus, we do not need to trust the involved soft and hardware layers during the communication as the OSAP and OIAP sessions provide authenticated and encrypted communication. The corresponding endpoints are in the TPM server inside the trusted environment and the TPM applet of the seTPM hosting smartcard. An example using the PC/SC subsystem of the rich OS is shown in Figure 5.10.

One pre-condition for the secure communication is obviously that during `Take_Ownership` when the credentials for the secure communications are established on both sides, the rich OS must not be compromised. Thus, we have to do the `Take_Ownership` in a controlled initial setup phase without any remote network connection.

Another mandatory pre-condition is that any generated `authdata` is kept in the TPM server. It is a known issue [CR10] of the OSAP and OIAP protocols in scenarios in which `authdata` is shared, for instance for accessing the SRK. The session key S is derived by an HMAC with the `authdata` key and two publicly exchanged $Nonces$ n_e, n_o. Thus, an attacker who has access to `authdata` and who has observed the nonce exchange could easily derive the session key S by computing:

$$S = HMAC_{\texttt{authdata}}(n_e, n_o)$$

However, as our architecture above specifies the TPM server as the trusted instance which is the only process directly communicating with the TPM, we can assure that authdata is only available within the TPM server.

Obviously, with this approach we are not able to realize a secure boot as described above but we are still able to run an authenticated boot procedure and attest the system integrity after the system has fully booted. Further, this approach also has the drawback that it is vulnerable against availability of the system as the attacker inside the rich OS just can block access to the smartcard which hosts seTPM. However, availability is not part of our attacker model, as also blocking access to the network of which drivers and stacks are located in the rich OS would make remote attestation infeasible. To mitigate this attack we propose to stop executing trusted applications after a certain timeout on the TPM connection inside the TPM server. This is possible as we have full control over timers and execution inside the trusted runtime in contrast to several trusted execution environments using the ARM TZ which do not have a scheduler inside the secure world. Note that within this work, we do not provide any implementation details and evaluation about the seTPM-Applet itself. This is provided in the joint work [PWS15]. Furthermore, for our herein provided concepts, we strongly believe it is sufficient to examine the real TPM chip only. The seTPM is just a transparent hardware replacement for the real TPM chip to make our architecture and evaluation results transparently applicable to mobile devices which possess an SD card slot but provide no option to connect a real hardware TPM chip.

5.5 Summary

In this chapter, we derived our global system architecture which combines approaches from the two working groups TCG and GlobalPlatform. We showed how we integrated the appropriate components which we denoted as building blocks both in hard- and software into a microkernel-based system architecture. We gave two realization examples for the TEE, namely one with Fiasco.OC/L4Re and the other with PikeOS, a widely used real-time operating system framework in the avionics industry which provides partition separation according to ARNIC-653 [Aer97]. Further, we described how to establish trust during boot and we specified a reduced TSS and TPM command set which allows us to reduce the TCB. We also defined some user space components like the TPM server (*TS*) and the integrity server (*IS*) which we need for the following Chapter 6. We finally discussed the architecture of *seTPM* a

Java Card-based TPM replacement, and the integration into our microkernel architecture to enable trusted computing on mobile devices.

All the described components together form the basic architecture which pursue the goal to provide a flexible architecture for mobile and generic embedded systems. Our approach enables the direct application of trusted computing based use-cases as well as GlobalPlatform mobile device use-cases. Further, due to the combination, we were able to improve the security for TPM-based secure storage, by the means of providing the missing runtime protection for symmetric session keys with the TEE concept.

6

System Integrity for the Trusted Environment

In the last chapter, we generally discussed the overall system architecture. In this chapter, we elaborate the trusted computing related building blocks even further. We do this by proposing a detailed system architecture with a TPM-based integrity verification service for the just derived microkernel-based trusted execution environment that allows to securely load remote binaries. 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 party. In addition, our secure loading service is able to assure that an encrypted remote binary is executed in a trusted environment on a specific device without relying on processor-specific security features, such as ARM TrustZone [ARM09]. By implementing the integrity verification and secure loading service as native microkernel tasks, we can also separate it from the rest of the system. So, compared to IMA (see Chapter 2.1.4), it does not rely on the trustworthiness of a rich operating system. That way, our approach not only adopts the main ideas of IMA for Linux-based systems to a microkernel system, it also reduces the trusted computing base for the integrity measurement components. After introducing the theoretical concept, we demonstrate the feasibility of our approach by a practical implementation. By means of this implementation we discuss code size, security and performance in a subsequent evaluation.

As a reminder, parts of this chapter were published in [WWHW14].

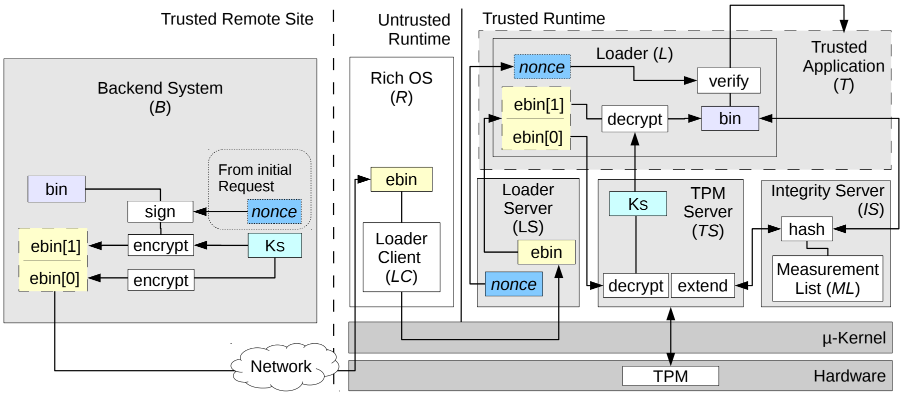


Fig. 6.1: Secure loading of a remote binary into our trusted runtime environment

6.1 Concept

In the following sections, we present the concept of our integrity verification and secure loading mechanism. In Section 6.1.1, we first specify the relevant components involved in verifying and loading remote binaries. After that, we discuss the necessary steps for the provisioning of our protocols in Section 6.1.2. We then describe the measurement in more detail in Section 6.1.3. Finally, we give a mapping for the remote attestation challenger protocol in Section 6.1.4, which provides local attestation abilities inside the microkernel runtime.

6.1.1 Loading External Trusted Applications

Our concept for loading external binaries from a secure backend system into the trusted runtime environment is depicted in Figure 6.1. It shows the entire loading architecture including all components until an external binary is loaded and measured in its final context inside the trusted runtime. Before we discuss the procedure in detail, we introduce the involved entities:

Backend System (B) The backend system B is the remote server, which wants to deploy an external binary to the embedded system over the network. B signs the binary *bin* with it's private key BK_{priv} and then encrypts it with the trusted runtime's public key TK_{pub} using a hybrid encryption. The binary is encrypted with a random session key K_S while that key is encrypted with TK_{pub} of the trusted runtime to which the corresponding private key TK_{priv} is only accessible inside the hardware TPM.

Trusted Application (T) The trusted application T is the final context where the measured binary *bin* is loaded and executed by loader L.

Loader Client (LC) The loader client LC is part of the untrusted runtime's rich operating system and acts as a relay between the trusted remote side and the trusted runtime. It just forwards the encrypted binary it requested from the trusted remote site to the loader server LS inside the trusted runtime over a well-defined interface.

Loader Server (LS) The loader server handles requests from LC and is responsible for starting new applications. It copies a runtime binary which contains the loader L into a new application context T and gives access to the encrypted binary to L. Also LS is responsible for generating the trusted runtime's *nonce* to enable freshness verification in L.

Loader (L) The loader L is the instance of a runtime binary, which is executed by LS in the context T. It decrypts the symmetric session key using the TPM server's unbind mechanism. With the retrieved session key K_S, L is able to decrypt the encrypted binary *ebin*, retrieving the actual binary *bin* into a new private memory region. After checking the signature involving the *nonce* from LS with the backend's public key BK_{pub}, it measures the *bin* by computing a hash and using the IS to store the measurement.

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 integrity server IS or the Loader L. If access to TS is granted to the untrusted runtime, the TPM server needs to restrict access to certain resources inside the hardware TPM. For example, the untrusted runtime must not use the trusted runtime's decryption key (cf. Section 6.1.2).

Integrity Server (IS) The integrity server IS manages a measurement list ML, which contains hash values of each started binary. It handles requests of L, in which the measurement results of the loading procedure are included. IS further uses TS to protect the measurement list inside a PCR of the TPM.

It is immutable for our concept that proper access control to certain objects and IPC between the above described entities are instantiated. Our prototype implementation uses the capability system from the microkernel for this purpose. This is discussed in Section 6.2.4.

Loading Procedure Our loader concept is based on a special startup procedure, which is for instance provided by L4Re. Instead of directly loading the binary, the *init* process loads an instance of a runtime binary into a new context. That runtime binary then loads and executes the actual binary in its own context. Therefore, we are able to decrypt the binary directly in its target context using the runtime binary. This assures that the binary can only be executed in the trusted runtime. Also strong isolation in terms of confidentiality to other externally loaded trusted applications is assured at any time, due to the binary's plaintext is only available in its own context. The unprotected version never passes any shared resource, where it may leak information to others using those shared resources at the same time. If for instance we did not apply the encryption with a key only usable in the trusted environment, an attacker could execute the binary anywhere skipping the signature check and also the measurement in a non-secure context. However, due to the encryption with the trusted runtime's public key bound only to one specific TPM, the binary can only run in one specific devices' trusted environment. We discuss this assumption later in our security analyses in Section 6.3.2. This further allows us to utilize a modified version of the challenger protocol from IMA [SZJvD04] for local attestation (see Section 6.1.4) as first routine in the external binary. If the local attestation fails, the binary can instantly decide to stop execution on its own, without any further backend communication. This mitigates the TOCTOU problem which we introduced in Chapter 4, as the time window between measuring the binary and its actual attestation approval is rather small since all operations are performed directly in the trusted runtime. To protect the trusted runtime from being compromised by random binaries generated by an attacker in the untrusted rich OS we need to apply a signature before the encryption inside the backend system. Furthermore, to prevent replay attacks the trusted runtime generates a nonce, which is sent in combination with the initial request to the backend. A formal representation about the just described loading procedure is depicted in Figure 6.2.

6.1.2 Provisioning

The necessary cryptographic keys and certificates of our protocols need to be generated and exchanged between the backend system and the device. Therefore, we need a provisioning (or activation) phase before the device can be used.

As it is not advised to use the Endorsement Key (EK) directly, we let the TPM generate a non-migratable storage key pair TK for encryption/decryption and store the resulting public key TK_{pub} in the backend system. If the backend later encrypts data with this key, it is assured that this data can only be

1. R : requests *nonce* from LS
2. $R \rightarrow B$: *nonce*, request for *ebin*
3. B : generates symmetric session key K_S
4. B : $Sig = sign(\{bin, nonce\}, BK_{priv})$
5. B : $ebin[0] = encrypt(\{K_S\}, TK_{pub})$ // asymmetric
6. B : $ebin[1] = enc(\{bin, Sig\}, K_S)$ // symmetric
7. $B \rightarrow LC$: *ebin*
8. $LC \rightarrow LS$: *ebin*
9. LS start L
10. $L \rightarrow LS$: requests encrypted binary *ebin*
11. $LS \rightarrow L$: *ebin*
12. $L \leftrightarrow TS$: $K_S = decrypt(\{ebin[0]\}, TK_{priv})$
13. L : $\{bin, Sig\} = dec(\{ebin[1]\}, K_S)$
14. L : $verify(\{bin, nonce\}, Sig, BK_{pub})$
15. $L \leftrightarrow IS$: runs measurement protocol (see below)
16. L : starts *bin*

Fig. 6.2: Secure remote loading procedure

decrypted on the device with that specific TPM. We use the TPM command `CreateWrapKey` for this purpose, it not only encrypts the key pair with the TPM's Storage Root Key (SRK), but also binds its usage to a certain platform state. Remember, we cannot use an AIK here since AIKs merely are signature keys; see Chapter 2.1.2. The embedded platform needs to create the key pair in the following manner. The device is booted into a safe system state after initial setup of the trusted runtime. At this point, only boot time integrity protected applications are running. The `CreateWrapKey` command is issued with current platform registers' state. As a consequence, the resulting wrapped key is bound to a platform state before any untrusted component is started. After roll out, the init process always needs to start the TPM server in advance of the untrusted runtime components, as *TS* loads the wrapped key for usage into the TPM. Later on, it is not possible to access the wrapped key anymore as the first user application which is not part of the measurement setup, for instance the untrusted runtime's rich OS kernel, changes the platform state. *TS* also denies access to the key slot which is used for TK_{priv} through its external API as TK_{priv} is only used internally in the TPM server itself. Remember *TS* is the only software component which communicates with the TPM directly after the system is fully booted. Assuming proper resource access rights to the device resources, it is not feasible by untrusted applications to access the TPM by-passing the *TS* or the trusted runtime. This prevents an untrusted user application to use *TK*. The same procedure is done for an attestation key

1. $L : h_i := h(bin_i) = SHA1(bin_i)$
2. $L \rightarrow IS : h_i$
3. $IS : append(ML, h_i)$
4. $IS \leftrightarrow TS : extend(h_i, PCR\#)$

Fig. 6.3: Measurement protocol

pair *AIK* which we need for attestation. Since we only have one attestation key pair in our concept, we denote this as *AIK* in the following. However, instead of `CreateWrapKey` the `MakeIdentity` command is used to generate the encrypted key pair. On the other side, to provide the ability to verify code signatures on the embedded system's trusted runtime, we need a PKI infrastructure, maintaining also certificate revocation and recertification in case of a certificate is going to expire. The initial root certificate CA_{pub} for the backend's code signing key BK_{pub} is delivered in the read-only storage of the embedded device during provisioning.

6.1.3 Measuring Integrity

In this section, the measurement of applications is contemplated. The measurement takes place before the application is being started. Measured are all applications running locally on the system, except for the measurement components itself, as stated above these components have to be measured and integrity checked during boot by the secure boot.

Figure 6.3 shows the measurement protocol for a binary bin_i, where i specifies a certain binary. For $i = 1$, measurement is calculated and stored inside the TPM as follows:

1. Before the binary bin_1 is started, the loader L calculates a hash value $h_1 := h(bin_1) = SHA1(bin_1)$ over the binary file of bin_1.
2. h_1 is sent to the integrity server.
3. The integrity server appends this measurement entry h_1 to its measurement list (ML) which holds all measurements performed before.
4. Then the integrity server uses the TPM server to aggregate h_1 into a PCR using the `TPM_extend` command. The aggregate is calculated securely inside the TPM as follows: $PCR \leftarrow SHA1(PCR \,|\, h_1)$, where $|$ denotes a concatenation.

The protocol is also run for any new binary bin_2 to bin_n so that the PCR contains an aggregate of all measured binaries $bin_1 \ldots bin_n$ which were executed

1. C : generates $nonce$
2. $C \rightarrow IS$: $nonce$
3. $IS \rightarrow TS$: $nonce$
4. TS : loads AIK_{priv} into TPM
5. TPM : $Sig_{TPM} = sign(\{PCR, nonce\}, AIK_{priv})$
   TS : $Quote = \{PCR, Sig_{TPM}\}$
6. $TS \rightarrow IS$: $Quote$
7. $IS \rightarrow C$: $Quote, ML$
8. C : $verify(\{AIK_{pub}\}, Sig_{AIK}, CA_{pub})$
9. C : $verify(\{Quote\}, Sig_{TPM}, AIK_{pub})$
10. C : calculates $A = aggregate(aggregate(ML), nonce)$
11. C : checks if $A == PCR$
12. C : checks if $A \in$ whitelist WL

Fig. 6.4: Challenger protocol

inside the trusted runtime. Because measurement takes place before the application is started, the application has no influence on its measurement process, but it could try to alter a PCR value after it has been started. However, this is prevented by the TPM chip, which does not allow to set PCRs to arbitrary values.

6.1.4 Attesting Integrity

The measurements of the loaded applications alone do not provide a trustworthy system. For this, there must be an instance that evaluates if the aggregate of all measured applications represents a trusted state of the system. This is accomplished by a trusted challenger instance. Usually, this is done remotely by the trusted backend. However, due to our secure loading concept, where we can assure that the binary could only be executed in the trusted environment, the trusted backend can embed the challenger into the application itself. The application can verify the system state locally directly after execution. For this purpose, a current white list is embedded in the application binary itself. Figure 6.4, shows the challenger protocol adapted from IMA [SZJvD04] to our microkernel based trusted runtime for local and remote attestation. The detailed protocol sequence depicts as follows:

1. A *nonce* value is generated. It prevents replaying old trustworthy system states instead of the current one.
2. The challenger C sends a request for the current measurement list to the integrity server IS. Further, C also passes the nonce value to IS.

3. *IS* forwards the nonce to the TPM server TS.
4. The TPM server first needs to load the private attestation key AIK_{priv} into the TPM.
5. After that the `TPM_Quote` command is invoked for the PCRs involved during boot and the PCR used for protecting the measurement list *ML*. The TPM returns the PCR contents to *TS*. Furthermore, it extends the indicated PCR with the nonce value in a temporary register and signs it with AIK_{priv}. The signature is also passed back to *TS*.
6. The PCR values as well as the signature returned as *Quote* by the TPM are passed back to *IS*.
7. The integrity server transfers *ML* to the challenger, as well as the *Quote*, which contains the PCR contents and the signature of the nonce-extended PCR contents.
8. *C* checks if the TPM's AIK_{pub} is valid. For this purpose, it traverses the certificate chain until it finds a valid signature. In Figure 6.4, this is indicated by Sig_{AIK}, which is verified with the root CA's public key CA_{pub}.
9. If the preceding step succeeds, *C* verifies with AIK_{pub} whether the signature fits to the signed data, i.e., the nonce extended PCR contents.
10. In the case of success, the challenger calculates the aggregate A of the *ML*. For this purpose, it iterates over the hash values contained in *ML* and calculates $A \leftarrow SHA1(A \mid h_i)$ for each binary hash h_i in the list. Since hashing is not commutative, the order has to be exactly the same as during aggregation of the measurements during the measurement protocol. At the end, the *nonce* value has to be added, i.e., $A \leftarrow SHA1(A \mid nonce)$.
11. The aggregate A should now equal the value as the PCR contents returned by the integrity server including the nonce.
12. If this condition is fulfilled, and the aggregate is in the whitelist *WL*, then the system is regarded as trustworthy by the challenger. Otherwise the challenger will not trust the system and terminates execution in the local case.

