Fakultät für Informatik
der Technischen Universität München

Lehrstuhl für Sicherheit in der Informatik

Leveraging Logical and Physical Separation for Mobile Security

Julian Maxim Horsch

Dissertation

Fakultät für Informatik
der Technischen Universität München

Lehrstuhl für Sicherheit in der Informatik

Leveraging Logical and Physical Separation for Mobile Security

Julian Maxim Horsch

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

Die Dissertation wurde am 02.04.2019 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 25.07.2019 angenommen.

Abstract

The importance of mobile computing and the number of mobile computing devices have both grown strongly over the recent years. Mobile devices, such as smartphones, laptops, tablets, and also wearables, show an interesting combination of traits. They are, as the name suggests, mobile, massively connected, gather valuable data with various sensors, and can typically be customized with apps. Together, those characteristics make mobile devices very popular among users and a prime attack target. More specifically, their mobility exposes mobile devices to a variety of, possibly hostile, environments, putting them at risk of loss and, thus, physical attacks. Their connectivity exposes them to numerous remote attacks via one of their various wireless interfaces. In reaction to this increasing threat potential, mobile devices and their CPUs gain more and more security functions. While some of those functions fulfill a very specific purpose, such as the secure boot, guaranteeing the integrity of the boot chain, others are very versatile and not fixed to a specific use case. This especially includes new functions for *logical separation*, such as Trusted Execution Environments (TEEs) and hardware-assisted virtualization, providing multiple isolated execution contexts inside the same CPU core. Logical separation enables novel, powerful security concepts, but can be overcome, for example, by cache timing attacks, raising the need for additional *physical separation* to protect high-value secrets.

In this work, we explore ways to leverage both logical and physical separation to improve a mobile device's confidentiality and integrity against remote and physical attackers. We define a system architecture consisting of the mobile *target platform* and a physically separated *security token*, connected to the target platform, storing high-value secrets. In order to enable the token to protect itself and its assets against a compromised target platform, we first propose a software-based trusted boot process for ARM application processors. Using a timing-based primitive for externally verifiable execution, this boot process allows the token to gain trust in the boot integrity of the target platform without relying on pre-shared or trusted key material, building the basis for all following token-based security concepts.

A remote attacker typically tries to exploit a vulnerability to take control over a process communicating via one of the target platform's interfaces. In order to detect and eventually prevent such remote attacks, we present a framework for transparent kernel and user space execution tracing from a minimal hypervisor. The framework leverages hardware-assisted virtualization in modern ARM CPUs to transparently restrict the target platform to a small set of executable memory pages, gathering control flow information on each resulting page fault in the hypervisor. The page-granular control flow data can be used as basis to detect and prevent control flow hijacking attacks and, thus, protect the target platform's run-time integrity.

Run-time integrity does not necessarily protect a system against physical attacks. A physical attacker might, for example, still be able to extract valuable data from the main memory of a system via DMA or cold boot attacks. In order to protect a target platform against such memory attacks, we present a concept for run-time kernel and user space main memory encryption from a minimal hypervisor. Like our execution tracing framework, our memory encryption scheme leverages hardware-assisted virtualization to transparently restrict the system to a small working set of recently accessed pages. By encrypting other pages, using a key stored in a TEE, our concept is able to effectively protect a running system against memory attacks. To provide stronger protection against attacks breaking logical separation, we furthermore propose the combination with a suspend-time memory encryption scheme, able to encrypt suspended processes with keys stored securely in the security token. Finally, to put our software-based encryption schemes into perspective, we present an attack on AMD Secure Encrypted Virtualization (SEV), a hardware-based memory encryption mechanism in recent AMD CPUs.

Recent years have shown that logical separation, even when provided by TEEs, can be broken by advanced attackers in various ways. Apart from exploiting software bugs, attackers can perform microarchitectural attacks, for example, using cache timing, to extract secrets, even from bug-free, logically separated contexts. In order to protect valuable symmetric keys, such as the target platform's Full Disk Encryption (FDE) key, against such attacks, we present a concept that leverages physical separation and partially moves the required cryptographic operations to our token, in a way fast enough for data-intensive applications. Our concept forces the target platform and token to cooperate during data encryption and decryption and binds encrypted data to the specific token used for its encryption. In order to protect user identities against advanced attackers, we propose a concept for deriving and using IDs in form of asymmetric key pairs, storing them securely on our physically separated token. Our concept is able to securely derive trusted IDs from a RootID device into the token, even when a completely compromised target platform brokers their communication with each other and the user.