With both the measurement and the challenger protocol we now can accomplish the security objective of platform integrity.

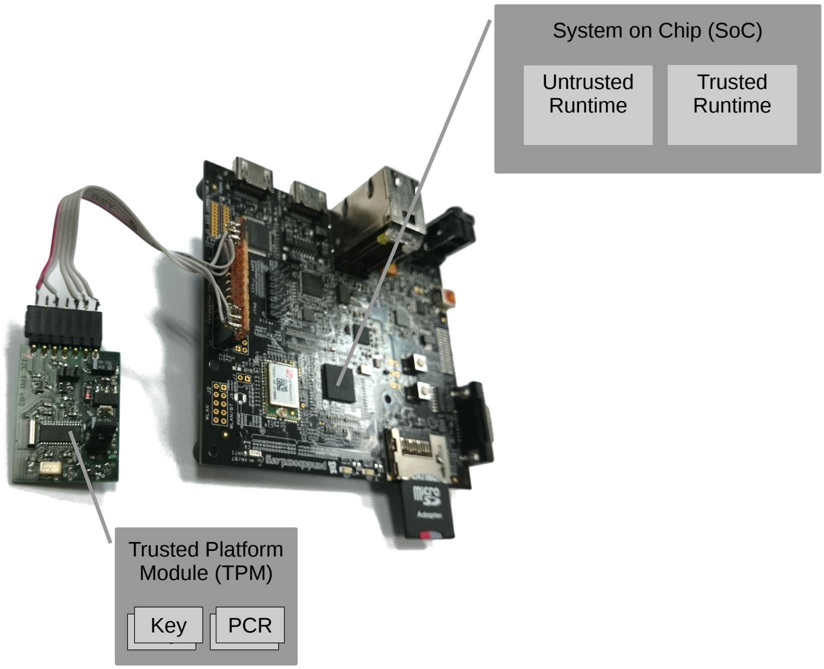


Fig. 6.5: Prototype platform OMAP 4 utilizing dual-core ARM Cortex-A9

6.2 Implementation in L4Re

For a concrete implementation of the proposed loading and integrity protocols we use the Fiasco.OC/L4Re framework as pointed out in Chapter 5. As hardware platform, we used the BeagleBoard[1] as well as the PandaBoard[1]. The BeagleBoard shown in Figure 6.6 is based on an ARM Cortex-A8 single core SoC (TI OMAP3) and the PandaBoard (Figure 6.5 is based on an ARM Cortex-A9-based dual-core SoC (TI OMAP4). Both represent modern hardware of widely used, low-cost and mid-range smartphones. We connected a TPM chip according the TPM v1.2 specification [Tru11] over the Inter-Integrated Circuit (I2C) bus to our development boards. Note that the Backend is out of scope of the implementation as we want to show the feasibility of the implementation on the mobile/embedded device. That the Backend side could be implemented is beyond question.

[1] http://beagleboard.org, http://pandaboard.org

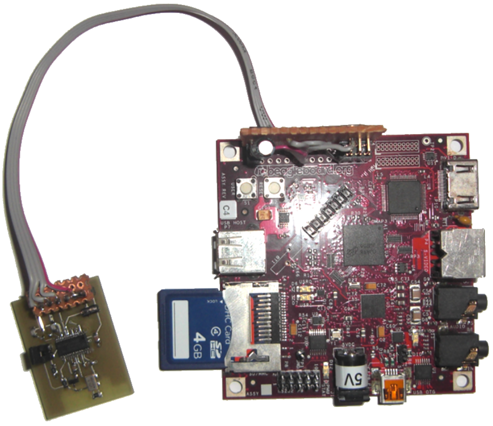


Fig. 6.6: Prototype platform OMAP 3 utilizing ARM Cortex-A8

6.2.1 IPC Abstraction in L4Re

For IPC between applications inside L4Re, a C++ abstraction framework based on IOStreams is available. We utilize this framework for the communication between our components. To establish connections between applications, the init process Ned is used. Ned connects applications usually over a new communication channel inside its Lua configuration script. Further, Ned builds the applications' initial set of capabilities which include access to those channels. On server side as well as client side, the reference to a capability for a channel can then be requested from the application's environment. The references are only local to the initial startup. Therefore, we have to provide other means to establish the communication between the external binary which is started during runtime and the initially started services. We use a global namespace for that and provide the capability to access this namespace over Ned. The server needs to register its server object to that namespace instead of the communication channel created by Ned's Lua script. Despite that, the client server communication over the IOStreams works the same as if the connections would have been accomplished with the Lua mechanism.

6.2.2 TPM Library

We implemented a simple C library, which contains all necessary TPM commands for our concept as specified in Chapter 5.3.1. This library is designed to run standalone. Due to performance reasons, we made one exception, we use an external crypto library for host-side SHA1 computations instead of our TPM based SHA1 implementation since the used TPM chip only provides a rather slow software I2C implementation. In case of the boot loader, this is PolarSSL and for L4Re we use the provided libcrypto package. From a high level point of view, we need and implemented the following commands: `TPM_Unseal`, `TPM_MakeIdentity`, `TPM_LoadKey2`, `TPM_Quote`, `TPM_Extend`. Those commands depend on several other commands, e.g., for initializing the TPM and handling OSAP sessions which provide low-level bus encryption. We implemented those commands according to the TPM command specification. Whenever possible, command parameters are hard-coded for our specific application scenario. This keeps the function header as simple as possible, and makes our 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 to generate an AIK key pair are directly specified in the function itself. This reduces errors in higher abstraction layers since wrong key type, encryption and signature schemes could not be set accidentally.

6.2.3 TPM Integration

To protect the boot chain we integrated our TPM library in U-Boot, which is the bootloader of both of our prototype boards. Further, we implemented the necessary driver to access the TPM inside the bootloader. For the low-level connection to the TPM, we ported parts of the Linux 3.0 driver for OMAP boards and also the tpm_tis driver, which is needed for the Infineon TPM chip, to U-Boot.

In L4Re, we completely reuse our small TPM library for U-Boot. Also the low-level I2C device driver components are completely reused. However, we are not able to directly access the memory-mapped I2C device with the physical addresses used in the boot loader code. L4Re provides the IO Server which can be used to forward a physical memory region to several user space device servers. We configured the I2C memory region of the bus where the TPM is connected to be handled by the IO Server, e.g., for the PandaBoard:

```
i2c4 => new Device() {
  .hid = "I2C";
  new-res Mmio(0x48350000 .. 0x48350fff);
}
```

The IO Server can also be configured for handling and redistribution of IRQs. However, the TPM does not provide interrupt support for asynchronous noti-

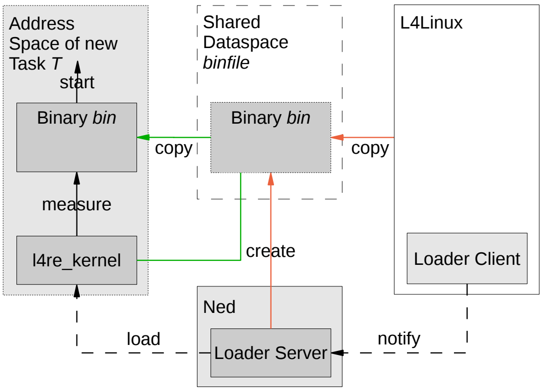


Fig. 6.7: Implementation of secure loading of an external binary into trusted runtime environment

fications. Instead, it is required to check the status register for completion by a polling mechanism. Thus, it is sufficient to map the above shown memory region into user space without taking care of an IRQ mapping.

Our implementation comprises a TPM server which on the one side is a client to the IO server handling low level I2C communication and on the other side provides high level TPM functionality as server for other applications. In our case, these are the integrity server and the loader inside the runtime binary *l4re_kernel*. The TPM server interface is rather simple and only maps to the high-level commands `Extend`, `Quote`, `CreateWrapKey`, `MakeIdentity` and `LoadKey2`. For those commands Opcodes are defined in a header file which the corresponding clients include 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. However, for key generation commands like `CreateWrapKey` or `MakeIdentity` we cannot transfer the payload through IPC due to size limitations. Instead, a shared memory page is used. The encrypted key structure is written on a previously created shared memory page in the TPM server's context.

6.2.4 Loading Binaries

In our implementation, we extend the init process Ned by our loader server *LS* to allow a dynamic loading of remote binaries. The original version of Ned only provides the ability to do this with a Lua script for binaries, which need to be known at startup and stored in the ROM file system that must also be part of L4Re. *LS* is realized as a thread inside of Ned. After Ned has done all its initial setup and generated server loops in several threads for the initial environment, we register another server loop at the end of the main thread for *LS*. The corresponding loader client *LC* in our implementation is realized as stand alone L4Re application which loads a binary from the global ROM file system. It stores that file in the temporary `binfile` dataspace which is shared between *LS* and *LC*. Later on, *LS* should be part of the L4Linux server and can be realized as a Linux kernel module which copies the actual binary *bin* from L4Linux user space to Ned in the way depicted in Figure 6.7. Access to the shared dataspace between *LC* and Ned is given by Ned to *LC* through a global namespace. This is discussed in detail in Section 6.2.5. Further, an IPC channel using the previously mentioned C++ IPC abstraction is set up using the standard Lua mechanism during Ned's initial startup. This channel is used from *LC* to notify *LS* as soon as a binary is completely copied from ROM to the shared dataspace. Then *LS* starts the runtime binary l4re_kernel in a new address space for Task *T* by executing a second Lua file. This internal default Lua file is read to configure the command line arguments and capabilities of the binary, especially access to the global namespace including the shared dataspace `binfile`, as depicted in Figure 6.8. Before the Lua interpreter for the second Lua file is started, the capability of the dataspace is registered to the global namespace where the new Task *T* is given access to. The l4re_kernel includes the loader which then reads the binary from the dataspace and starts the binary in its context. One important part here concerning security is that the binary which is measured may not be loaded out of a shared dataspace where the untrusted loader client *LC* has write access to, we discuss this in detail in Section 6.3.2.

6.2.5 Capability Transfer

To transfer the capabilities needed for IPC communication and access to shared memory, we use global namespaces as depicted in Figure 6.8. We create these namespaces already in MOE the root task, where they can be registered to the global environment which then is easily accessible in Ned from inside Lua code and native C++ code. As the loader client *LC* is part of the untrusted runtime, access to resources, which are part of the integrity and attestation framework is forbidden by our implementation. We introduce the `ldrs` namespace for communication between *LS* and *LC* and the `ims` namespace for communication between the entities inside the trusted runtime. Access to the namespaces is configured by Ned and access to the `ldrs` namespace is given on read-only basis to *LC*. This means that *LC* cannot create objects inside of the `ldrs` namespace, preventing the untrusted runtime to create other than the specified communication channels to the trusted runtime. Objects inside of the namespace,

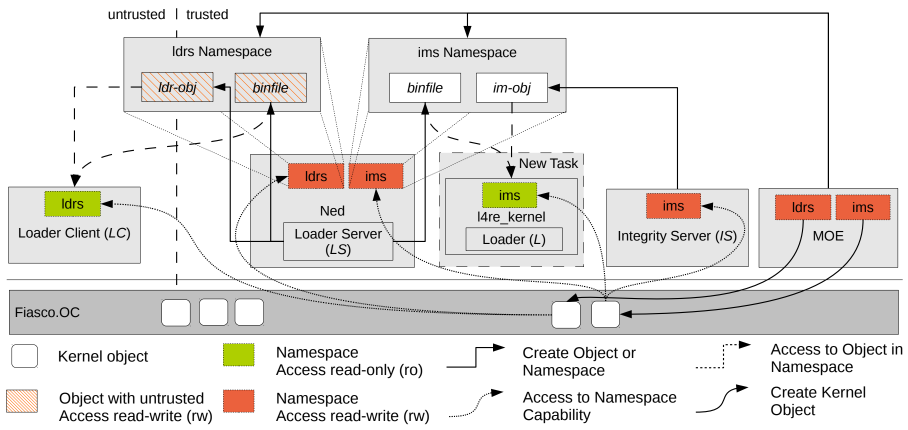


Fig. 6.8: Access to global namespaces and capabilities to server objects for IPC and data exchange

however, can be registered writable by Ned. *LS* registers the `binfile` object with write permissions inside the `ldrs` namespace. Further, for IPC communication the `ldr-obj` is registered by *LS*. On the other side, Ned configures access to the `ims` namespace with write access to the integrity server *IS* as *IS* needs to register the `im-obj` to which the *L* inside the l4re_kernel runtime binary can send IPC calls to conduct the measurement and challenger protocols. Also Ned registers the `binfile` object read-only to that namespace, thus *L* can access it for decryption and loading. The communication between *TS* and *IS* is not done over namespace but standard local IPC channels configured by the first Lua script during initial startup of Ned. In total, Ned can be seen as firewall between untrusted and trusted runtime.

6.2.6 Integrity and Attestation

We implemented *IS* as L4Re server. *IS* provides a public interface for the measurement and the challenger protocol. At startup the *IS* registers the `im-obj` in the global *ims* namespace to export two functions, one for the measurement and one for the challenger protocol, over the C++ IOStream abstraction for IPC.

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 challenger and measurement protocols.

Measurement Protocol We now describe some details about the implementation of the measurement protocol (Figure 6.3). 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 *bin* from the shared dataspace into a new local dataspace and computes a SHA1 hash of the received binary. Then, L calls the integrity measurement routine over IPC and sends the SHA1 hash and filename of the just measured binary over the IOStream to *IS*. The call returns, when the protocol is finished with a success or an error code. *IS* uses the `Tpm` class method `extend()` to send the SHA1 hash of the binary over *TS* to the TPM. Further, *IS* creates a new list item which contains the filename and hash value, which then is appended to the measurement list.

Challenger Protocol The implementation of the challenger protocol (Figure 6.4) is more complex than the measurement protocol. However, from a client perspective (we implemented a challenger application C for this) the main protocol is hidden and done in only one initial call over IPC including the *nonce* for freshness generated by the challenger. However, the results of the attestation are delivered in a single data structure containing the *AIK*'s public key as well as the raw signature returned by `TPM_Quote` and the measurement list *ML*. The TPM server returns the AIK_{pub} in the `TPM_Key12` format. The `Tpm` class abstraction transparently does the conversion of the `TPM_Key12` to the raw 2048 bit RSA key for signature verification. In our current implementation, we also generate a new AIK key pair with `TPM_MakeIdentity` when *IS* requests the AIK_{pub} on behalf of the challenger C, due to the lack of persistent storage in our trusted runtime. At the end of the protocol, when the IPC returns, C needs to do the verification steps 8 to 12 of the protocol (Figure 6.4). The necessary information are read and parsed from the shared memory. For this purpose we have implemented the `tpm_extend()` function in software which computes the aggregate over all hash values contained in *ML* and the *nonce*. To verify the signature from TPM_Quote which is an RSA encrypted hash over a `TPM_QUOTE_INFO` structure, C needs to recreate this structure with its locally computed values for the aggregate and *nonce* and also compute a fresh hash value over this structure. If the comparison of the fresh hash and the decrypted hash are equal and also the hash is contained in the whitelist *WL* the system is trustworthy and the program execution can continue.

6.3 Evaluation

In the following, we evaluate our concept and prototype regarding code size, security and performance.

6.3.1 Reduced TCB

In this section, we show that the TCB of our approach is much smaller compared to approaches, which rely on a rich OS kernel like Linux for measuring integrity values.

The trusted runtime allows individual composition of the TCB for each user level task. As for instance the TPM server uses the IO server for mapping the right memory to the TPM chip, the IO Server is also part of the TCB for the TPM server. Further, MOE as the root task and Sigma0 as root pager are also part of the TCB for every other task. This results in the following per-application TCB:

Sigma0:	Fiasco.OC, Sigma0
MOE:	Fiasco.OC, Sigma0, MOE
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 for our external binary loading implementation, which is protected by the boot time integrity protection mechanism in U-Boot. To estimate the size of the TCB we counted the lines of source code for all these components in the L4Re source tree as well as the libraries on which they depend. We used the tool cloc[2] for this. 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.

For the rich OS, in our case L4Linux, we count ≈9.5 MLOC. This also includes the IMA code. An Android version 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 kLOC for Fiasco.OC. This also includes the in kernel debugger jdb with about 20 kLOC. For the

[2] CLOC Count Lines of Code: http://cloc.sourceforge.net/

Table 6.1: Code sizes

TPM Server

	TrouSerS	**tpm library**	**tpm driver**	**i2c driver**	**total**
LOC	74687	1908	398	751	3308

Additions to L4Re

entity	***IS***	***TS***	***LS***	***L***	***MOE***	**total**
LOC	703	251	215	206	6	1381

Overall TCB

ARCH	**L4Linux**	**L4Re**	**Fiasco.OC**	**total**
total (kLOC)	≈11,000	≈751	≈120	≈871
arm (kLOC)	≈9,500	≈751	≈95	≈851

trusted runtime including all libraries needed for our integrity measurement and loader service we measured ≈750 kLOC.

Code size of our Addition to L4Re In the following, we estimate the additional size of code we introduce with our integrity setup to L4Re. The results are summarized in Table 6.1.

Our integrity server is very small with 703 lines of code. This includes the challenger and measurement protocol implementation. The TPM server has 3308 lines of code. However, as the TPM server shares the TPM library and the I2C driver with the U-Boot code, we subtract this from the size of the runtime environment, as this is already counted to the TCB by the bootloader. Thus, the addition to the TCB at this stage is only 251 LOC by the TPM server. Further, compared to a full TSS implementation like TrouSerS with about 74 kLOC in version 0.3.13, our tiny TPM library indeed significantly reduces the TCB additionally of approximately 72 kLOC. The exact values are compared in Table 6.1.

The loader server *LS* adds 215 LOC to the original Ned of L4Re. Additionally, to integrate the measurement code for *L* into l4re_kernel, we added 206 LOC to the original loader.cc. To register the *ims* and *ldrs* namespaces in MOE's root namespace, we only had to add 3 lines for each namespace. In total we added 1381 LOC to L4Re.

Total TCB Code Size If we now compare our solution, which is in total, Fiasco.OC plus L4Re including our additions, to other approaches which rely on a Linux kernel, this is 851 kLOC versus 9.5 MLOC. We are one order of magnitude smaller in code size. For this comparison, we do not need to take the boot loader into account, as this needs to be count 1:1 to both approaches. 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.

6.3.2 Attack Prevention

In this section, we present common attack goals and discuss how our concept can prevent those attacks.

Run Arbitrary Code undetected in Trusted Runtime To prevent an attacker from running arbitrary code in the trusted runtime, we have to assure that our measurement code is called before the binary gets executed and that the correct binary is measured. Then, when a challenger later runs the attestation challenger protocol, it will recognize the untrusted system state. As the l4re_kernel is loaded before the actual binary and the measurement code is part of the l4re_kernel, it is assured that the binary is measured. To guarantee that the correct binary is measured, it is immutable to set proper access rights for shared dataspaces. Further, it is necessary to copy the binary to a private context inside the trusted runtime. From a complexity point of view in this scenario, one would give the loader client direct access to the dataspace shared between Ned and l4re_kernel for transferring the binary. However, the loader client needs write access to that dataspace to place the binary there. This write access is permanently given to LC. The problem arises that the signature check and measuring of the binary is not atomic. If the binary would be loaded directly out of that shared dataspace, the attacker can watch for the moment in time where the signature and measurement has finished successfully and then replace the application code with arbitrary code. However, if the binary is copied to another context not accessible by LC before it is measured and the signature verification takes place, the process is kind of atomic from outside of the trusted environment. This is accomplished by the decryption, where the plain text is retrieved to a private buffer in context of the trusted Task T.

Execute Code in Trusted Runtime In contrast to the previous attack, where the challenger of another application or remote challenger can later detect the untrusted system state, now the attack is defined successful if the binary is executed at all. We prevent this attack by signing the code with the backend's private key BK_{priv}. The signature is checked before execution and only authentic code from the backend system is allowed for execution. However, to verify the authenticity of the code we need the ability to verify the certificate of the backend system including all implications of a PKI, as described in Section 6.1.2. A limitation of the prototype implementation is the lack of those PKI services. However concerning, code size of the TCB on the embedded device, this would be of no real concern as the corresponding crypto library is already used for SHA1 computation and thus already part of the TCB.

Run Binary in Untrusted Environment To prevent an attacker from executing a trusted remote binary in the compromised untrusted rich OS, the binary is encrypted with a random session key K_S. As TK_{priv} needed for decryption of K_S is only known inside the trusted environment and bound to the TPM, it is not possible to decrypt the binary without the TPM. Further, the wrapped key of TK is stored in a memory region of the trusted runtime. If the encrypted key would be accessible from inside L4Linux for instance through the ROM file system which is part of every L4Task started by Ned, the untrusted L4Linux could load the trusted environment's private key into the TPM if access to the TPM is granted to the Linux virtual machine. In our prototype, it suffice to restrict the TPM access to the trusted runtime only.

Compromise Integrity Services and Communication Communications between our user space components are firewalled by LS using separate namespaces. Hence, the only interface where an attacker can compromise the trusted runtime is the one exported by LS. This Interface is strictly controlled and protected by the capability system of the microkernel as described in Section 6.2.5. Further does the microkernel protect the boundaries of the shared dataspace, which is allocated and configured inside LS. Untrusted code execution, which may compromise components of our integrity and attestation framework from inside the trusted runtime later, is successfully prevented by code signing as described above. Attacks on the communication between backend and trusted runtime are prevented by the loading procedure, which assures confidentiality, authenticity and integrity of the message *ebin*.

Extracting the symmetric session key

From a protocol level, an attacker may only gain access to the symmetric Key K_S if he has either access to the backend system B or to TS. However, in our architecture the backend system is defined as trusted and attacks to gain access to the backend system are out of scope of this work. Further access to TS is prohibited through the microkernel address separation. However, as an attacker inside the rich OS can compromise LC, it is feasible to mount a time-driven cache attack. Explicitly the decryption $dec(K_S, \{ebin[1]\})$ (Step 13) of the loading procedure as shown in Figure 6.9 is the target. Steps introduced

```
 1. $R$ : requests $nonce$ from $LS$
 2. $R \rightarrow B$ : $nonce$, request for $ebin$
 3. $B$ : generates symmetric session key $K_S$
 4. $B$ : $Sig = sign(\{bin, nonce\}, BK_{priv})$
 5. $B$ : $ebin[0] = encrypt(\{K_S\}, TK_{pub})$      // asymmetric
 6. $B$ : $ebin[1] = enc(\{bin, Sig\}, K_S)$          // symmetric
 7. $B \rightarrow LC$ : $ebin$
    replay loop
    $LC$ start timing
    $LC$ randomly choose $ebin[1]$
 8. $LC \rightarrow LS$ : $ebin$
 9. $LS$ start $L$
10. $L \rightarrow LS$ : requests encrypted binary $ebin$
11. $LS \rightarrow L$ : $ebin$
12. $L \leftrightarrow TS$ : $K_S = decrypt(\{ebin[0]\}, TK_{priv})$
13. $L$ : $\{bin, Sig\} = dec(\{ebin[1]\}, K_S)$
14. $L$ : $verify(\{bin, nonce\}, Sig, BK_{pub})$
    verify fails
    $LC$ stop timing
15. $L \leftrightarrow IS$ : runs measurement protocol
16. $L$ : starts $bin$
```

Fig. 6.9: Timing attack on loading procedure

by a Man-in-the-Middle attacker inside the rich OS are highlighted in red. The attacker runs an replay loop by altering the $ebin[1]$ as random chosen plaintexts. The signature verification then obviously will fail, due to signature mismatch and an old nonce. However, the timing of the symmetric crypto algorithm which in our case is AES leaks information about K_S. This is a major threat to our system architecture. We deal with those kind of attacks separately in Chapter 7.

Extracting the asymmetric keys

Even though asymmetric cryptography is also vulnerable against cache-based side channel attacks, in our concept asymmetric encryption is performed only inside the TPM which does not have a cache and thus provides timing constant results. For the asymmetric keys used inside the backend, we make the same assumption as for K_S: the backend is defined trusted and out of scope of this work.

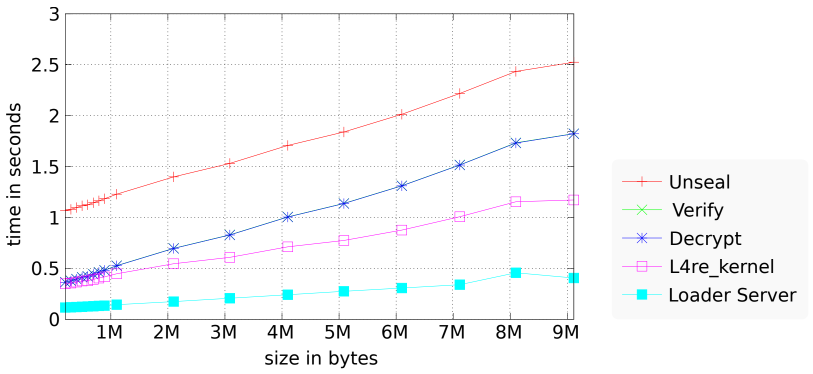


Fig. 6.10: Loader performance results for binaries with different size

6.3.3 Performance

We measured the performance of our measurement protocol on the OMAP4 based PandaBoard for several binary sizes starting at 200 kByte up to 9 MByte. The results are depicted in Figure 6.10. We measured 13.7 MByte/s throughput for *AES_256* and 595.1 verify operations per second for *RSA_2048*. The resulting accumulated timing is shown in the plot as blue line (Decrypt) for AES and green line (Verify) for RSA. The signature verification is independent of the binary size and could be neglected as we only have one signature to verify during binary loading. We identified one major time consumer in our setup, which is the Unseal command used for decryption of the session key K_S. We measured about 701 ms (Unseal) for this. The measurement generation including SHA1 computation and saving the hash value as well as extending it to the TPM is represented by the purple line (l4re_kernel). This is less then 300 ms for binaries up to 1 MByte. The plot also shows that the measurement protocol duration time increases linear with the binary size, which is caused by the SHA1 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 cyan line (Loader Server). In total, we see that the load time increases linear with the binary size and that the TPM is more or less the bottleneck. This is caused on the one hand due to the unoptimized implementation of the TPM driver, as we set polling timings rather conservatively. On the other hand, TPM chips are designed to optimize costs and not performance.

6.4 Summary

In this chapter we discussed integrity and secure loading of dynamic binaries in static microkernel contexts. Embedded systems based on microkernels are often statically configured to ensure safety and security. Consequently, the dynamic loading of remote binaries is usually not possible or allowed, since remote binaries might potentially compromise the trustworthiness, which is often not acceptable for safety- and security-critical applications. That is why we proposed an integrity verification and secure loading mechanism, which measures the integrity of a remote binary and securely stores the measurement inside the TPM before loading the binary. As a result, we adopted the main ideas of IMA to a microkernel. Furthermore, our approach separates the integrity measurement components from the rest of the system while reducing its TCB at the same time. Our prototype implementation additionally does not rely on processor-specific security extensions, such as ARM TrustZone, which many low-cost embedded platforms do not support. In the evaluation of our concept and implementation, we also showed that common attacks, which compromise systems by executing malicious remote binaries, can be detected and prevented. One exception remains for our attacker model. The symmetric session key is vulnerable to cross-VM cache-based timing attacks. We deal with the described attack in far more detail in Chapter 7.

7

Security of the Separation Layer

In this chapter, we get more specific about the isolation characteristics of our virtualization architecture. Although the proposed architecture in Chapter 5 provides strong isolation due to the isolation abilities of the microkernel to run para-virtualized untrusted rich operating systems, we show that this is not sufficient against an attacker as described in Chapter 4. Such an attacker is able to mount a cache-based side channel attack to circumvent the isolation mechanism.

At first we state related work on cache-based side channels in Section 7.1 to motivate our own attack vector and countermeasures. In Section 7.2, we then derive a time-driven cache attack from Bernstein [Ber05b] and show the vulnerability of TEEs, such as our own microkernel-based architecture or the GlobalPlatform TEE implementations using ARM TZ. Practical results on ARM-based embedded platforms are provided in Section 7.3.

We also performed Bernstein's original attack locally, measuring the timing not in the attacked system as Bernstein but in the attacking client application. We provide our results on this in Section 7.4 before we proceed dealing with countermeasures in Section 7.5.

Parts of this chapter were published in [WHS12, WWAS15].

7.1 Related Work on Cache-based Side Channels

In [Ber05b], Bernstein proposes a cache-based timing attack to recover the secret key of an AES encryption on a remote server. Bernstein's paper contained no thorough analysis of the attack and no explanation why the attack is successful. Neve et al. fill this gap in [NSW06, Nev06] by presenting a full analysis of Bernstein's attack methodology and explaining the correlation model. They argue that Bernstein's original technique cannot be used easily as a real remote-only attack where timings need to be measured by the attacker. Moreover, they improve Bernstein's attack by also considering second round information and thus lowering the number of required samples. To get accurate timings, Bernstein avoided the noisy network channel between the attacked server and the attacker by measuring the encryption time directly on the server, which is a rather unrealistic scenario since the server needs to be modified. In virtualization environments, however, the noise is negligible since local communication channels with only a small and almost constant timing overhead are used, as we will show in our evaluation in Section 7.3.

7.1.1 Attacking Virtualization

Ristenpart et al. [RTSS09] consider side-channel leakage in virtualization environments on the example of the Amazon EC2 cloud service. They show that there is cross virtual machine (VM) side-channel leakage. They used the access-driven *Prime+Probe* technique from [OST05] for analyzing the timing side-channel. However, Ristenpart et al. were not able to extract a secret encryption key from one VM. In the following sections, we consider a virtualization-based system in which the trusted environment runs an AES server. Under the assumption that the untrusted environment could be hijacked by an attacker, we showed that a man-in-the-middle attack via an adapted version of the cache-timing attack by Bernstein [Ber05b] is generally able to significantly reduce the key space, thus making brute-force attacks feasible. The impact of noise under realistic workloads is examined by Spreitzer and Plos [SP13], who evaluate time-driven attacks on conventional mobile devices (ARM Cortex-A8 and A9). Unlike our approach, they consider noise induced by the Android operating system and applications running simultaneously on the device. However, they do so using a slightly unrealistic attacker model where the attacker captures timings in the very same process where the AES encryption routine is implemented and called, which likely reduces the effects of the OS and concurrent processes. Recently Liu et al. [LYG$^+$15] showed a practical attack on several versions of GnuPG inside a cross-VM server scenario similar to the one of Ristenpart et al. [RTSS09] also using the *Prime+Probe* technique from [OST05].

7.1.2 Countermeasures

There are several ways to defend against time-driven cache timing attacks: One option is to switch to hardware-based implementations as provided by some processor manufacturers, e.g., Intel with its AES-NI instruction set [Gue12], thus entirely avoiding cache-based attacks against the algorithm. If no hardware support is available, it is possible to change the implementation of the algorithm itself and get rid of the table lookups or randomize them. While earlier software-based suggestions [MN07, Kön08] as well as our RMC-AES (see section 7.5) are generally slow[1] compared to table-based implementations, Käsper et al. [KS09] present an efficient constant-time implementation based on bit-slicing that is suitable for stream and packet encryption and is also available for ARM-based systems using the NEON SIMD instruction set extension.

Kim et al. [KPMR12] present a novel countermeasure against cache-based side channel attacks in a virtualization environment called STEALTHMEM. This countermeasure works at hypervisor level by assigning dedicated cache lines to each CPU in a group of CPUs with shared L3 cache. These so-called stealth cache lines are never evicted; therefore, sensitive data, such as S-boxes in AES, can be stored in these cache lines without introducing cache or timing side channels for an attack. Stefan et al. [SBY$^+$13] propose instruction-based scheduling to prevent cache-based timing attacks on a single CPU. Instead of having a fixed amount of time, a process has a fixed amount of instructions it can execute before the next process is scheduled. The authors examine a simple timing attack and show that this attack is prevented by the proposed scheduler with negligible increase in the size of binaries and execution time. These countermeasures require considerable changes to the hardware, the hypervisor, or the cryptographic algorithms, whereas neither of which is necessary for our *discrete-time* approach as we will described in Section 7.5. Varadarajan et al. [VRS14] have proposed a similar approach to our *discrete-time* scheduler scheme for cloud systems which they call soft-isolation. In contrast to our approach for real-time based schedulers, their approach relies on a feature of the Xen hypervisor scheduler called minimum run time (MRT) guarantee.

7.2 Attacking Trusted Execution Environment

For our attack setup, we focus on a virtualization-based system architecture of an embedded mobile device as stated in Chapter 5. In the following, we show that an attacker who has overtaken the rich OS in the untrusted environmentcan circumvent the isolation mechanism with a cache timing side-channel.

[1] Implementation [MN07] reaches its maximal performance only when processing 128 blocks in parallel.

Table 7.1: Mutual authentication protocol using symmetric AES encryption

Verifier B		**Prover A**
shared key: k		shared key: k
$r_B := rnd()$		$r_A := rnd()$
	$\xleftarrow{\text{connect()}}$	
	$\xrightarrow{ID_B, r_B}$	
		$m_A := h(r_B \| r_A \| ID_A)$
	$\xleftarrow{ID_A, r_A, c_A}$	$c_A = enc_{AES}(m_A, k)$
$m'_A := h(r_B \| r_A \| ID_A)$		
$c_A \stackrel{?}{=} enc_{AES}(m'_A, k)$		
$m_B := h(r_A \| ID_B)$		
$c_B := enc_{AES}(m_B, k)$	$\xrightarrow{c_B}$	
		$m'_B := h(r_A \| ID_B)$
		$c_B \stackrel{?}{=} enc_{AES}(m'_B, k)$

7.2.1 Authentication Scheme

To keep the trusted computing base (TCB) small and to reduce implementation complexity, the drivers and communication stacks of our system architecture as well as TEE architectures in general, are implemented in the rich operating system executed in the untrusted environment. Thus, to achieve for example authenticity of a transaction in an online banking application running in the trusted environment, a protocol resistant to man-in-the-middle attacks has to be used. The protocol's end points have to be in the trusted environment and not in the rich OS since the rich OS could be compromised. When the trusted application wants to communicate with its backend system, it has to prove its authenticity against the backend and vice versa. For this purpose, a mutual authentication protocol as shown in Table 7.1 between both parties needs to be employed. Note that this is only a simple example authentication scheme and also more sophisticated authentication schemes could be used. We assume that both parties have negotiated a secret symmetric key. The protocol uses random nonces as challenges and AES with the shared secret key k to generate the responses. Also an identifier of the particular sender is included in the encrypted response. Before the execution of the encryption, this ID is concatenated with the challenges. Further, this concatenation is hashed to prevent concatenation attacks.

Both verifier and prover execute the mutual authentication protocol depicted in Table 7.1. The prover in this case is the trusted application whereas the

Table 7.2: Timing attack on a trusted application

Untrusted VM		
To/From remote		**To/From trusted**
$\xleftarrow{\text{connect()}}$		$\xleftarrow{\text{connect()}}$
$\xrightarrow{ID_B, r_B}$	startClk()	$\xrightarrow{ID_B, r_B}$
$\xleftarrow{ID_A, r_A, c_A}$	stopClk()	$\xleftarrow{ID_A, r_A, c_A}$
	$m_A := h(r_B \Vert r_A \Vert ID_A)$	
	$\vdots$	

verifier is a remote backend system. The untrusted environment is not taking part in the protocol and just acts as transparent relay. After execution of this scheme, the prover A has proven to the verifier B the knowledge of the secret k and vice versa. Further, the freshness of the communication is provided by this scheme. This simple mutual authentication is used to demonstrate the vulnerability of virtualization-based trusted execution environments against the timing attack depicted in the next section.

Our introduced authentication scheme is secure against man-in-the-middle attacks on protocol level. However, due to the fact that the untrusted environment is relaying the messages between the client application and the remote server, our attacker can use a time-driven cache attack to at least partially recover the AES-encryption key k. To this end, we use a template attack derived from the attack in [Ber05b] which is conducted in two phases, first the profiling phase (offline and online) and second the correlation phase. We assume that an attacker has gained access to the rich operating system. The attacker is then able to execute a small attack process which is used to generate the timing profile.