We show the feasibility of all our concepts by presenting prototype implementations. Our performance and security evaluations confirm that our concepts are able to effectively and efficiently improve a mobile device's integrity and confidentiality against different remote and physical attackers.

Zusammenfassung

Die Wichtigkeit und Verbreitung mobiler Computernutzung hat in den vergangenen Jahren stark zugenommen. Die dabei verwendeten mobilen Geräte sind tragbar, besitzen eine große Anzahl zumeist drahtloser Kommunikationsschnittstellen und Sensoren und sind durch benutzerdefinierte Software flexibel einsetzbar. Diese Eigenschaften machen mobile Geräte höchst populär und zu einem bevorzugten Ziel für Angriffe. Aufgrund ihrer Mobilität sind die Geräte oft wechselnden Umgebungen und erhöhter Verlust- bzw. Diebstahlgefahr und damit physischen Angriffen ausgesetzt. Durch ihre hohe Konnektivität sind die Geräte zudem von einer Vielzahl an entfernten Angriffen bedroht. Als Reaktion auf diese Entwicklung werden die Geräte zunehmend mit neuen Sicherheitsfunktionen ausgestattet. Während einige dieser Funktionen einem ganz bestimmten Zweck dienen, wie zum Beispiel der sichere Bootprozess, der die Integrität von Boot-Komponenten sicherstellt, sind andere sehr flexibel einsetzbar. Hierzu zählen insbesondere neue Funktionen zur *logischen Separierung*, d.h. zur Unterteilung des CPU-Kerns in mehrere isolierte Ausführungsumgebungen. Logische Separierung, zum Beispiel durch Trusted Execution Environments (TEEs) oder Hardwarevirtualisierung, ermöglicht die Umsetzung neuartiger und mächtiger Sicherheitskonzepte, kann allerdings unter bestimmten Umständen von starken Angreifern umgangen werden. Zum Schutz von wertvollen Daten ist daher eine zusätzliche *physische Separierung* sinnvoll.

In dieser Arbeit erforschen wir die Nutzung logischer und physischer Separierung zum Schutz der Integrität und Vertraulichkeit mobiler Geräte gegen physische und entfernte Angreifer. Wir definieren zunächst eine Systemarchitektur bestehend aus einer mobilen *Zielplattform* und einem verbundenen, jedoch eigenständigen *Sicherheitstoken* für die Speicherung von Geheimnissen. Zum Schutz des Tokens vor einer kompromittierten Zielplattform stellen wir das Konzept eines softwarebasierten vertrauenswürdigen Bootprozesses für ARM Anwendungsprozessoren vor. Ohne zuvor geteiltes Schlüsselmaterial schafft der Bootprozess mittels eines zeitbasierten Verfahrens ein initiales Vertrauen des Tokens in die Integrität der auf der Zielplattform geladenen Komponenten und damit die Basis für weitere Token-basierte Konzepte.

Entfernte Angriffe nutzen typischerweise Sicherheitslücken in nach außen kommunizierenden Prozessen, um deren Kontrollfluss zu übernehmen. Um solche Angriffe zu erkennen und letztlich zu verhindern, stellen wir ein Framework zur transparenten Überwachung der Programm- und Kernelausführung aus einem minimalen Hypervisor vor. Das Framework nutzt hardwareunterstützte Virtualisierungsfunktionen, um die Zielplattform auf eine kleine Menge an ausführbaren Speicherseiten zu beschränken. Die Fehler, die beim Ausführen von Seiten außerhalb dieser Menge entstehen, werden vom Hypervisor abgefangen und zur Gewinnung von seitengranularen Kontrollflussinformationen genutzt. Diese können wiederum als Basis für die Verhinderung von Angriffen und daher dem Schutz der Laufzeitintegrität der Zielplattform dienen.

Ein laufzeitintegres System ist nicht notwendigerweise gegen physische Angriffe geschützt. Ein physischer Angreifer ist, zum Beispiel, möglicherweise weiterhin in der Lage mittels einer Kaltstart- oder DMA-Attacke Daten aus dem Hauptspeicher zu entwenden. Zum Schutz gegen solche Speicherangriffe stellen wir ein Konzept für Hauptspeicherverschlüsselung zur Laufzeit aus einem minimalen Hypervisor vor. Das Konzept nutzt, ebenso wie unser Framework zur Ausführungsüberwachung, Funktionen der hardwarebasierten Virtualisierung, um die Zielplattform transparent auf eine kleine Menge zuletzt zugegriffener Seiten zu beschränken. Durch die Verschlüsselung von Seiten außerhalb dieser Menge mit einem durch eine TEE geschützten Schlüssel, schützt unser Konzept laufende Systeme effektiv gegen Speicherangriffe. Zum Schutz vor Angriffen auf die dabei genutzte logische Separierung präsentieren wir darüber hinaus die Kombination mit einem unterbrechungsbasierten Konzept zur Speicherverschlüsselung, welches ergänzend in der Lage ist, temporär unterbrochene Prozesse mit Token-geschützten Schlüsseln zu verschlüsseln. Zur Einordnung unserer softwarebasierten Konzepte in einen größeren Kontext präsentieren wir zum Abschluss einen Angriff auf AMD Secure Encrypted Virtualization (SEV), einen hardwarebasierten Mechanismus zur Speicherverschlüsselung.

Die vergangenen Jahre haben eine Vielzahl von Angriffen hervorgebracht, die sogar von TEEs bereitgestellte logische Separierung erfolgreich umgehen. Neben Angriffen, die Softwarebugs ausnutzen, sind insbesondere mikroarchitekturelle Angriffe in der Lage, Geheimnisse aus separierten Kontexten zu entwenden ohne dabei auf Implementierungsfehler angewiesen zu sein. Zum Schutz wichtiger symmetrischer Schlüssel der Zielplattform vor ebendiesen Angriffen, zum Beispiel für die Festplattenverschlüsselung, präsentieren wir ein Konzept, welches die auszuführenden kryptographischen Operationen teilweise auf das physisch separierte Token verlagert, ohne dabei zuviel Geschwindigkeit einzubüßen. Unser Konzept zwingt die Zielplattform und das Token zur Zusammenarbeit während Ver- und Entschlüsselung und verknüpft die verschlüsselten Daten untrennbar mit dem spezifischen Token, das zur Verschlüsselung genutzt wurde. Zum Schutz von Nutzeridentitäten präsentieren wir ein Konzept zur Ableitung und Verwendung von IDs basierend auf asymmetrischen Schlüsseln. Unser Konzept ist in der Lage, vertrauenswürdige IDs von einem RootID-Gerät abzuleiten und diese sogar dann sicher im Token abzulegen, wenn die Zielplattform, welche die Verbindung zwischen beiden Geräten und dem Nutzer herstellt, kompromittiert ist.

Die praktische Umsetzbarkeit unserer Sicherheitskonzepte zeigen wir mit prototypischen Implementierungen. Unsere Geschwindigkeits- und Sicherheitsevaluationen bestätigen darüber hinaus, dass unsere Konzepte in der Lage sind, die Integrität und Vertraulichkeit mobiler Geräte effektiv und effizient zu verbessern.

Acknowledgements

First of all, I would like to thank Prof. Dr. Claudia Eckert for giving me the opportunity to pursue a doctoral degree in a field as fascinating and challenging as IT security. Her guidance and supervision, as well as her constant support and encouragement have been a great help in writing this thesis.

I would also like to thank Prof. Dr. Uwe Baumgarten for his interest in my work and for being the second examiner of this thesis.

Further, I would like to thank all current and former colleagues at Fraunhofer AISEC for inspiring discussions, trusting companionship, and support in major and minor questions. In particular, I would like to thank Dr. Michael Velten, Manuel Huber, Sascha Wessel, Dr. Steffen Wagner, Dr. Michael Weiß, Philipp Zieris, Dieter Schuster, Mathias Morbitzer, Konstantin Böttinger, and Dr. Frederic Stumpf.

Almost last but certainly not least, I would like to thank my family, extended family, and friends for their unconditional love and support and for always putting things into greater perspective.

Finally, I would like to thank Lisa, my partner and best friend, for her patience, wisdom, and strength, her constant encouragement, unwavering confidence in my abilities, and for, literally and figuratively, traveling with me on this journey.

Contents

List of Figures

List of Tables

List of Listings

Introduction 1

In recent years, mobile computing has become ubiquitous. Personal mobile computing devices, such as smartphones, laptops and tablets are heavily used[1] by an already vast and ever-increasing number of people[2]. Furthermore, the trend towards the Internet of Things (IoT) causes also mobile everyday objects to gain more and more computing and communication capabilities[3]. This results in a huge number of additional mobile devices, including small, personal ones, such as smartwatches and other wearables, but also larger, safety-critical devices like intelligent cars. A slow-down of the mobile computing growth is not to be expected[4].