7.2.2 Profiling Phase

The profiling phase is run twice, one time offline with a known key k and a second time online on the real target with an unknown key k'. However, the malicious program which is running on the attacked system only has to generate the online profile. The profiling phase in this context looks as follows. The attacker process has to hook into the messaging system between rich OS and the trusted execution environment as depicted in Table 7.2. Since the protocol stack is implemented in the rich OS, this could be done in the rich OS kernel. Thus, the attacker is able to capture the server's challenge r_B and measure the time between relaying this challenge to the client and receiving the client's response message. This provides him the timing of the AES encryption of the known plaintext $m_A = h(r_B||r_A||ID_A)$, of course with the noise introduced by the hashing and other operations executed in addition to the actual encryption.

To recover the key in the later correlation phase, a several millions of challenge-response observations are needed to deal with the noise by averaging over all samples as we will show with our evaluation in Section 7.3. Therefore, the attacker has to increase the number of challenge-response pairs to be collected. To achieve this, he has several options depending on the used implementation of the virtualization layer and the client application. In TEE implementations, like the GlobalPlatform TEE, an untrusted user application may be used to initiate the trusted application. Thus, a malicious program could initiate the trusted application as well and some kind of trigger application could be used to initiate the authentication process of the trusted application. The connection request to the remote server can be blocked by the attacker as he has full control over the untrusted rich operating system and thus can intercept any communication. Instead of relaying the connection request to the remote server, the attacker establishes a local fake connection and sends an own generated nonce to the trusted application. After receiving the answer with the ciphertext, the attacker can send a connection reset and depending on how the trusted application is implemented, the protocol will just restart and a new challenge can be sent. If we consider a mobile device with many sensors the attacker can hide its activity from the user by only performing the attack if those sensors signalize that the phone is not used interactively. For instance if the accelerometer indicate that the device is not moving and the display is off, it is much likely that the phone lies unused on a table. If the attacker additionally considers that the phone is connected to the charger, he also could hide the battery drain induced by the computational expensive crypto operations. Thus, the attack obviously takes even longer, but remains undetected and it is more likely to retrieve the necessary amount of samples.

7.2.3 Correlation Phase

After receiving sufficient challenge-response pairs for the online timing profile, the attacker can correlate the profiles to recover at least partially the key k'. We provide detailed measurement results in Section 7.3. We use a correlation based on timing information during the first round of AES. It would be possible to also use information from the second round to reduce the amount of samples needed. However, to show that time-driven cache attacks are a threat to virtualization-based systems, it is sufficient to use the easier first round attack.

At first we define the function *timing*() which computes the timing difference between the start and end of an operation. During the first run of the profiling phase, for each plaintext p, the overall encryption time is stored accumulated in a matrix $\mathbf{t}$ which is indexed by the byte number $0 \leq j < 16$ and the byte value $0 \leq b < 256$.

$$\mathbf{t}_{j,b} = \mathbf{t}_{j,b} + timing(enc_{AES}(p, k)) \tag{7.1}$$

Further, the total amount of captured samples for each plaintext byte value is traced in a matrix **tnum** as shown in Equation 7.2.

$$\mathbf{tnum}_{j,b} = \mathbf{tnum}_{j,b} + 1 \tag{7.2}$$

After several samples the matrix $\mathbf{v}$ which is computed as depicted in Equation 7.3 is stored in the profile.

$$\mathbf{v}_{j,b} = \frac{t_{j,b}}{\mathbf{tnum}_{j,b}} - t_{\text{avg}} \tag{7.3}$$

t_{avg} shown in Equation 7.4 is the accumulated timing measurements of all plaintexts p_m divided by the total number of encryptions l.

$$t_{\text{avg}} = \frac{\sum_{m=0}^{l} timing(enc_{AES}(p_m, k))}{l} \tag{7.4}$$

During the online part of the profiling phase, the matrices $\mathbf{t}'$ and $\mathbf{tnum}'$ are generated and the output $\mathbf{v}'$ is generated for the unknown key k'.

Finally, for every key byte j the correlation $\mathbf{c}$ for each possible value $0 \leq u < 256$ is computed as shown in Equation 7.5.

$$\mathbf{c}_{j,u} = \sum_{w=0}^{255} \mathbf{v}_{j,w} \cdot \mathbf{v}'_{j,(u \oplus w)} \tag{7.5}$$

The output list of possible keys is sorted according to the value of $\mathbf{c}$. Thus, key candidates with highest correlation $\mathbf{c}$ are added first to the output list. Further, the key values with the lowest correlation below a certain threshold as defined in [Ber05b] are sorted out. However, this threshold needs to be chosen carefully depending on the hardware.

Understanding Bernstein's correlation Bernstein himself did not provide a theoretic background why this attack works. However, Neve [Nev06] did an in-depth analysis of Bernstein's attack technique and showed that the following heuristic holds for $p_i \oplus k_i$, $p'_j \oplus k'_i$ as the results of the first *AddRoundKey* operation (see Equations [2.4, 2.7]) during offline profiling respectively online attack phase.

Heuristic: *the pairs satisfying the equality*

$$p_i \oplus k_j = p'_j \oplus k'_i$$

will have a matching timing-profile. [Nev06, p. 48]

As Bernstein computes the timing matrix for $t_{i,j}$ and $t'_{i,j}$ according to Equation 7.1 he averages out the individual timings of each possible value for a byte $j \in [0, 15]$. Thus, for each byte an own timing profile is generated out of random plaintext encryptions depending only on k or respectively k'. As a result, the heuristic is applicable and indeed gives the above described correlation between t and t' respectively v and v'.

Finally, the unknown key k' can be recovered by Equation 7.6

$$\begin{aligned} k'_j &= p_j \oplus k_j \oplus p'_j \\ \Rightarrow k'_j &= p_j \oplus p'_j, \text{ for } k_j = 0 \end{aligned} \tag{7.6}$$

7.3 Evaluation of TEE Attack – Empirical Results

For practical analyses of the above described attack, we built two testbeds based on embedded ARM SoCs with an L4 microkernel as virtualization layer. The first testbed represents low-cost mobile phone devices and utilizes a single-core processor while the second one represents current virtualized cyber physical systems with a quad-core processor.

7.3.1 Testbed1 – Mobile Phone

As hardware platform for the first testbed, we decided to use the BeagleBoard in revision c4 because it is widely spread community driven open source board and also comparable to the hardware of currently available low-cost smartphones, for instance the Apple former iPhone 3G as well as some Android smartphones. This is the same platform we used to initially implement our security architecture of Chapter 5 and protocols in Chapter 6. It is based on Texas Instruments' OMAP3530 SoC which includes a 32-bit Cortex-A8 core with 720MHz as central processing unit. The Cortex-A8 implements a cache hierarchy consisting of a 4-way set associative level 1 and an 8-way set associative level 2 cache. The L1-cache is split into instruction and data cache. The cache line size of both is 64 byte. For precise timing measurement, we used the ARM CCNT register, which provides the current clockcycles, the CPU

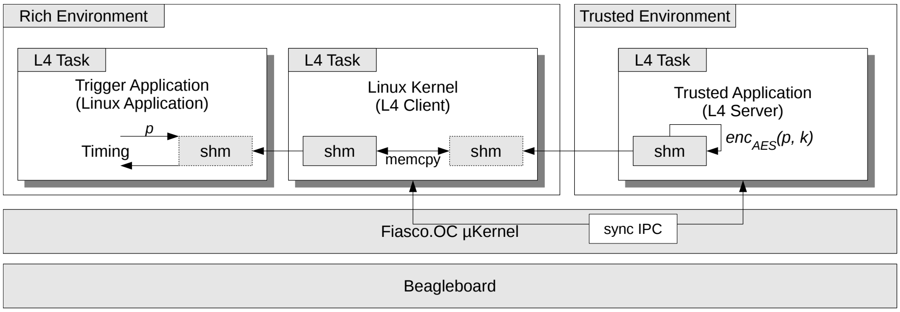


Fig. 7.1: Linux trigger application (simulating malware) connecting through L4Linux kernel services to trusted application executed as L4Server

spent since last reset. This is a standard feature of the Cortex-A8 and thus also available in current smartphones. However, it needs system privileges by default.

We implemented our TEE as described in Chapter 5 using the Fiasco.OC/L4Re realization shown in Figure 5.4 and employed the mutual authentication scheme from Table 7.1 in the trusted environment. We give a short recap on this: The Fiasco.OC microkernel in cooperation with the L4Re provides the functionality of a hypervisor for para-virtualized Linux machines. Further, it enables real time application and security applications to run directly on top of the microkernel in separated address spaces (L4Tasks) besides the Linux VMs. In fact, the L4Re virtualization runs Linux in user mode also in an L4Task. Further, each Linux application is executed in its own L4Task, however, with a special restriction that the L4Linux machine where the application belongs to is the registered pager of that task.

The rich OS is simulated by an L4Linux system. In L4Re an IPC mechanism in form of a C++ client server framework exists, see also Chapter 6.2.1. This provides a synchronous control channel. The trusted application is implemented as a L4Server while the client part is implemented in the L4Linux kernel. A user level application is implemented on top of the L4Linux kernel to trigger the authentication of the trusted application. Instead of real challenges of a remote server, we also used this trigger application to generate random nonces as server challenges. This approach makes no difference to the timing measurement. The actual plaintext data (the remote server's nonce r_B) is written to a shared memory page by the client. The client, in our case the L4Linux kernel, requests this shared page in advance from the trusted application. The trusted application L4Server registers the page in the microkernel and transfers the capability for the page through the established IPC control channel to the Linux kernel. A detailed view of the software architecture of this attack is provided in Figure 7.1. As the rich OS is running in user mode, it is necessary to enable the access to the CCNT register beforehand in system mode. We used the boot loader *U*-Boot to set this instruction before the microkernel is executed. However, if the TEE would be realized for example with ARM TrustZone [ARM09], the rich OS is executed in the so called NormalWorld. The SecureWorld of the processor is used for the trusted execution environment. An attacker could then access the CCNT register directly from the rich OS kernel since access rights of the NormalWorld's system mode are sufficient. Also in newer versions of Fiasco.OC and L4Re for the Cortex-A9-based PandaBoard which we used to demonstrate the secure loading approach in Chapter 6 is the CCNT register enabled for user mode access per default.

7.3.2 Testbed2 – Cyber Physical Systems

As second testbed we elaborated the following setup which represents an embedded system as part of currently deployed cyber physical systems, e.g., in aircrafts. The untrusted runtime is implemented using the para-virtualized Linux distribution ELinOS (version 5.2) including the necessary code for the attacker to conduct the timing attack. The AES server in the trusted runtime is implemented as an application based on the native PikeOS API (version 3.3). Obviously, both applications have their own partition. To enable the communication between the two partitions, two unidirectional queuing ports and a shared memory page were set up. The rich OS and the AES server use these ports to communicate via a simple handshake protocol and use the shared page as buffer for plain- and ciphertexts. Queuing ports are unidirectional communication channels defined in the ARNIC-653 [Aer97] standard that can be set up between two partitions statically at compile-time and then initialized at run-time by the applications. A detailed view of the software architecture of this attack is provided in Figure 7.2.

As hardware platform, we chose the Freescale i.MX6 SabreLite board (shown in Figure 7.3) which comprises a Quad-Core ARM Cortex-A9 CPU with 1.2GHz. We did not use the PandaBoard as in Chapter 6 as it only comprises a dual core CPU. However, they share the same micro architecture and thus the results can be mapped to that system to. The cache architecture consist of a 32KB I- and D-Cache (L1) per Core and a 1MB shared L2 cache. The L1 cache is 4-way associative and has a cache line size of 32 byte. For precise timing measurements, we again used the ARM CCNT register as in Testbed1.

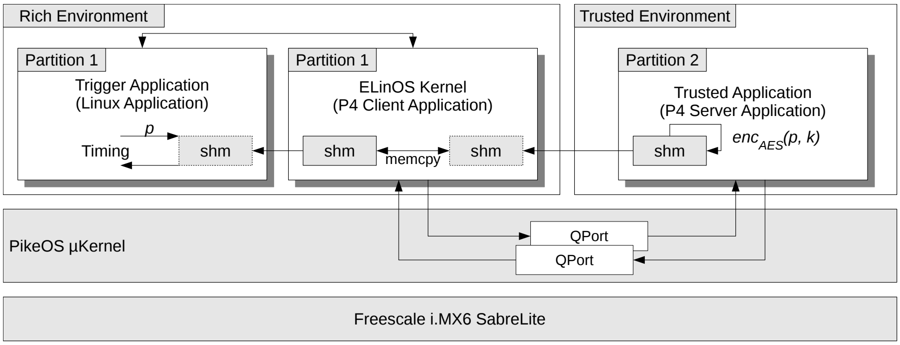


Fig. 7.2: Linux trigger application (simulating malware) connecting through ELinOS kernel services to trusted application executed as PikeOS native server application

7.3.3 Measurement Setup

The side-channel leakage depends on the used AES implementation. We will evaluate this by analyzing different AES implementations on Testbed1 using our authentication protocol shown in Table 7.1.

During the profiling phase, we used the null key for the offline part and for the online part we generated the randomly chosen key k':

$$k' = \texttt{0x 2153 fc73 d4f3 4a98 1733 bb3f 1892 008b}$$

Further, we encrypt the plaintext generated by the trigger application directly and do not perform the hashing operation as described in the protocol. The reason for this is that the hashing generates more noise and makes the comparison between the different AES implementations less clear. Nevertheless, we provide the measurement result with the full protocol implementation exemplary for the AES implementation of Bernstein [Ber05a] in Testbed1. However, noise is not really considered in our work but clearly has an impact on the measurements as we show in Section 7.4.

We generate a profile every time when additionally 100K samples for each possible plaintext byte value are observed until 2M of each such samples were reached. To generate N samples for each possible value of all plaintext bytes, approximately $N \cdot 256$ messages with 16-byte random plaintexts have to be observed.

Fig. 7.3: Prototype platform i.MX6 utilizing quad-core ARM Cortex-A9

To analyze the success rate of our timing attack, the effect of a broad range of parameters was examined. For the comparison between different values for these parameters, two criteria were used.

1. The number of different candidates for each key byte
2. The average position of the correct candidates in the ordered output lists

The first one directly gives information about how much the key space could be reduced by the attack. To quantitatively measure the effectiveness of the attack, this is therefore the best parameter. In the best case only one candidate, namely the correct one, remains for each byte and the key is hence revealed completely. But even only a significant reduction of the number of candidate bytes is already valuable to the attacker as he then can launch a brute-force attack in the reduced key space with the remaining possible values. However, this score does not use all information of the output of the attack. As the list of possible candidates for each key byte is ordered, it is interesting to know at which position in these lists the correct values can be found. This is a measure for the ability of the attack to separate the correct hypotheses from the other remaining ones. In the best case, the correct value for each key byte always has the highest correlation and is therefore at first position in the list. That information is also of high interest to an attacker as he can use this information to significantly speed up his brute-force attack. Since he knows the correlation of all remaining possible byte values, he can order the possible keys by the correlation and then test for candidates with higher correlation first. This will usually require much less than the average $\frac{n}{2}$ guesses, n being the number of key candidates. Another approach to reduce brute-force complexity could be to use the key-rank estimation procedures proposed by Veyrat et al. [VCGS13, VCGRS13] as shown by Spreitzer and Gérard [SG14].

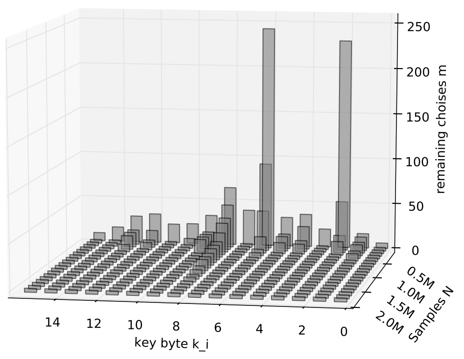
Table 7.3: Timing profile comparison between the different implementations

Implementation		Time (in cycles)		Variation				Time AES
		min	max	min	max	median	interval	(in cycles)
Barreto	offline 0	33745.96	33772.29	-9.57	16.77	-0.47	26.34	≈ 4231
[BBR00]	online k'	33745.71	33772.31	-9.87	16.73	-0.49	26.59	≈ 4230
OpenSSL	offline 0	33584.26	33605.61	-8.04	13.31	-0.16	21.35	≈ 4222
[The11]	online k'	33585.64	33607.81	-8.99	13.18	-0.14	22.17	≈ 4221
Bernstein	offline 0	33731.61	33778.54	-11.44	35.49	-0.94	46.93	≈ 4546
[Ber05a]	online k'	33745.04	33781.29	-5.24	31.00	-0.78	36.24	≈ 4573
Gladman	offline 0	35139.63	35158.00	-6.26	12.10	-0.16	18.37	≈ 5689
[Gla08]	online k'	35139.48	35157.03	-5.72	11.82	-0.16	17.55	≈ 5689
Niyaz	offline 0	59266.99	59280.43	-8.39	5.05	0.03	13.44	≈ 24840
[PK]	online k'	59265.01	59278.61	-8.88	4.72	0.01	13.60	≈ 24834

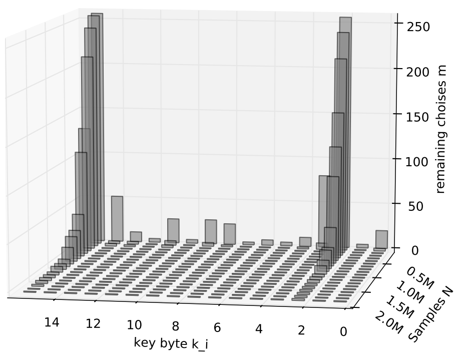
7.3.4 Results for Testbed1

We evaluated a broad range of different AES implementations as shown in Table 7.3. The implementations of Bernstein [Ber05a], Barreto et al. [BBR00] and OpenSSL [The11] are optimized for 32-bit architectures like the Cortex-A8 whereas Gladman's [Gla08] is optimized for 8-bit micro controllers. Niyaz' [PK] implementation is totally unoptimized. As there are many OpenSSL versions in use, for our tests we explicitly used the C implementation of Version 1.0.0d. Due to the massive amount of samples we took over three weeks, we parallelized the measurements to several different BeagleBoards. This also could have an influence due to different Versions of u-boot were deployed, depending on the production charge.