While some mobile devices, such as laptops, resemble traditional PCs, most mobile devices are, by definition, *embedded devices* in the sense that they are highly integrated, combining computing components with peripherals and mechanical components to fulfill a specific function. Depending on the actual use case, embedded devices are typically driven by low-performance *microcontrollers* or high-performance *Systems on a Chip (SoCs)*, integrating multiple processor cores with other components, such as memory and peripheral devices, on a single die. Smartphones and tablets typically use high-performance ARM SoCs, whose performance is comparable to laptop processors by now. Despite those technical differences, a lot of mobile embedded devices are, in fact, capable of general-purpose computing, being fixed to a specific function only by their software. This means that, despite the strict definition of embedded devices, mobile devices are much more similar to each other and constitute a more homogeneous device class with regard to their complexity, versatility and attack surface than it might initially seem.

Mobile devices share a unique set of characteristics that makes them an interesting attack target and *mobile security* challenging. First of all, as the name suggests, they

[1]At the end of 2016, according to StatCounter [Sta16], the Internet usage from mobile devices exceeded desktop computers for the first time.

[2]As of 2016, 78 percent of the adult population in Germany use a smartphone [Bit17]

[3]The Bluetooth SIG predicts 4 billion devices with Bluetooth technology to ship in 2018 [Blu18].

[4]In 2019, Cisco predicted [Cis19] that by 2022, wireless and mobile devices will be responsible for 71 percent of the total IP traffic. According to Cisco, smartphones alone will account for 44 percent of the traffic.

are *mobile* and, hence, exposed to changing, possibly hostile environments and at risk of getting lost or stolen[1]. They are *massively connected*, almost exclusively using wireless communication technologies, such as mobile networks, Wi-Fi and Bluetooth, offering a variety of remote attack vectors. Typically being equipped with various sensors, mobile devices are *aware of their environment*, gathering very personal or valuable data. Especially smartphones nowadays provide a variety of sensors including cameras, GPS receivers, microphones, gyroscopes, accelerometers and even biometric sensors, such as fingerprint readers. Last but not least, mobile devices typically can be *customized with apps* installed by their users. All those properties make mobile devices extremely *versatile* and the primary, most personal computing devices for most of their users. Consequentially, mobile devices carry a lot of valuable data, such as payment information, biometrics, and cryptographic keys and identities. The recent years have shown that the combination of being extremely exposed and carrying valuable data indeed makes mobile devices, especially smartphones, prime targets for attackers[2].

Many attacks on mobile devices take advantage of one or more of their specific characteristics. In contrast to other device classes, such as desktops, mobile devices are at higher risk of getting lost or stolen, opening them up to *physical* attacks. An attacker gaining possession of a running device might, for example, try to extract data from its main memory using a Direct Memory Access (DMA) or cold boot attack. In a *DMA attack*, the attacker typically uses a peripheral port of the device with DMA capability, such as FireWire or Thunderbolt, for data extraction [Boi06; BDK05; SB12; Mar+19]. In a *cold boot attack*, the attacker physically moves the device's memory to another machine or reboots the device into a custom bootloader to extract data [Hal+09; Gut01]. While cold boot typically targets traditional PCs or laptops, it has been shown to be effective against smartphones as well [MS13; Tau+15].

Their mobility and various wireless communication interfaces, furthermore, open mobile devices up for *proximity attacks*, exploiting a vulnerability in the hardware, implementation or protocol of a wireless interface from a relatively close distance. *Key Reinstallation Attacks (KRACKs)* [VP17] target the Wi-Fi Protected Access 2 (WPA2) 4-way handshake used to join a Wi-Fi network. In a KRACK attack, the attacker takes a Man-in-the-Middle (MitM) position and re-installs a key that is already in-use, resetting nonces and counters. With a following known-plaintext attack, the attacker can then decrypt packages. While this attack does not target the device directly and most importantly does not allow for Remote Code Execution (RCE), it still illustrates the specific challenges of mobile security. *Broadpwn* [Art17] is another recent proximity attack, which exploits

[1]In 2014, the FCC reported more than a million stolen smartphones per year in the US [Fed14]. In the UK, almost half a million mobile phone thefts were reported in the year ending march 2016 [Off16]. In Germany, in 2013 almost a quarter million mobile phones were reported stolen [Bun14].

[2]In 2017, with over 800 by far the most Common Vulnerabilities and Exposuress (CVEs) of any software product were reported for Android [CVE]. With almost 400 CVEs iOS, the other important mobile Operating System (OS), occupies the third place.

Broadcom Wi-Fi chipsets, widely used in smartphones. By exploiting a flaw in the implementation of the Wi-Fi association protocol, Broadpwn achieves RCE on the Wi-Fi chip itself without user interaction. From there, it can attack the device, for example, using the Wi-Fi chip's DMA access to extract main memory data remotely. *Blueborne* [SV17] is a collection of vulnerabilities and exploits targeting Bluetooth stacks on desktops and mobile devices. From within Bluetooth range, at the time of its discovery, Blueborne could exploit vulnerabilities in the complex Bluetooth software stack to achieve RCE on Android and iOS, among others. While all three presented vulnerabilities are fixed in current versions of the affected devices and software, they illustrate how large and diverse the attack surface of mobile devices actually is.

The versatility and complexity of mobile device software also results in an increased number of *remote exploits*. Compared to proximity and physical attacks, remote exploits are typically rather traditional but can have a huge impact considering today's prevalence of mobile devices and the value of the data at risk. *Pegasus* [Baz+16], discovered in 2016, is a highly developed iOS malware, able to spy on the user of an infected device. The malware distribution is initiated as a *phishing* attack. As soon as the user visits a specially crafted website, the attack exploits the browser and achieves RCE. Two further vulnerabilities are then used for privilege escalation, persisting the malware and enabling it to gather sensitive data. *Stagefright* [ADB15] is a well-known group of vulnerabilities and corresponding exploits in the Android multimedia framework. At the time of its discovery, Stagefright allowed an attacker to achieve RCE on a victim's device using numerous ways, including Messengers, Email and Bluetooth. The Pegasus and Stagefright exploits are patched in current versions of iOS and Android, but they are being followed by newer, more complex attacks, using advanced techniques such as Return-Oriented Programming (ROP) [Sha07; Kor10], emphasizing the need for advanced mobile security.

On the other hand, the security of mobile devices improves constantly. While some improvements are completely based in software, such as sandboxing and the use of safe programming languages, software security features are often directly or indirectly driven by new hardware features. The introduction of virtual memory and relative addressing enabled Position-Independent Code (PIC) and, thus, *Address Space Layout Randomization (ASLR)*, a widely used defensive mechanism that impedes ROP and other Code-Reuse Attacks (CRAs)[1]. Access permissions on virtual memory pages enabled *Data Execution Prevention (DEP)* and *Write XOR Execute (W^X)*, powerful basic techniques, preventing an attacker from injecting new or modifying existing code. While those techniques were originally introduced on traditional computers first, they are, by now, standard on high-performance mobile device, as well. Other hardware-driven security features were first introduced on mobile devices, such as the *secure boot*, enabling systems that exclusively boot and run signed software. Another mobile-driven technique is the usage of device-unique,

[1]While ASLR, for example, helps to prevent Stagefright, as a probabilistic method it is often considered an insufficient defense against advanced attackers [Sha+04].

secret hardware keys that cannot be extracted by software. Those allow, for example, for the encryption of secondary storage without an attacker being able to brute-force the encryption off-device. These are only a few examples of hardware features driving the development of new security mechanisms.

In a relatively recent development, mobile Central Processing Units (CPUs) gain more *logical separations*, i.e., separated code execution contexts typically associated with a privilege level of the software running in them. Traditionally, a CPU at least separates user and kernel space. Trusted Execution Environments (TEEs), such as ARM TrustZone [AF04] and Intel Software Guard Extensions (SGX) [McK+13], additionally partition a system into a *secure world* and a *normal world*. Hardware-assisted virtualization extensions introduce new CPU modes and features to enable efficient platform virtualization. While the latter has traditionally been a feature of server CPUs, it has, by now, found its way into high-performance mobile ARM CPUs [MN11]. Besides their intended uses, for example, for platform virtualization, those new logical separations offer interesting possibilities for novel mobile security mechanisms. In this thesis, we want to leverage those possibilities to protect a mobile device against different remote and physical attackers.