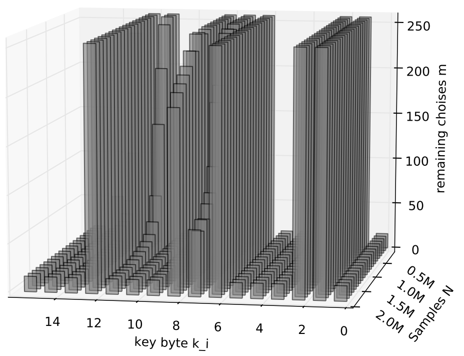
Table 7.3 visualizes the online and offline profile of each implementation. The first column shows the minimum and maximum of the overall timing in CPU cycles which is used for the correlation. The second column shows information about the variation of this timing computed over all measurements. To make propositions over the signal to noise ratio, we also provide the average time spent in the AES encryption method. In Figure 7.4, the result of the correlation is shown. The plots depict the decreasing possibilities for each key byte by increasing samples. For each implementation, a subfigure is provided which plots the left choices m with $m \in]0; 256]$ in z-direction for each key byte k_i with $i \in [0; 15]$ from left to right, while the amount of samples N for the online profile with $N \in [100K; 2M]$ is plotted in y-direction from behind to front. For this result, a constant sample amount of 2M was used for the offline profile with the null key.



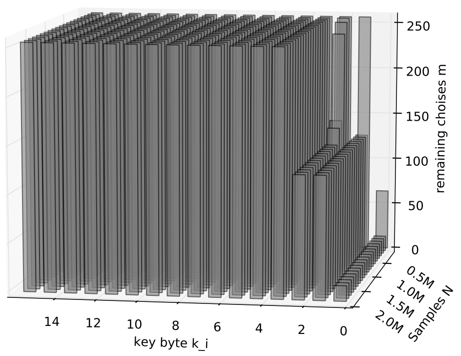
(a) Barreto



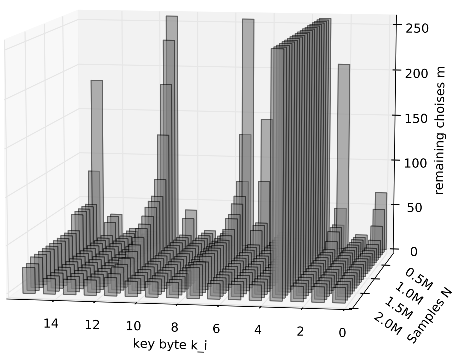
(b) Bernstein



(c) Gladman



(d) Niyaz



(e) OpenSSL

Fig. 7.4: Reducing key space by timing attack of different AES implementations

Barreto

The implementation of Barreto et al. [BBR00] which is part of many crypto libraries is showing a high vulnerability against this time-driven attack. Barreto uses four lookup tables, each of 1 KByte in size. Thus, the lookup tables do not fit into one cache line. Additionally for the last round, a fifth lookup table is used. This type of implementation is also called T-Tables implementation which is described in Chapter 2.4.3. After 100K samples, only key byte 3 and 7 have more than 200 possibilities left and for key byte 9, the choices are above 50. The other 13 key bytes are all below 50. After 800K almost any key is pinpointed to 4 choices except key byte 9. However, this seems to be the limit for this implementation. That means, using additional samples do not improve the results any further. After 1.6M samples also for key byte 9 the limit is reached and only 4 choices are left. Nothing changes afterwards until 2M samples are reached. See Figure 7.4a.

OpenSSL

The OpenSSL implementation [The11] is almost the same as the implementation of Barreto et al. However, the results of both implementations differ. For the OpenSSL implementation, the limit is reached at 16 choices per key byte. Furthermore, the attack was not able to reduce the key space for key byte 4 at all. One could believe that the results of Barreto's implementation and the results of OpenSSL have to be the same as the encryption function is exactly performing the same operations. However, as listed in Table 7.3, the overall time which is measured during the attack is about 200 cycles higher for Barreto et al.'s implementation because of the encryption function definition. Barreto et al. pass parameters by value which are passed by reference in the OpenSSL encryption function header. Also the performed operations outside the measurement in the trigger application influences the cache evictions. In total, this causes more cache evictions and thus a higher variation of the AES signal, resulting in better correlation behaviour. Another explanation could be that we measured on different boards where the boot loader configured the frequency of the processor differently than the other.

Gladman

The same holds for the implementation of Gladman [Gla08] which we compiled with tables and 32-bit data types enabled. Here, also the choices for several

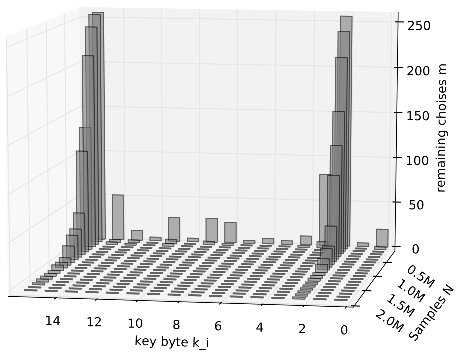
Table 7.4: Correlation results after 100K samples of online profile received with the C version of Bernstein's AES implementation; offline profile with 2M samples

Choices	Byte#	Key Values
		$\longleftarrow$ higher correlation **c**
20	0	**21** 20 23 22 fc 25 26 ..
4	1	**53** 52 51 50
256	2	**fc** cb 9b a1 fd a6 a4 ..
80	3	**73** 70 76 71 75 74 72 ..
10	4	**d4** d6 d5 d7 d3 0a df ..
4	5	**f3** f1 f0 f2
6	6	**4a** 49 4b 48 4f 4d
3	7	**98** 9a 99
23	8	**17** 15 ce c9 13 12 ca ..
27	9	**33** 31 32 ec ea 30 ed ..
4	10	**bb** b8 ba b9
27	11	**3f** 3e 3c 3b 3a e2 e5 ..
4	12	**18** 1b 19 1a
11	13	**92** 90 91 93 97 96 9a ..
51	14	**00** c0 01 02 20 e9 21 ..
256	15	**8b** 06 93 8f 33 b3 0f ..

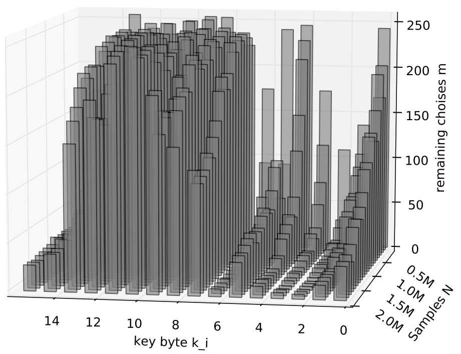
key bytes are reduced to 16 possibilities. However, Gladman uses both the Rijndal SBoxes the SBox S_{RD} and its inverse S_{RD}^{-1} for encryption. However, the concatenation of the SBoxes which results in the T-Tables implementation is done on demand only partially for the current state computation. Therefore, the performance is comparable to the T-Tables implementation but less memory expensive. Thus, only two 256-byte lookup tables are stored in memory which means the signal to noise ratio is even worse than in the other large table-based implementations. Further, there are less competitive lines for the lookups reducing cache evictions by AES itself drastically. Even though other variables used during the computation can compete with the same lines in cache, this reduces the amount of cache evictions a lot in comparison to the 4 KByte tables implementations. So, there is no reduction of the key space for four key bytes at 2M samples.

Niyaz

The implementation of Niyaz [PK] seems almost secure against this attack as shown in Figure 7.4d. Niyaz implementation is the reference implementation of Rijndal, described as SBox implementation in Chapter 2.4.3. This implementation uses also only the 256 byte Sbox S_{RD} for encryption. Further, as the cache is 4-way associative with a cache line size of 64 byte, the lookup table fits into one cache block at once. This makes evictions by AES itself nearly impossible. Thus, the timing leakage generated by the SBox lookups is reduced. Further, the unoptimized code beside the table lookups in the encryption method will decrease the signal-to-noise ratio to make it even harder to extract information from the measurements using the correlation.



(a) Bernstein



(b) Bernstein with hashing

Fig. 7.5: Impact of noise caused by hashing for full mutual authentication protocol using Bernstein's Poly1305-AES

Bernstein

Our results show that Bernstein's AES implementation is most vulnerable to our cache timing attack. However, we used the C compatibility version which is part of his Poly1305-AES [Ber05a] message authentication code since no ARM implementation is available. This implementation is the only one which totally leaks the secret key k'. Already after 400K samples, the key is almost completely recovered by the correlation and only 2 key bytes need to be computed using brute-force. Further, during the correlation phase, the possible key bytes are sorted by the correlation. Thus, already after 100K, the correct key k' can be extracted as shown in Table 7.4. The first column of Table 7.4 shows the possible choices which are left after correlation. In the second column, the corresponding key byte index is listed while the third column shows the key values sorted by their correlation c. The values with highest correlation are also the correct bytes of k' we introduced in this section. The correct values are printed bold in the table.

For this implementation, we also executed the attack with the full mutual authentication protocol, with hashing enabled. We used the reference SHA1 implementation of the L4Re crypto package. In Figure 7.5b, it can clearly be seen that the additional noise generated by the hashing function increases the amount of samples needed for the attack significantly.

Table 7.5: Summary of results for different scheduler configurations

No.	Utilized Cores	Scheduler Configuration	Average Position	Remaining Key-space
1	1	1 Core shared (Single)	3.625	$\approx 2^{80.2}$
2	4	4 Cores shared (Quad)	4.25	$\approx 2^{81.7}$
3	4	4 Cores Server, 1 core shared rich OS (Server)	4.0	$\approx 2^{82.8}$
4	2	1 Core dedicated each	4.0	2^{72}
5	4	2 Cores dedicated each	4.375	$\approx 2^{72.7}$

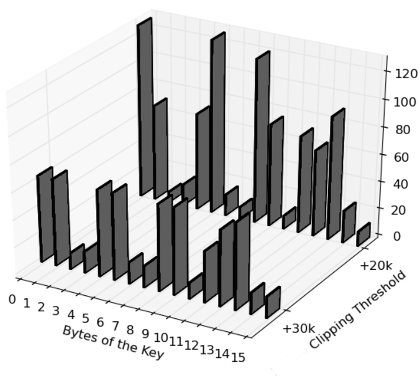
7.3.5 Results for Testbed2

We used Testbed2 mostly to analyze several multi-core configurations of the PikeOS scheduler. For all the experiments summarized in Table 7.5, normal priority-based scheduling was used and the profiling and attack phase were done on the same device. This might not always be possible in a real-world setting, but was done to have an optimal setting for the evaluation. If not stated otherwise the attacked key was and both profiling and attack phase were conducted with 512 million samples to have approximately 2 million samples for each possible key candidate as in the section before for Testbed1.

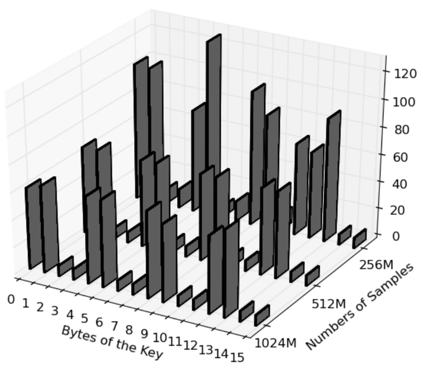
Identifying and Tuning of Attack Parameters for Testbet2

To reduce the noise in the measurements, Bernstein disregards all measurements above a certain threshold. In the original code, this threshold was set to a value fitting the timing behavior of his implementation. This value was therefore changed in this implementation. To evaluate the effect of this clipping, two different thresholds were investigated both with 512 million samples for attack and profiling phase. The threshold that was initially set to about 30,000 clock cycles higher than the average of the timing samples was compared to the threshold 20,000 above average. The results are displayed in Figure 7.6a and Table 7.6a. The results clearly show that the lower threshold leads to a significant lower reduction of the key space. This implies that the timings lying in the interval between the two thresholds indeed contained information about the key. This also complies to the findings in [SG14], which show that the minimal timing attack of [AE13] does not leak any information on ARM.

One parameter that comes to mind very quickly when thinking about analyzing a side channel attack is the number of samples. One would assume that an increasing number of samples automatically results in a higher success rate as the noise gets averaged out more and more, leaving only the relevant information behind. To verify this assumption, we conducted the attack with 256, 512



(a) Clipping thresholds



(b) Increasing samples

Fig. 7.6: Histograms describing the numbers of possible candidates for all bytes of the key and for varying clipping thresholds and samples

and 1024 million samples. The results are displayed in Figure 7.6b. Table 7.6b shows the average position of the correct key byte candidates. As expected, increasing the number of samples does in fact also increase the success rate of the attack. However, the increase of the success rate shows a logarithmic behavior. This behavior is derived directly from the cache architecture. As only the upper k bits of a data word are used to index the cache lines, the timing behavior is independent of the lower bits. In the best case, the attack could therefore only reveal the upper k bits of each key byte. This explains the observed boundary of the reduction of the key space. It furthermore explains why the remaining number of possible values per byte is in almost all cases a power of 2. A similar behavior was described by Neve et al. [NSW06].

Single-core vs. Quad-core

The PikeOS scheduler allows the use of a CPU mask to specifically select the cores that shall run a partition. As each core has its own L1 cache but all cores share the L2 cache, it is interesting to examine how the success rate of the attack changes when only one or all cores are used. To do this, three different configurations were regarded. For the first one both partitions were run by a single shared core (configuration 1) while for the second one both partitions were run on all four cores (configuration 2). The third configuration involved the AES server running on all four cores (configuration 3) while ELinOS was assigned only one core. Note that the AES server application

Table 7.6: The average position of the correct key byte candidates for the different clipping thresholds and numbers of samples

(a)

Clipping Threshold	Average Position
+30k	3.625
+20k	4.3125

(b)

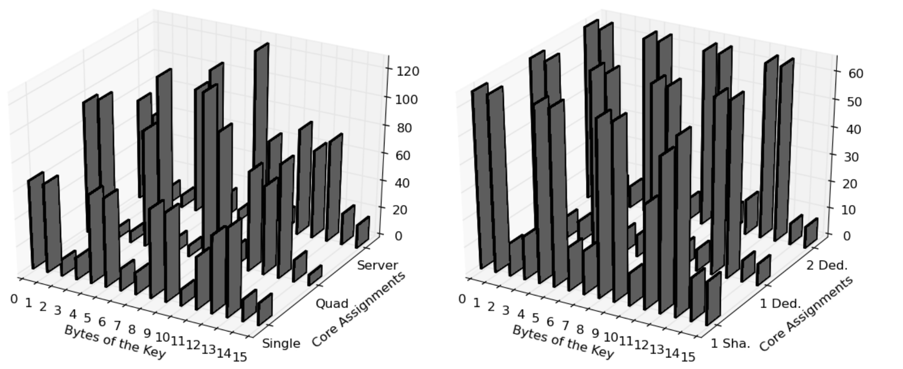
Number of Samples	Average Position
256M	5.5625
512M	4.0
1024M	3.875

itself does not implement any concurrent block computation. The results are depicted in Figure 7.7a and Table 7.5.

It can be seen that configuration 1 gave the best results for both criteria, and scenario 2 yielded the worst. This is understandable since in the first scenario, the T-tables are stored in a single L1 cache and the L2 cache, whereas in scenario 2 the T-tables are most likely scattered over the four L1 caches and the L2 cache. This decreases the signal to noise ratio with high certainty and thus lowers the success rate. Additionally, when both the rich OS and the AES server use the same core, their cache usage will interfere which also reduces the quality of the timing samples. This effect is visible in the difference between scenarios 2 and 3. Although the AES server uses four cores in scenario 3 as well, it only interferes with the other application in one of them which leads to an overall better success rate of the attack.

Dedicated Cores

Using a dedicated core for the AES server is expected to be not advisable because it reduces the noise. Therefore, it was investigated how the success rate of the attack is affected when the two partitions have one or two cores for their own in comparison to configuration 1 where both partitions share only one core. The results are shown in Figure 7.7b. The use of dedicated cores leads to a significantly better success rate in terms of the total number of remaining key candidates. The setup with one dedicated core also shows a slight decrease in the average position of the correct key byte values. As it can be seen, assigning one core to each partition (configuration 4) thereby results in a slightly better attack result than using two dedicated cores (configuration 5). This can be explained by the already discussed effect of using multiple L1 caches. However, the slight increase of the average position compared to the scenarios where 4 cores are utilized seems to be caused by measurement inaccuracies. In summary, when using an ordinary priority-based scheduling



(a) Shared scheduler configurations **(b)** Dedicated scheduler configurations

Fig. 7.7: Histograms describing the numbers of possible candidates for all bytes of the key and for varying scheduler configurations.

scheme on a multi-core system without any countermeasures, it is not recommended to use a dedicated core for the cryptographic algorithm as this would reduce the noise significantly.

7.3.6 Comparing Results of Testbed1 and Testbed2

In this section we compare the results from the OpenSSL implementation of Testbed1 (mobile device) and Testbed2 (cyberphysical system). This is more or less a comparison between the CPU architectures and the utilized microkernel frameworks. Note that we use the same key for both setups as described in Section 7.3.3 to provide comparable results. For the Cortex-A8-based Testbed1, we were able to reduce the byte value space of almost all bytes to 16 possibilities for the OpenSSL implementation of AES. For byte 3, no reduction was possible which seems to be an error of measurement, while bytes 7, 11 and 15 could only be reduced to 32 respectively 24 possible values. This result was achieved with 2 million samples for each byte value, translating to the overall number of 512 million samples.