While logical separation is invaluable for protecting the integrity and confidentiality of most devices, in some cases it is not sufficient to protect high value secrets. First of all, bugs in privileged components are frequent [Ros14; Ben17] and typically expose secrets protected by the separation. But also without exploiting software bugs, logical separation can be overcome by advanced attackers. First, a separation may be crossed by physical attackers, for example, by using cold boot or DMA attacks as discussed before. But more importantly, a whole class of *microarchitectural attacks* can exploit the fact that logically separated contexts still share most hardware components of the system, such as cache, branch predictor and Dynamic Random Access Memory (DRAM), to leak secrets. This way, basic cache timing side channels [Ber05; YF14] can extract keys from separated contexts. Furthermore, recently discovered attacks *Meltdown* [Koc+18; Hor18] and *Spectre* [Lip+18b; Hor18] leverage speculative execution on modern, high-performance CPUs for cache timing attacks that can read arbitrary data from privileged contexts, including TEEs [Che+18]. Finally, *Rowhammer* attacks break logical separation in main memory by leveraging a physical property of DRAM, which, when accessing a specific DRAM address in rapid succession, can show bit flips in nearby memory cells [Kim+14]. Rowhammer attacks have been shown to be effective on mobile devices [Vee+16], can be used for privilege escalations [Sea15], remotely [Lip+18a; Tat+18] and can be able to cross the strong logical isolation by TEEs [Car17]. Those attacks illustrate that, while sufficient for most data, some high value secrets, such as keys and user identities, require stronger protection than provided by logical separations in modern CPUs. Hence, as another part of this thesis, we want to explore new ways to leverage physical separation to protect valuable secrets.

1.1 Research Questions

It should be clear by now that mobile devices are exposed to a large variety of attacks making mobile security challenging. Often, the ultimate goal of an attacker is to extract valuable user data, keys or identities. Based on the discussion in the previous section, in this thesis, we want to explore new ways to use *features of modern CPUs*, especially *logical and physical separations*, to improve the security of mobile devices against physical and remote attackers. More specifically, we want to answer the following research questions:

How can we monitor and protect run-time integrity? A remote attacker typically gains access to a mobile device by exploiting a process via one of the remote communication interfaces. Controlling a user space process, the attacker typically tries to elevate his privileges to gain access to the kernel space and, ultimately, extract secrets and data from the device. Monitoring the execution of the system is the first step to protection. Monitoring execution from a privileged, separated context could result in a protection mechanism that works transparently and independently of the device's OS. Hence, we want to explore ways to use the new logical separations in modern CPUs, more specifically hardware-assisted virtualization, to monitor and, ultimately, protect the run-time execution of a device.

How can we protect data in main memory? Without relying on software bugs, strong physical attackers might be able to extract valuable user data from main memory of a mobile device, for example, via a cold boot or DMA attack. While Full Disk Encryption (FDE) is widely used on mobile devices, Random Access Memory (RAM) is typically unencrypted. Hence, we want to explore ways to protect the confidentiality of main memory, at run-time *and* while the device is suspended. Again, we want to leverage logical and physical separation to develop new approaches and to improve the security of existing ones.

How can we protect keys and identities? While logical, in-CPU separation is essential for most security concepts, and new logical separations allow for novel approaches, in some cases, the achieved security is not sufficient. Software bugs in privileged software, physical attacks and microarchitectural attacks might be able to break through logical separation, emphasizing the need for stronger protections. Especially for small, high value secrets, such as cryptographic keys and identities, better protections seem feasible and necessary. Hence, we want to explore ways to leverage physical separation to protect keys for symmetric cryptography *and* asymmetric keys used as user identities.

Securing mobile devices poses some unique challenges. By answering our research questions, we want to approach some of these challenges and improve overall mobile security, especially the confidentiality of valuable user data and secrets.

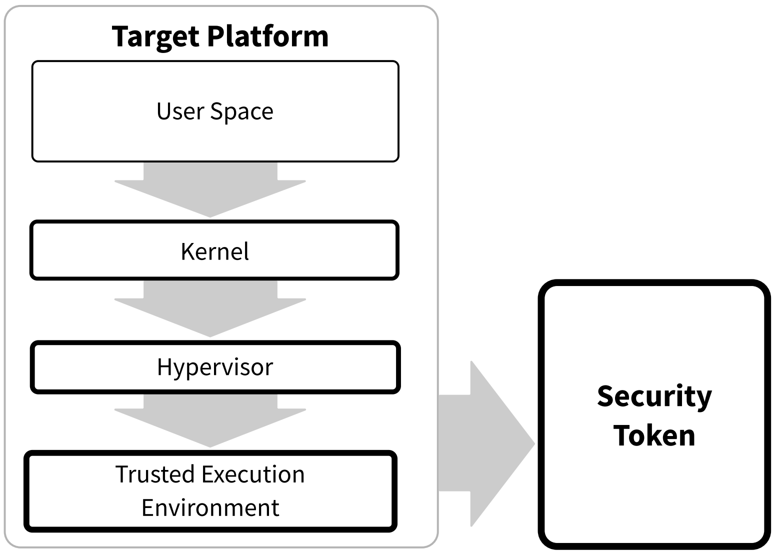


Figure 1.1: Logical and physical separation of security-critical components.

1.2 Contributions

In order to find answers to our research questions, we want to follow a basic scheme for our security concepts. Based on the assumptions that complex software always contains vulnerabilities proportional to the code size, we want to move security-critical code and assets into separated contexts, isolating them from the majority of software running on a mobile device and reducing the Trusted Computing Base (TCB) for our concepts.

Our goal is to protect a high-performance mobile device, our *target platform*, against different physical and remote attackers, as described before. To achieve this, we want to use all logical, in-CPU separations available and develop novel concepts that leverage their specific features. The separations include the kernel layer, the virtualization layer and the secure world of a TEE. As explained before, for some valuable assets, the isolation provided by a logical separation might be insufficient. Hence, we extend our architecture with a dedicated, physical security device, the *security token*, providing an additional, *physical* separation for some of our concepts. Our basic vision is summarized in Figure 1.1, showing the logical and physical separations in our two-part system architecture.

The main contributions in this thesis towards reaching this vision are summarized in the following:

Page-granular transparent tracing of code execution. We present a framework for logically separated kernel and user space execution tracing as a basis for protecting a target platform against remote attackers. Our framework uses hardware-assisted virtualization features to restrict the number of executable pages in a guest from a custom minimal hypervisor. The framework analyzes the resulting page faults to infer information about the guest's control flow. Based on this, we furthermore propose a concept using the framework to transparently enforce a particular page-granular control flow in the guest. We present a prototype implementation on an ARM Cortex-A15 development board running Android OS. Based on our prototype, we conduct a detailed performance evaluation of our tracing framework and hardware-assisted virtualization on ARM in general.

Publication: [HW15]

Run-time main memory encryption. We present a run-time memory encryption scheme protecting data confidentiality of processes and kernel running on our target platform against physical attackers. We leverage hardware-assisted virtualization and use a minimal hypervisor to restrict a guest, including kernel and user space, to a small working set of recently accessed pages, while keeping other pages encrypted. The mechanism is transparent and works independently of the guest OS. We, furthermore, propose a mechanism to transparently detect and exclude pages for which the encryption would lead to malfunction, for example, because they are shared with DMA devices. Based on our fully functional prototype implementation on an ARM Cortex-A15 development board running Android OS, we present a detailed evaluation of our run-time memory encryption.

Publication: [HHW17a]

Combination of run- and suspend-time memory encryption. Typically, not all functions of a mobile device are required at all times and many components can be safely suspended until needed. In order to provide even stronger protection for the data of suspended components, we describe the combination of our run-time memory encryption approach with a suspend-time memory encryption scheme, which is the result of joint work research. We describe how the suspend-time encryption scheme is able to protect data of suspended processes or process groups on our target platform against strong physical attackers using our token as physically separated key storage. The mechanism is orthogonal to the run-time encryption and leverages Linux functions for process grouping and suspension.

Our approach competes with recent hardware-assisted memory encryption schemes. Hence, additionally, as the result of joint work, we provide a security analysis of AMD's virtual machine encryption. The scheme aims to protect the data confidentiality of virtual machines against a malicious hypervisor. We describe and realize an attack able to extract memory contents from a virtual machine encrypted with this mechanism using a malicious hypervisor. Our successful attack shows that concepts that try to remove privileged components from the TCB are hard to realize.

Publication: [HHW17a]
Joint work publications: [HHW17b; Hub+17; Hub+18; Mor+18]

Externally verifiable execution on ARM. To protect valuable assets of our target platform against attacks breaking logical separation, we introduce an additional, physical separation in form of our security token. In order to protect the assets on the token against a compromised target platform, we want the token to be able to verify the integrity of the target platform's boot process before unlocking its security functions. To achieve this without requiring special attestation hardware and key material in the target platform, we introduce a timing-based software primitive that enables the token to verify that a specific piece of code is executed unmodified on our target platform. This primitive must be specifically designed for the target platform's CPU, i.e., in our case an ARM Cortex-A mobile processor. Based on this primitive, we introduce a *software-based trusted boot* process which proves the target platform's boot time integrity towards the token. We present a full implementation of the primitive and the boot process with an ARM Cortex-A8 development board as target platform. Based on our prototype, we provide a detailed experimental evaluation of the concept, showing that malicious modifications of the boot-integrity can be detected reliably.