The best result achieved in terms of the reduction of the key space for the Cortex-A9-based Testbed2 draws a very different picture. For one dedicated core for the ELinOS and the AES partition respectively, the highest reduction found was a reduction down to 8 possible values for the bytes 2,3,6,7,10,11,14,15. For the remaining bytes a reduction was possible only down

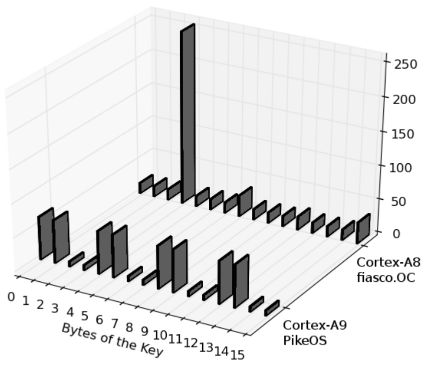


Fig. 7.8: Results on Cortex-A8 (Testbed1) compared to Cortex-A9 (Testbed2)

to 64 different values. This pattern is interesting in itself as every consecutive 2-byte tuple seems to be highly correlated in the reduction capability. However, it is also very different from the result stated above. For this implementation, the maximally achieved reduction is twice as high as for Testbed1. Nevertheless, only half of the bytes could be reduced that far while for Testbed1 nearly all byte spaces could be reduced to the respective minimum. Then again, in the Testbed2 all bytes could be reduced to at least 64 different values. This was not the case for Testbed1 which is using the Fiasco.OC kernel. Both implementations have in common that there seems to be a limit for the reduction of the key space that depends on the implementation. This was already mentioned above. The two results are compared in Figure 7.8. The difference of the reduction pattern reflects the different cache architectures in terms of the cache line size. On the Cortex-A8 with a 64 Byte cache line size every fourth key byte is harder to reduce, while on the Cortex-A9 with 32 Byte cache line size every first 2 bytes are harder to reduce. Both pattern repeat every 4 bytes, this is due to both caches are 4-way associative.

The total number of possible keys was reduced to 2^{72} for the worst PikeOS setup (Testbed2) and to roughly 2^{70} for the Fiasco setup (Testbed1) showing a slight advantage for the PikeOS setup. However, compared to the setup utilizing only one core with a reduction of key space to $\approx 2^{80.2}$, the PikeOS setup is about three orders of magnitude harder to attack.

7.4 Reproducing Bernstein's Attack in local Environment

Bernstein stated in [Ber05b] that modern cache-architectures are vulnerable against statistical timing attacks which we now called time-driven cache attacks. He showed a practical attack setup on an Intel Pentium III based machine as shown in Figure 7.9. Nevertheless, he provides a rather unrealistic measurement setup. Instead of really attacking the remote host from the client, he let the server under attack provide the timings. Thus, the noise of the operating system and the network channel is completely eliminated.

Thus, our first approach was to run his attack locally, measuring the timing in the client application. For that purpose we came up with the following attack setup shown in Figure 7.10. Instead of using two separate machines, we just use the same kernel connecting the client and server application over the Linux kernel's loopback network interface.

Further, as first virtualization setup we thought about a full system virtualization using two para-virtualized L4Linux machines on top of the Fiasco.OC microkernel. Both VMs using the Linux kernel's network stack on top of a virtual ethernet interface communicating over a shared memory with each other. This setup is shown in Figure 7.11.

However, it turned out that the operating systems' network stack and scheduling introduced too much noise to get viable results in both cases. For the Linux native setup, we at least got a small reduction of the key space as shown in Figure 7.12.

For reference values, we performed the same measurements for the Linux localhost scenario on our Testbed1 hardware platform using the same parameters as in Section 7.3. Hence, we measured the same AES implementations of Bar-

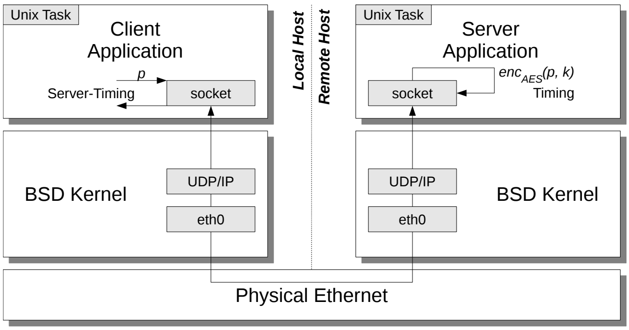


Fig. 7.9: Original attack setup of Bernstein using two Unix machines

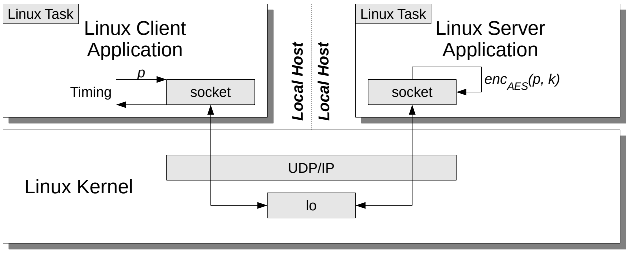


Fig. 7.10: Linux client application connecting through loopback device to Linux server application

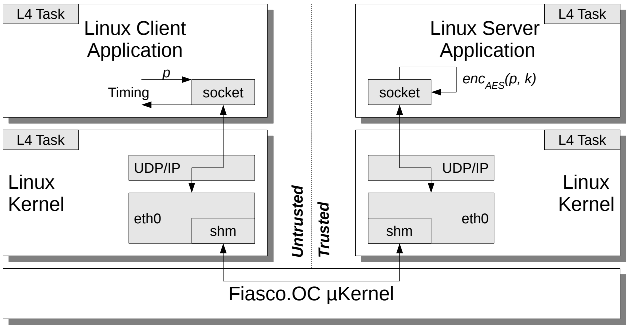
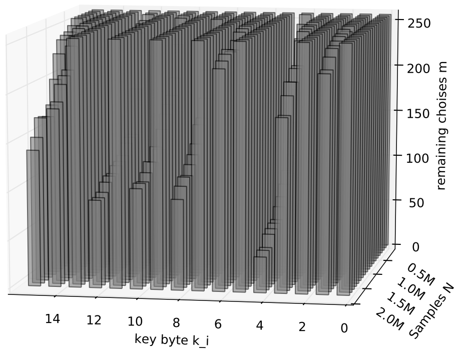


Fig. 7.11: Linux client application connecting through virtual ethernet of two L4Linux instances to Linux server application

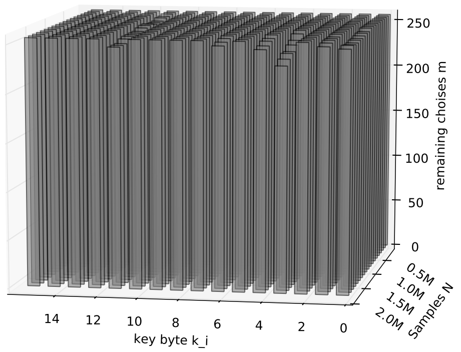
reto, Bernstein, Gladman, Niyaz and OpenSSL. The results are summarized in Figure 7.12 for the adapted original setup with timing measurement in the server respectively Figure 7.13 with timing measurement in the client. What we clearly can see is that the use of the Linux kernels loop back network device and UDP/IP stack already introduce many noise which makes a key recovery almost impossible. The *shm* network interface of L4Linux used in the virtualization setup with two L4Linux machines introduces even more noise as there are also two user space para-virtualized Linux kernels involved. As a result we did not complete our measurements with any reduction of the key space.

We also conducted the attack in Bernstein's original setup shown in Figure 7.9 on our Testbed1 hardware. Instead of BSD we used a Linux 3.0 distribution on two interconnected BeagleBoards. We used a direct link-to-link connection without any switch in between the two hosts. However, the attack was not reproducible in this setup. This is similar to what Neve et al. [NSW06] also stated for their experiments with the original x86-Setup with client timing measurements.

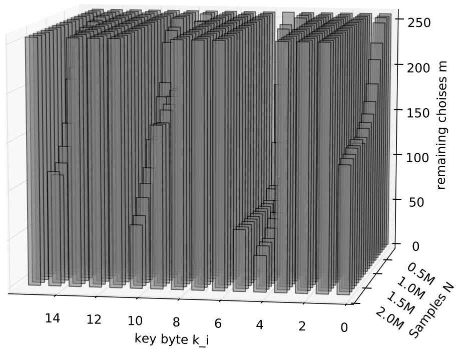
Nevertheless, these steps brought us from the rather unrealistic attacker model of Bernstein's original attack setup to the former described realistic attack not only applicable on our microkernel-based TEE architecture, but also on other TEE architectures.



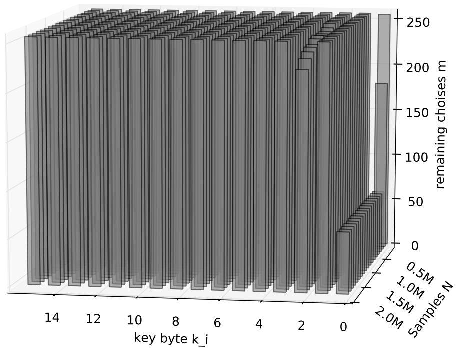
(a) Barreto



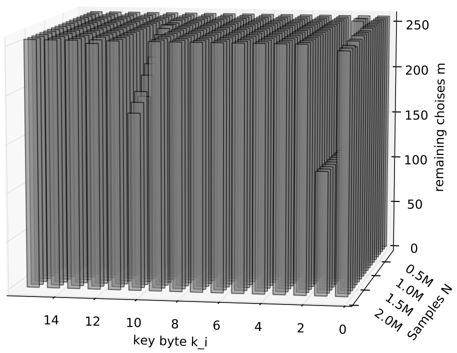
(b) Bernstein



(c) Gladman

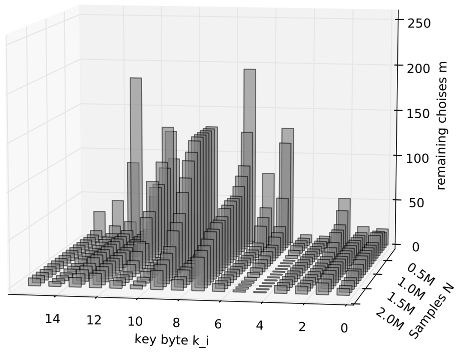


(d) Niyaz

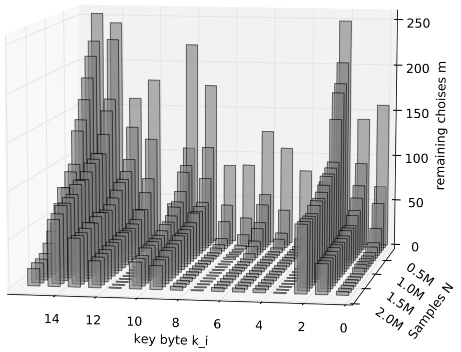


(e) OpenSSL

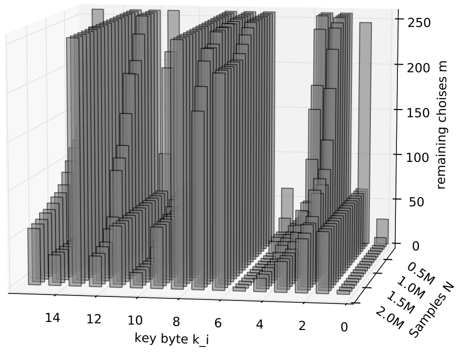
Fig. 7.12: Reducing key space in local host setup (client timing) of different AES implementations



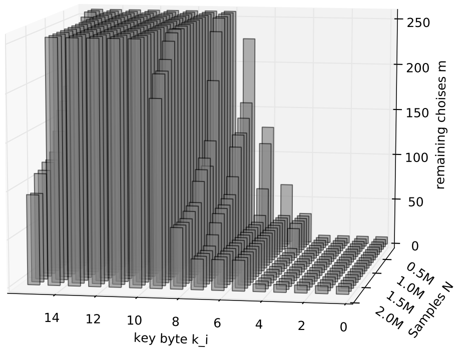
(a) Barreto



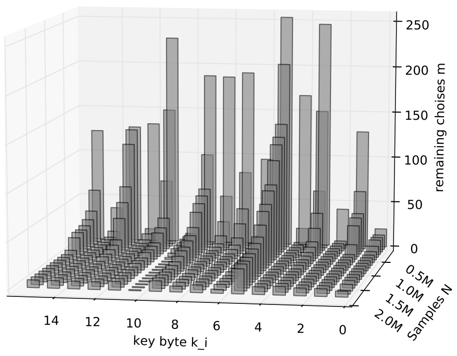
(b) Bernstein



(c) Gladman



(d) Niyaz



(e) OpenSSL

Fig. 7.13: Reducing key space in local host setup (server timing) of different AES implementations

7.5 Countermeasures

The first countermeasure we will discuss in this section is the RMC-AES which we more or less see as a dead end due to performance issues. However, for single operations in authentication protocols it may still be an option. As second one we provide the *discrete-time* countermeasure which is very cheap regarding development costs as it is only a scheduler configuration. We published this also in Contribution [WWAS15]. Further, we discuss similar approaches of others.

7.5.1 RMC-AES Counter Mode

To prevent the timing side channel we came up with the idea of a randomized algorithm executed on a multi-core platform which we call RMC-AES randomized multi-core AES. In contrast to other approaches like bit-slicing which produces a constant time execution due to completely avoid table lookups, our approach does not completely redesign the algorithm but splits the table lookups randomly across the available cores. Thus, our approach works platform independent as the original T-Tables implementations, while bit-slicing only works for platforms which provides 128 bit registers and SIMD like Intel MMX[2]. Even though, for newer ARM architectures the NEON extension also provides 128 bit registers and instructions, for embedded platforms in general it need not be the case that such CPU architecture extensions exist.

Design

Usually a T-Tables implementation does 4 operations including XOR ($\oplus$) and table lookups in one round of AES, see Chapter 2.4.3. These operations are independent. As a consequence we could execute them concurrently on different CPU cores using the CPU cores' individual cache for the lookups. Thus, if we randomize those lookups across the available cores, we are able to produce a random lookup pattern for each of the cores resulting in an unpredictable cache timing behavior. This idea is sketched in Figure 7.14.

However, we have to do a synchronization between the cores after each round as the result is needed for the next round. This is an expensive operation and thus, we came up with the idea to combine the approach with the counter mode of AES which allows us to encrypt multiple blocks in parallel. We now

[2] Intel MMX: Intel Multi Media Extension

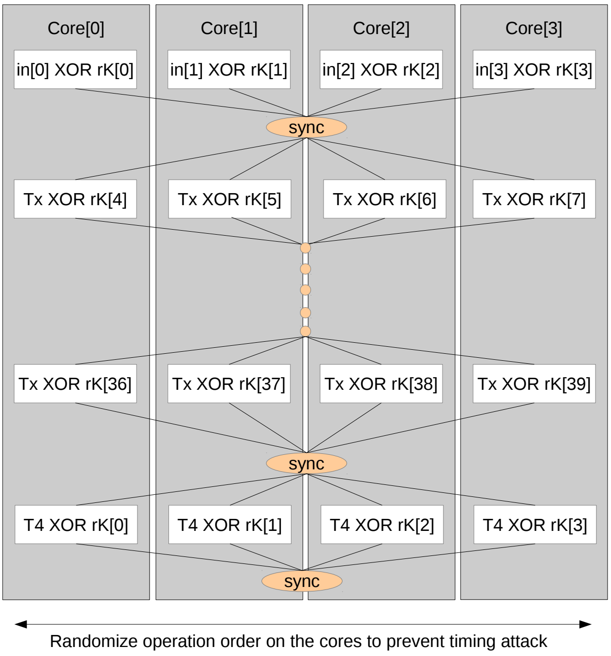


Fig. 7.14: Encrypting one block AES with randomized lookups

concurrently compute a fixed number N of AES blocks. We define those N AES blocks as the RMC-AES block. Furthermore, N is a security parameter of the algorithm. A higher value of N produces more different timing patterns for each AES key byte, namely $N \cdot 4! \cdot 10$ while $4! \cdot 10$ are the possible timing profiles for the generic approach without the additional ctr mode.

We introduce an intermediate lookup table for block indices denoted as CTR which we can use for counter and input output mapping. This index lookup table is then shuffled before each RMC-AES block computation. As a result, the temporary memory consumption compounds as follows, three arrays, input output and index array with each of the size $N \cdot \texttt{AES\_BLOCK\_SIZE} = N \cdot 16$. To reduce indices in the description of Algorithm 4, we denote this step as Shuffle() with those three arrays as parameters which are shuffled simultaneously. During each round the algorithm only computes one of the partial state word $s_o^{r,i}$ while o denotes the operation/index of the total status word and r the corresponding round as described for the original T-Tables implementation (see definition in Chapter 2.4.3), i is the index of the corresponding AES block inside the RMC-AES block.

We denote the capitalized parameter $P = P^0, P^1 ... P^N$ as the total input plaintext RMC-AES block while C is the same size output block $C = C^0, C^1 ... C^N$.

Algorithm 4 RMC-AES encryption

```
 1: procedure $enc_{RMC\_AES}(P, Nonce, k)$
 2:     K = KeyExpansion(k)
 3:     Generate($CTR$)
 4:     Shuffle($CTR, C, P$)
 5:     for $i$ in $core * N/4$ to $(core + 1) * N/4$ do
 6:         $CTR^i = ctr128\_get\_counter(Nonce, i)$
 7:         $o$ = select(0 to 3)
 8:         $\mathbf{s}_o^{0,i} = \mathbf{ctr}_i \oplus \mathbf{K}_o^0$
 9:     for $r$ in 1 to 9 do
10:         for $i$ in $core * N/4$ to $(core + 1) * N/4$ do
11:             $o$ = select(0 to 3)
12:             $\mathbf{s}_o^{r,i} = T_0[s_{0+4*o}^{r-1,i}] \oplus T_1[s_{5+4*o}^{r-1,i}] \oplus T_2[s_{10+4*o}^{r-1,i}] \oplus T_3[s_{15+4*o}^{r-1,i}] \oplus \mathbf{K}_o^r$
13:             ($with\ all\ index\ computations$ mod 16)
14:     for $i$ in $core * N/4$ to $(core + 1) * N/4$ do
15:         $o$ = select(0 to 3)
16:         $\mathbf{s}_o^{10,i} = T_4[s_{0+4*o}^{9,i}] \oplus T_4[s_{5+4*o}^{9,i}] \oplus T_4[s_{10+4*o}^{9,i}] \oplus T_4[s_{15+4*o}^{9,i}] \oplus \mathbf{K}_o^{10}$
17:         ($with\ all\ index\ computations$ mod 16)
18:         $C^i = P^i \oplus S^{10,i}$
19:     return $c$
```

Prototype Implementation

We implemented the above described algorithm, by replacing the `aes_ctr_mode` module in OpenSSL. The used OpenSSL version was 1.0.1c. As described above, our performance concerns were justified. We tried to minimize overhead by introducing thread pools which are synchronized with barriers. We used a fast barrier implementation by Lockless Inc.[3], as our synchronization primitive. Even though, barriers are not the fastest synchronization mechanism, the performance of our algorithm implementation is quite slow in general. This is due to we contradict any CPU architectural constraints here. Usually, in CPU architectures there are branch prediction and pipelining. As our implementation intentionally only executes partially on each core, which is also decided at runtime on randomly shuffled index lookups, those mechanisms do not provide good results anymore. The performance results are depicted in Figure 7.15. We did the measurement on our multi-core platform of Testbed2.

[3] http://locklessinc.com/articles/barriers/

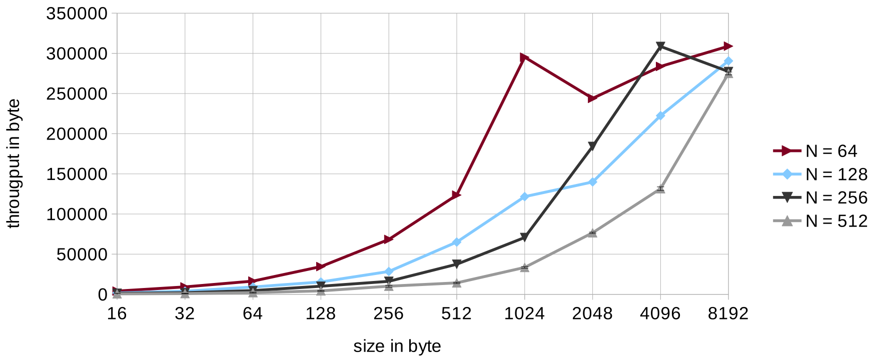


Fig. 7.15: Performance of RMC-AES for different values of N

However, we measured performance on a Linux system based on Linux 4.0.1 using the OpenSSL speed application. As one can see the best results are achieved for 4k buffers and $N = 256$ with about 300kB/s. This is a good choice anyway because this is usually the size of a memory page. For instance, our AES server in the trusted execution environment in both of the above prototype implementations for PikeOS and L4Re uses exactly one 4k memory page as input and output buffer.

7.5.2 Discrete-time Countermeasure

One main pitfall of novel countermeasures is that some of them require changes to already established systems that are too substantial to be easily implemented, hence making these countermeasures practically irrelevant. The *discrete-time* countermeasure that is presented in the following therefore aims at making cache-based time-driven attacks infeasible for attackers while demanding as few changes and inducing as little overhead as possible. Our discrete-time countermeasure targets a PikeOS scheduler configuration, therefore we do a short re-cap of scheduling in PikeOS in the following subsection.

Scheduling in PikeOS

PikeOS features a special scheduler that uses a combination of *time-driven and priority-based scheduling* to account for the different needs of the applications. To allow for deterministic real-time responsiveness, the scheduler uses a time-driven approach. Every real-time application is statically assigned to a time slot of a defined length. The length of these time slots can vary between applications but has to stay within a certain relation to the length of the other time slots. Every application is periodically scheduled for the length of the slot it is assigned to. As every partition gets assigned a defined amount of CPU time at defined points in time, they are able to schedule real-time processes themselves. This, so far, is a standard approach for scheduling real-time applications. To also support non real-time applications, a straightforward extension of this approach is to just create a new time slot and assign all applications without timing constraints to it. Within this slot, a standard round robin scheduling scheme can be applied. However, this approach is inefficient since it wastes a lot of CPU time. The PikeOS scheduler refines this approach to a more efficient strategy. It might occur that the processes of a real-time application finish before its time slot end or that it does not have any processes to run at all. As it would harm the temporal determinism, the scheduler cannot simply switch to the next application in this situation. Rather than wasting this time, the PikeOS scheduler uses this excess CPU time to schedule applications with no real-time constraints. For this purpose, it leverages priority-based scheduling. All real-time applications are assigned the same mid-level priority number while low priority numbers are assigned to the other applications. Now, the scheduler continues to schedule the real-time applications periodically but uses the excess time to schedule the low-priority non real-time applications in a round robin fashion. In this way, no computing time is wasted and the overall amount of time needed to execute all applications decreases drastically when compared to a standard RTOS scheduler.

Configuring the Scheduler

Assume the rich OS and the trusted environment are implemented as partitions in PikeOS and are hence handled by the scheduler. Now assume the attacker has compromised the rich OS and is able to launch the timing attack against the AES server that runs in a trusted partition as stated as Scenario Secure VPN in Chapter 4.2.1. In order for the attacker to successfully carry out the attack, two conditions must be fulfilled:

1. He must be able to retrieve enough samples from the AES server, in the order of several hundred millions.
2. The samples must leak enough information for the correct hypothesis on the key to yield a higher correlation on average than all wrong hypotheses.

The *discrete-time* countermeasure aims at these two points. It works straightforward in that both applications, the rich OS and the AES server in the trusted environment, are treated as real-time applications such that each is assigned an own time slot. Note, that it is not necessary for either of the two applications to have any real-time time constraints in order for the scheduler to be configured as described above. Using this configuration of the scheduler the time measured by the attack for one encryption t_{enc} is now given by Equation 7.7.

$$t_{enc} = n \cdot t_{OS} + m \cdot t_{serv} \tag{7.7}$$

with t_{OS} being the length of the time slot of the rich OS and t_{serv} being the length of the time slot of the AES server. The two variables n and m represent the number of executions of the two time slots. Note that we ignore negligible timing quantities that are independent of the AES server, such as the remaining time in the slot of the rich OS after the encryption was requested and the time passing in the first slot of the rich OS after the encryption is done before the attacker's process is scheduled. As it can be easily verified the time is always a multiple of the two time slot lengths which gives rise to the countermeasure's name. This has two major effects on the attack. Firstly, as the scheduling for these two applications is strictly time-driven, the rich OS will be scheduled a number of times while still waiting for the encryption to finish and hence being idle. This will increase the time needed by an encryption in a way that, given carefully chosen values for t_{OS} and t_{serv}, a single encryption as it is needed for benign purposes can still be done without noticeable delay. However, a number of encryptions as needed for an attack will take a significantly larger amount of time. This already will make an attack time-wise more difficult. Secondly, as the information that can be gained by one sample is now very coarse-grained, there is only a very small correlation left between the timing information and occurring cache-misses or hits. This will make it very hard for the attacker to distinguish the correct key hypothesis from false ones and will increase the number of necessary samples. Therefore, the discrete-time countermeasure is a strong shield against the kind of attacks considered here. Furthermore, the countermeasure requires no change of any kind in the code and also causes arguably only little timing overhead. It is also straightforward to implement, can be extended to multiple applications and is most likely also applicable to other RTOS schedulers working in a similar manner as the PikeOS scheduler. Although not in the focus of this work, access-driven attacks can be prevented similarly by a simple configuration in the scheduler to flush the cache when switching partitions.

7.5.3 Evaluation of the Discrete-time Countermeasure

To evaluate the effectiveness of the discrete-time countermeasure, a range of different scheduler configurations was tested. The ELinOS and the AES server partition were assigned one time slot each and the length of these slots was

Table 7.7: Performance comparison of the countermeasure

Scheduling Scheme	Average Clock Cycles per AES-block	
	one block (16 Byte)	one page (4 KByte)
Priority-based	$\approx 125,000$	≈ 1770
Discrete-time	$\approx 210,000$	≈ 2286

then varied. It was quickly found that the length of both slots would have to be in a certain relation in order to ensure that the rich OS and the AES server work correctly. One configuration that led to a behavior of the system indistinguishable from the behavior with simple priority-based scheduling was found to be to set the slot length to 5 ticks for both partitions. The default duration of one tick was set to 1 ms. Using this configuration, the delay of single AES encryptions increases significantly by roughly about 70% while in contrast the encryption of a whole buffer with the size of a memory page may be conducted with only a small overhead of less than 30%, see Table 7.7. This configuration was therefore chosen for the attack. Both partitions were assigned one dedicated core and the rest of the setup remained unchanged from previous experiments.

After running the profiling phase for one day we were able to retrieve ≈ 34 million samples. To capture the whole amount of 512 million samples for both phases, this means a total run-time of about one month for the above configuration of the scheduler. Remember that due to the different timing behavior induced by the countermeasure an even higher number of samples is needed in order to recover the key as good as possible. Therefore, it is reasonable to assume that for the attack to produce a useful output at least twice the number of samples and hence, with the overhead caused by our countermeasure, even more than twice the time is needed. Even if the attacker would do the profiling phase off-line, he would still need to be able to access the system for about one month. It is very unlikely that such a computational intensive attack would remain unnoticed for the entire time frame. Furthermore, depending on the actual use of the AES server, a rescheduling of the key might occur during that time, too. It can be seen from this that the proposed countermeasure indeed protects a device very well while simultaneously requiring almost no effort to be set in place. Also, the user experience does not change with the countermeasure which might be an important factor for the mobile device market. The different run-times of one encryption for priority-based scheduling and the countermeasure are shown in Table 7.7. For a more thorough evaluation of the discrete-time countermeasure, additional experiments need to be conducted.

Comparison to other Countermeasures

In [KPMR12] and [SBY$^+$13], two novel countermeasures against cache-based attacks are introduced. Since these countermeasures target the same class of attacks as the discrete-time countermeasure, it is interesting to compare their approaches with ours. As the focus of this work was put on time-driven attacks, the comparison will focus on this aspect as well.

The STEALTHMEM countermeasure [KPMR12] tries to prevent both active and passive time- and access-driven attacks in virtualization environments. To that end, it uses dedicated cache lines in the shared cache for each CPU. Depending on the variant of STEALTHMEM used, this either reduces the total available amount of memory and shared cache, or it takes extra time to ensure that the stealth cache lines are not evicted from the cache. Both variants imply a small penalty in performance of about 5.9% and 7.2% respectively, and AES encryptions of 50,000 bytes are about 5% slower with the first variant. Unlike the STEALTHMEM approach, the discrete-time countermeasure has no impact on the available cache and system memory. However, due to the larger time slots in our countermeasure, the overall performance degrades by about 30% for AES encryptions on 4 KB of data, as explained above. To use STEALTHMEM, the hypervisor is extended with a special driver offering an API to the VMs that manages access to the dedicated cache lines. For Windows Server 2008 R2 with Hyper-V, this amounts to 5,000 lines of C code to be added to the hypervisor and 500 lines of C code added to the Windows boot loader modules. Furthermore, the implementations of cryptographic algorithms have to be modified to make use of the stealth cache lines via the provided API. For using our discrete-time countermeasure on the contrary, only a reconfiguration of the scheduler is needed. Neither the system nor the implementation of the cryptographic algorithm has to be changed. Also note that the required modification of the algorithm presents a potential pitfall. If not done correctly, some leakage remains and therefore breaks the countermeasure. Moreover, the amount of available cache lines that can be reserved for a core is limited so it has to be made sure that all relevant lookup tables fit inside to prevent information leakage.

The instruction-based scheduling scheme suggested in [SBY$^+$13] aims at preventing cache-based attacks that exploit certain scheduling-induced race conditions between processes that arise due to the dependency of the execution time on the cache content. Both methods are similar in that they use a fixed value as their criterion for the scheduling. As the name implies, instruction-based scheduling uses a specified number of executed instructions as scheduling criterion. This prevents only those attacks that try to exploit the mentioned race conditions – but only when the processes are run on a single-core. Furthermore, it is not sufficient to prevent time-driven attacks such as Bernstein's, since an attacker can still measure the total execution time which still depends on the cache. The discrete-time countermeasure on the other hand prevents this kind of race conditions even with multiple cores, and masks the overall execution time of an AES encryption. Also, instruction-based scheduling is a novel approach and hence not widely supported by current microkernels. Therefore, extra effort has to be done to integrate it into existing systems or implement a new one which supports instruction-based scheduling. This is not the case for the discrete-time method, where our countermeasure can be readily configured. With respect to the overhead, both methods are fairly similar as they do not need any adaption of the applications and only induce a small time overhead.

7.6 Summary

In this chapter, we stated an attack scenario using a time-driven cache attack against mobile devices such as smartphones (Testbed1) or embedded devices used in cyber-physical systems (Testbed2). We have shown that the isolation characteristic of virtualization environments can be circumvented by this kind of attacks. For that purpose, we evaluated the attack for different AES implementations on Testbed1 as well as different scheduler configurations on Testbed2. The results show that dedicated cores for the crypto routine provide the most timing leakage. Further, we compared the results of both virtualization-based TEEs. We showed that using a shared core, the PikeOS setup of Testbed2 is about three orders of magnitude less vulnerable in reduction of key space, but at least almost one order of magnitude in the worst configuration using dedicated cores. Furthermore, we discussed several countermeasures. We provided a randomized variant of the T-Tables software implementation as well as the scheduler-based discrete-time countermeasure against time-driven cache attacks. We conclude that the RMC-AES approach can be seen as a dead end for research on general purpose CPUs due to performance issues on these kind of hardware. However, both countermeasures indeed close the open attack vector to our integrity and secure loading protocols described in Chapter 6. Compared to other novel countermeasures, the countermeasures do not depend on any hardware, software architecture or algorithm changes. Thus, the discrete-time approach can be used as a drop-in configuration update for running CPSs, or other embedded platforms using a configurable scheduler. However, since the bit-sliced implementation for ARM NEON is available in newer OpenSSL versions since 2013, we propose to use this for ARM-based systems.

8 Conclusion

With concepts of trusted computing we can also improve the security of generic embedded and mobile devices. In this thesis, we saw an approach to merge those concepts with the provided software and hardware solutions of the major mobile working group GlobalPlatform. The use of those concepts as building blocks for our system architecture, allows us to generate a flexible generic system architecture which simultaneously supports trusted computing based applications as well as GlobalPlatform mobile applications.

A first step to use trusted computing based concepts such as the remote loading procedure which could also be used to securely update trusted components inside a trusted runtime, this thesis provided an architecture using a TPM chip connected to an embedded platform. This however, would restrict the application to new hardware designs. Therefore, as a second step to allow our concepts also to work on genuine mobile devices, we proposed an architecture relying on a SE-based TPM emulation. From an application program perspective this makes no difference to an off-the-shelf hardware TPM chip.

On the software side, this thesis provided an approach using the TPM or seTPM inside of a trusted environment for integrity and secure loading of remote binaries. The proposed architecture utilizes a microkernel framework to separate the untrusted compartment including a possible attacker from this trusted environment. As a consequence, we adopted the ideas of the Linux-based IMA [SZJvD04] being able to reduce the TCB due to our microkernel approach. Further, we showed the feasibility of the provided concepts inside a prototype implementation on an ARM-based embedded system.

The provided system architecture relies on the concept of the so called TEE which provides software isolation using a microkernel to separate the trusted compartment from untrusted user and system code. However, we saw in Chapter 7 that this isolation alone cannot totally resist our attacker model. For a strong remote attacker it is feasible to mount a cache-based side channel attack. We evaluated this in two testbeds representing current mobile and embedded system platforms using ARM-based System on Chips (SoCs).

We also addressed the question of how multiple CPU cores of modern embedded and mobile devices could affect the vulnerability to cache based side channels by an empirical evaluation in an adequate testbed using a quad-core ARM SoC.

As a consequence we have to deal with such kind of attacks to provide a leakage resilient trusted environment. We saw several countermeasures which are mounted on different levels in software and hardware. Our first approach to provide a random execution of table lookups on the example of AES is a software and architecture independent solution. Even though our prototype implementation is not performing very well on multi core CPU architectures, it requires further research if other processor architectures or hardware assisted implementations, e.g., on GPUs could improve this. Our second approach is a scheduler configuration which also may apply to other ciphers and not just the AES cipher which we used exemplary in this thesis. Until, hardware instructions such as the Intel AES-NI are not available for embedded systems, we have to stick with software solutions such as the herein provided ones trading performance. However, for newer ARM-based systems it is also possible to replace vulnerable AES implementations by the bit-sliced NEON implementation of OpenSSL inside the trusted environment. Nevertheless, this is only a specific solution and is not applicable on all generic embedded systems and other symmetric cipher algorithms.

Further Research Directions

At the time of this writing, the trusted computing community is in the process to establish a new TPM specification. The TPM 2.0 specification includes some updates to the hardware chip which provides more flexible usage concerning the cryptographic and hash engines. It is to be examined how the herein proposed concepts can be mapped to that new specification. In [PWS15], we already made a proposal on how seTPM can be used to ease the transition from TPM 1.2 applications to TPM 2.0 applications by deploying two TPM Applets sharing some resources such as PCR registers and keys. Further research on how practical this could be is necessary. In our auxiliary contribution trustme [HHV^{+}15], we elaborate another light-weight isolation approach which allows separate user lands on top of a Linux kernel on Android mobile devices. It would be an interesting research topic to elaborate on how this architecture can be combined with the herein proposed architecture. For instance using some kind of virtual TPM inside a user space Linux/Android container, or use the trusted environment to protect some assets of the *Container Management* or *Security Management* which form together the privileged virtualization layer in [HHV+15].

The whole field of cache-based side channel attacks made progress in the last years. We exemplary used a time-driven attack based on Bernstein's original attack to show the vulnerability of TEEs to such kind of attacks. However, further research has to elaborate if the current more sophisticated attacks such as access-driven ones can also provide better attack results on the herein proposed architecture. For instance the recently demonstrated attack in [LYG+15] showed that the *Prime+Probe* approach of [OST05] is practical for cloud-based shared hosted virtual server environments. This would also be interesting to be researched for our architecture. In this field, it is also interesting to use alternate techniques to generate a concrete timing pattern on an ARM-based processor architecture. Since ARM Cortex-A8, the so called performance registers were introduced. In this thesis, we only elaborated the CCNT register which provides clock cycles to generate a timing profile. However, those performance registers also include registers to explicitly count cache hits and misses which could be used to generate a cache profile during cipher execution.

List of Figures

List of Tables

List of Algorithms

Bibliography

[AE13] Hassan Aly and Mohammed ElGayyar. Attacking aes using bernstein's attack on modern processors. In Amr Youssef, Abderrahmane Nitaj, and AboulElla Hassanien, editors, *Progress in Cryptology – AFRICACRYPT 2013*, volume 7918 of *Lecture Notes in Computer Science*, pages 127–139. Springer Berlin Heidelberg, 2013.

[Aer97] Aeronautical Radio, Inc. *Avionics Application Software Standard Interface, ARNIC Specification 653*, 1997.

[AK06] Onur Acıiçmez and Cetin Koç. Trace-driven cache attacks on aes (short paper). In Peng Ning, Sihan Qing, and Ninghui Li, editors, *Information and Communications Security*, volume 4307 of *Lecture Notes in Computer Science*, pages 112–121. Springer Berlin / Heidelberg, 2006.

[ARM09] ARM Limited. *ARM Security Technology - Building a Secure System using TrustZone Technology*, prd29-genc-009492c edition, April 2009.

[ASK06] Onur Acıiçmez, Werner Schindler, and Cetin Koç. Cache based remote timing attack on the aes. In Masayuki Abe, editor, *Topics in Cryptology – CT-RSA 2007*, volume 4377 of *Lecture Notes in Computer Science*, pages 271–286. Springer Berlin / Heidelberg, 2006.

[BBR00] Paulo Barreto, Antoon Bosselaers, and Vincent Rijmen. Optimised ANSI C code for the Rijndael cipher (now AES), 2000. `http://fastcrypto.org/front/misc/rijndael-alg-fst.c`.

[BCG$^+$06] Stefan Berger, Ramón Cáceres, Kenneth A. Goldman, Ronald Perez, Reiner Sailer, and Leendert van Doorn. vTPM: Virtualizing the Trusted Platform Module. In *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15*, USENIX-SS'06, Berkeley, CA, USA, 2006. USENIX Association.

[BDH+11] Ingo Bente, Gabi Dreo, Bastian Hellmann, Stephan Heuser, Joerg Vieweg, Josef Helden, and Johannes Westhuis. Towards Permission-Based Attestation for the Android Platform. In JonathanM. McCune, Boris Balacheff, Adrian Perrig, Ahmad-Reza Sadeghi, Angela Sasse, and Yolanta Beres, editors, *Trust and Trustworthy Computing*, volume 6740 of *Lecture Notes in Computer Science*, pages 108–115. Springer Berlin Heidelberg, 2011.

[Ber05a] D. J. Bernstein. Poly1305-AES for generic computers with IEEE floating point, February 2005. `http://cr.yp.to/mac/53.html`.

[Ber05b] Daniel J. Bernstein. Cache-timing attacks on AES. Technical report, 2005.

[BFG+11] Samuel A. Bailey, 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. Technical report, GlobalPlatform Inc., 2011.

[BM06] Joseph Bonneau and Ilya Mironov. Cache-collision timing attacks against aes. In *CHES'06*, pages 201–215, 2006.

[CR10] Liqun Chen and Mark Ryan. Attack, solution and verification for shared authorisation data in tcg tpm. In *Proceedings of the 6th International Conference on Formal Aspects in Security and Trust*, FAST'09, pages 201–216, Berlin, Heidelberg, 2010. Springer-Verlag.

[CYC+08] David Challener, Kent Yoder, Ryan Catherman, David Safford, and Leendert Van Doorn. *A Practical Guide to Trusted Computing*. Number ISBN-13: 978-0-13-239842-8. Pearson plc Publishing as IBM Press, 2008.

[DR02] Joan Daemen and Vincent Rijmen. *The Design of Rijndael*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2002.

[Eck14] Claudia Eckert. *IT-Sicherheit: Konzepte, Verfahren, Protokolle*, volume 9. Auflage. De Gruyter, 2014.

[EH13] Kevin Elphinstone and Gernot Heiser. From l3 to sel4 what have we learnt in 20 years of l4 microkernels? In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 133–150, New York, NY, USA, 2013. ACM.

[GBK11] D. Gullasch, E. Bangerter, and S. Krenn. Cache Games – Bringing access-based cache attacks on AES to practice. In *IEEE Symposium on Security and Privacy – S&P 2011*. IEEE Computer Society, 2011.

[GKT11] Jean-François Gallais, Ilya Kizhvatov, and Michael Tunstall. Improved trace-driven cache-collision attacks against embedded aes implementations. In Yongwha Chung and Moti Yung, editors, *Information Security Applications*, volume 6513 of *Lecture Notes in Computer Science*, pages 243–257. Springer Berlin Heidelberg, 2011.

[Gla08] Brian Gladman, 2008. `http://gladman.plushost.co.uk/oldsite/AES/aes-byte-29-08-08.zip`.

[Glo10] GlobalPlatform Inc. TEE Client API Specification Version 1.0, July 2010.

[Glo11] GlobalPlatform Inc. GlobalPlatform Card Specification Version 2.2.1, January 2011. Document Reference: GPC_SPE_034.

[GSS$^+$07] S. Gueron, G. Stronqin, J.-P. Seifert, D. Chiou, R. Sendag, and J.J. Yi. Where does security stand? new vulnerabilities vs. trusted computing. *Micro, IEEE*, 27(6):25 –35, nov.-dec. 2007.

[Gue12] Shay Gueron. Intel® advanced encryption standard (aes) new instructions set. Revision 3.01 323641-001, Intel Architecture Group, Israel Development Center, 2012.

[HBW$^+$14] Julian Horsch, Konstantin Böttinger, Michael Weiß, Sascha Wessel, and Frederic Stumpf. Trustid: Trustworthy identities for untrusted mobile devices. In *Proceedings of the 4th ACM Conference on Data and Application Security and Privacy*, CODASPY '14, pages 281–288, New York, NY, USA, 2014. ACM.

[HCF04] Vivek Haldar, Deepak Chandra, and Michael Franz. Semantic remote attestation: a virtual machine directed approach to trusted computing. In *Proceedings of the 3rd conference on Virtual Machine Research And Technology Symposium*, Berkeley, CA, USA, 2004.

[HHF$^+$05] H. Hartig, M. Hohmuth, N. Feske, C. Helmuth, A. Lackorzynski, F. Mehnert, and M. Peter. The nizza secure-system architecture. *International Conference on Collaborative Computing: Networking, Applications and Worksharing*, 0:10 pp., 2005.

[HHL$^+$97] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Jean Wolter, and Sebastian Schönberg. The Performance of μ-kernel-based Systems. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, SOSP '97, pages 66–77, New York, NY, USA, 1997. ACM.

[HHV$^+$15] Manuel Huber, Julian Horsch, Michael Velten, Michael Weiss, and Sascha Wessel. A secure architecture for operating system-level virtualization on mobile devices. In *Proceedings of the 11th International Conference on Information Security and Cryptology (Inscrypt 2015)*, 2015.

[ISO05] ISO. Identification cards – Integrated circuit cards – Part 4: Organization, security and commands for interchange. ISO ISO/IEC 7816-4:2005, International Organization for Standardization, Geneva, Switzerland, 2005.

[JSS06] Trent Jaeger, Reiner Sailer, and Umesh Shankar. PRIMA: policy-reduced integrity measurement architecture. In *Proceedings of the eleventh ACM symposium on Access control models and technologies*, SACMAT '06, pages 19–28, New York, NY, USA, 2006. ACM.

[KEH$^+$09] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 207–220, New York, NY, USA, 2009. ACM.

[Kön08] Robert Könighofer. A Fast and Cache-Timing Resistant Implementation of the AES. In Tal Malkin, editor, *Topics in Cryptology – CT-RSA 2008*, volume 4964 of *Lecture Notes in Computer Science*, pages 187–202. Springer Berlin Heidelberg, 2008.

[KPMR12] Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. Stealthmem: System-level protection against cache-based side channel attacks in the cloud. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, pages 189–204, Bellevue, WA, 2012. USENIX.

[KS09] Emilia Käsper and Peter Schwabe. Faster and Timing-Attack Resistant AES-GCM. In Christophe Clavier and Kris Gaj, editors, *Cryptographic Hardware and Embedded Systems - CHES 2009*, volume 5747 of *Lecture Notes in Computer Science*, pages 1–17. Springer Berlin Heidelberg, 2009.

[Lie93a] Jochen Liedtke. Improving ipc by kernel design. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, SOSP '93, pages 175–188, New York, NY, USA, 1993. ACM.

[Lie93b] Jochen Liedtke. A persistent system in real use - experiences of the first 13 years. In *Proceedings of the Third International Workshop on Object Orientation in Operating Systems*, pages 2–11, December 1993.

[Lie95] J. Liedtke. On micro-kernel construction. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 237–250, New York, NY, USA, 1995. ACM.

[LLL$^+$11] Matthias Lange, Steffen Liebergeld, Adam Lackorzynski, Alexander Warg, and Michael Peter. L4Android: A Generic Operating System Framework for Secure Smartphones. In *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '11, pages 39–50, New York, NY, USA, 2011. ACM.

[Lor12] Markus Lorenz. TPM-based secure boot for embedded hypervisors. Diploma thesis, Technische Universität München, June 2012.

[LW09] Adam Lackorzynski and Alexander Warg. Taming Subsystems: Capabilities As Universal Resource Access Control in L4. In *Proceedings of the Second Workshop on Isolation and Integration in Embedded Systems*, IIES '09, pages 25–30, New York, NY, USA, 2009. ACM.

[LYG$^+$15] Fangfei Liu, Y. Yarom, Qian Ge, G. Heiser, and R.B. Lee. Last-level cache side-channel attacks are practical. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 605–622, May 2015.

[MLQ$^+$10] J.M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, A. Datta, V. Gligor, and A. Perrig. TrustVisor: Efficient TCB Reduction and Attestation. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 143–158, 2010.

[MMSS13] Fabio Martinelli, Ilaria Matteucci, Andrea Saracino, and Daniele Sgandurra. Remote Policy Enforcement for Trusted Application Execution in Mobile Environments. In Roderick Bloem and Peter Lipp, editors, *Trusted Systems*, volume 8292 of *Lecture Notes in Computer Science*, pages 70–84. Springer International Publishing, 2013.

[MN07] Mitsuru Matsui and Junko Nakajima. On the power of bitslice implementation on intel core2 processor. In Pascal Paillier and Ingrid Verbauwhede, editors, *Cryptographic Hardware and Embedded Systems - CHES 2007*, volume 4727 of *Lecture Notes in Computer Science*, pages 121–134. Springer Berlin Heidelberg, 2007.

[MPP$^+$08] Jonathan M. McCune, Bryan J. Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. Flicker: An Execution Infrastructure for Tcb Minimization. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, Eurosys '08, pages 315–328, New York, NY, USA, 2008. ACM.

[Nev06] Michael Neve. *Cache-based Vulnerabilities and SPAM Analysis.* PhD thesis, Universite catholique de Louvain, 2006.

[NIS01] Fips pub 197, advanced encryption standard (aes), 2001. U.S.Department of Commerce/National Institute of Standards and Technology.

[NKZS10] Mohammad Nauman, Sohail Khan, Xinwen Zhang, and Jean-Pierre Seifert. Beyond kernel-level integrity measurement: enabling remote attestation for the android platform. In *Proceedings of the 3rd international conference on Trust and trustworthy computing*, TRUST'10, pages 1–15, Berlin, Heidelberg, 2010. Springer-Verlag.

[NSW06] Michael Neve, Jean-Pierre Seifert, and Zhenghong Wang. A refined look at bernstein's aes side-channel analysis. In *ASIACCS*, page 369, 2006.

[OST05] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of aes. In *Topics in Cryptology - CT-RSA 2006, The Cryptographers' Track at the RSA Conference 2006*, pages 1–20. Springer-Verlag, 2005.

[PK] Niyaz PK. Advanced Encryption Standard implementation in C.

[PMP10] Bryan Parno, Jonathan M. Mccune, and Adrian Perrig. Bootstrapping trust in commodity computers. In *In Proceedings of the IEEE Symposium on Security and Privacy*, 2010.

[PWS15] Sergej Proskurin, Michael Weiß, and Georg Sigl. seTPM: Towards Flexible Trusted Computing on Mobile Devices based on GlobalPlatform Secure Elements. In *Smart Card Research and Advanced Application, 14th International Conference, CARDIS 2015, Bochum, Germany, November 4-6, 2015. Proceedings*, 2015.

[RM11] Chester Rebeiro and Debdeep Mukhopadhyay. Cryptanalysis of clefia using differential methods with cache trace patterns. In Aggelos Kiayias, editor, *Topics in Cryptology – CT-RSA 2011*, volume 6558 of *Lecture Notes in Computer Science*, pages 89–103. Springer Berlin Heidelberg, 2011.

[Ros14] Dan Rosenberg. QSEE TrustZone Kernel Integer Overflow Vulnerability. *Black Hat confernce*, 2014.

[RTSS09] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM conference on Computer and communications security*, CCS '09, pages 199–212, New York, NY, USA, 2009. ACM.

[SBY$^+$13] Deian Stefan, Pablo Buiras, Edward Z. Yang, Amit Levy, David Terei, Alejandro Russo, and David Mazières. Eliminating cache-based timing attacks with instruction-based scheduling. In Jason Crampton, Sushil Jajodia, and Keith Mayes, editors, *ESORICS*, volume 8134 of *Lecture Notes in Computer Science*, pages 718–735. Springer, 2013.

[SdBR$^+$11] Emin Gün Sirer, Willem de Bruijn, Patrick Reynolds, Alan Shieh, Kevin Walsh, Dan Williams, and Fred B. Schneider. Logical attestation: an authorization architecture for trustworthy computing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 249–264, New York, NY, USA, 2011. ACM.

[SG14] Raphael Spreitzer and Benoît Gérard. Towards more practical time-driven cache attacks. In David Naccache and Damien Sauveron, editors, *Information Security Theory and Practice. Securing the Internet of Things*, volume 8501 of *Lecture Notes in Computer Science*, pages 24–39. Springer Berlin Heidelberg, 2014.

[SK10] Udo Steinberg and Bernhard Kauer. Nova: A microhypervisor-based secure virtualization architecture. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 209–222, New York, NY, USA, 2010. ACM.

[SN05] Jim Smith and Ravi Nair. *Virtual Machines: Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and Design).* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[SP13] Raphael Spreitzer and Thomas Plos. On the applicability of time-driven cache attacks on mobile devices. In Javier Lopez, Xinyi Huang, and Ravi Sandhu, editors, *Network and System Security*, volume 7873 of *Lecture Notes in Computer Science*, pages 656–662. Springer Berlin Heidelberg, 2013.

[SPvD05] E. Shi, A. Perrig, and L. van Doorn. BIND: a fine-grained attestation service for secure distributed systems. In *Security and Privacy, 2005 IEEE Symposium on*, pages 154–168, 2005.

[SS04] Ahmad-Reza Sadeghi and Christian Stüble. Property-based attestation for computing platforms: caring about properties, not mechanisms. In *Proceedings of the 2004 workshop on New security paradigms*, NSPW '04, pages 67–77, New York, NY, USA, 2004. ACM.

[SVJ12] Joshua Schiffman, Hayawardh Vijayakumar, and Trent Jaeger. Verifying system integrity by proxy. In *Proceedings of the 5th international conference on Trust and Trustworthy Computing*, TRUST'12, pages 179–200, Berlin, Heidelberg, 2012. Springer-Verlag.

[SZJvD04] Reiner Sailer, Xiaolan Zhang, Trent Jaeger, and Leendert van Doorn. Design and implementation of a TCG-based integrity measurement architecture. In *Proceedings of the 13th conference on USENIX Security Symposium - Volume 13*, SSYM'04, pages 16–16, Berkeley, CA, USA, 2004. USENIX Association.

[The11] The OpenSSL Project. OpenSSL: The Open Source toolkit for SSL/TLS, February 2011. `http://www.openssl.org`.

[TOS10] Eran Tromer, DagArne Osvik, and Adi Shamir. Efficient cache attacks on aes, and countermeasures. *Journal of Cryptology*, 23(1):37–71, 2010.

[Tru11] Trusted Computing Group. TPM Main Specification Level 2 Version 1.2, Revision 116, March 2011.

[VCGRS13] Nicolas Veyrat-Charvillon, Benoît Gérard, Mathieu Renauld, and François-Xavier Standaert. An optimal key enumeration algorithm and its application to side-channel attacks. In LarsR. Knudsen and Huapeng Wu, editors, *Selected Areas in Cryptography*, volume 7707 of *Lecture Notes in Computer Science*, pages 390–406. Springer Berlin Heidelberg, 2013.

[VCGS13] Nicolas Veyrat-Charvillon, Benoît Gérard, and François-Xavier Standaert. Security evaluations beyond computing power. In Thomas Johansson and PhongQ. Nguyen, editors, *Advances in Cryptology – EUROCRYPT 2013*, volume 7881 of *Lecture Notes in Computer Science*, pages 126–141. Springer Berlin Heidelberg, 2013.

[VRS14] Venkatanathan Varadarajan, Thomas Ristenpart, and Michael Swift. Scheduler-based defenses against cross-vm side-channels. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 687–702, San Diego, CA, August 2014. USENIX Association.

[Wei14] Carsten Weinhold. Microkernel-based operating systems - introduction, Oct 2014. `http://os.inf.tu-dresden.de/Studium/KMB/WS2014/01-Introduction.pdf`.

[WHS12] Michael Weiß, Benedikt Heinz, and Frederic Stumpf. A cache timing attack on aes in virtualization environments. In Angelos D. Keromytis, editor, *Financial Cryptography and Data Security*, volume 7397 of *Lecture Notes in Computer Science*, pages 314–328. Springer Berlin Heidelberg, 2012.

[WHSE15] Sascha Wessel, Manuel Huber, Frederic Stumpf, and Claudia Eckert. Improving mobile device security with operating system-level virtualization. *Computers & Security*, pages –, 2015.

[WWAS15] Michael Weiß, Benjamin Weggenmann, Moritz August, and Georg Sigl. *Trusted Systems: 6th International Conference, INTRUST 2014, Beijing, China, December 16-17, 2014, Revised Selected Papers*, chapter On Cache Timing Attacks Considering Multi-core Aspects in Virtualized Embedded Systems, pages 151–167. Springer International Publishing, Cham, 2015.

[WWHW14] Michael Weiss, Steffen Wagner, Roland Hellmann, and Sascha Wessel. Integrity Verification and Secure Loading of Remote Binaries for Microkernel-based Runtime Environments. In *13th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*. IEEE Computer Society, 2014.

[ZW10] Xin-jie ZHAO and Tao WANG. Improved cache trace attack on aes and clefia by considering cache miss and s-box misalignment, 2010. zhaoxinjieem@163.com 14645 received 2 Feb 2010, last revised 5 Feb 2010.