Publication: [Hor+14b]

Fast token-based external symmetric cryptography. In order to protect valuable symmetric keys of the target platform, for example, for FDE, against attacks breaking logical separation, we want to leverage the physically separated token. Moving the keys from the target platform to the token protects them but executing all related cryptographic operations externally is too slow for data-intensive use cases, such as FDE. To solve this problem, we propose a concept that *partially* moves those cryptographic operations to the token, ensuring the security of the externally stored keys while still offering comparably high performance. Our concept combines keys of both entities and forces the target platform and token to cooperate during data encryption and decryption. The key combination, furthermore, binds the encrypted data to the specific token used for its encryption. We present a complete prototype implementation and a detailed performance evaluation showing that our prototype is fast enough for data-intensive use cases.

Publication: [HWE16]

Token-based identity derivation and usage. Mobile devices are often very personal devices and, hence, typically store a lot of user authentication credentials. As an alternative to, for example, passwords, the authentication can be secured using asymmetric cryptography Identities (IDs), i.e., a key pair and a certificate trusted by the targeted service. Those IDs can be protected against attacks on logical separation by storing them on our token. Asymmetric cryptography used for authentication happens much less frequently and, therefore, in contrast to the symmetric case, the operation can be fully moved to the token without performance issues. But the challenge remains how to securely generate IDs on the token that are immediately trusted by a service the users wants to authenticate to. To solve this problem, we extend our architecture with a trusted *RootID* device and propose a protocol able to securely *derive* trusted IDs from the RootID into the token, even when assuming a completely compromised target platform. We, furthermore, present a protocol for securely *using* derived IDs for authentication. Our prototype implementation confirms the feasibility of our approach.

Publication: [Hor+14a]

1.3 Publications

Parts of the contributions in this thesis have been published in the following scientific, peer-reviewed articles. Some of the publications contain additional contributions not covered in this thesis.

[Hor+14a] 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. San Antonio, Texas, USA: ACM, Mar. 2014, pp. 281–288.

[HHW17a] Julian Horsch, Manuel Huber, and Sascha Wessel. “TransCrypt: Transparent Main Memory Encryption Using a Minimal ARM Hypervisor”. In: *Proceedings of the 16th International Conference on Trust, Security and Privacy in Computing and Communications*. TrustCom ’17. Sydney, Australia: IEEE, Aug. 2017, pp. 152–161.

[HW15] Julian Horsch and Sascha Wessel. “Transparent Page-Based Kernel and User Space Execution Tracing from a Custom Minimal ARM Hypervisor”. In: *Proceedings of the 14th International Conference on Trust, Security and Privacy in Computing and Communications*. TrustCom ’15. Helsinki, Finland: IEEE, Aug. 2015, pp. 408–417.

[HWE16] Julian Horsch, Sascha Wessel, and Claudia Eckert. “CoKey: Fast Token-based Cooperative Cryptography”. In: *Proceedings of the 32nd Annual Conference on Computer Security Applications*. ACSAC ’16. Los Angeles, California, USA: ACM, Dec. 2016, pp. 314–323.

[Hor+14b] Julian Horsch, Sascha Wessel, Frederic Stumpf, and Claudia Eckert. “SobTrA: A Software-based Trust Anchor for ARM Cortex Application Processors”. In: *Proceedings of the 4th ACM Conference on Data and Application Security and Privacy*. CODASPY ’14. San Antonio, Texas, USA: ACM, Mar. 2014, pp. 273–280.

[Hub+17] Manuel Huber, Julian Horsch, Junaid Ali, and Sascha Wessel. “Freeze & Crypt: Linux Kernel Support for Main Memory Encryption”. In: *14th International Conference on Security and Cryptography (SECRYPT 2017)*. INSTICC. ScitePress, 2017.

[Hub+18] Manuel Huber, Julian Horsch, Junaid Ali, and Sascha Wessel. “Freeze and Crypt: Linux Kernel Support for Main Memory Encryption”. In: *Computers & Security* (2018).

[HHW17b] Manuel Huber, Julian Horsch, and Sascha Wessel. “Protecting Suspended Devices from Memory Attacks”. In: *Proceedings of the 10th European Workshop on Systems Security*. EuroSec’17. Belgrade, Serbia: ACM, Apr. 2017, 10:1–10:6.

[Mor+18] Mathias Morbitzer, Manuel Huber, Julian Horsch, and Sascha Wessel. “SEVered: Subverting AMD’s Virtual Machine Encryption”. In: *Proceedings of the 11th European Workshop on Systems Security*. EuroSec’18. Porto, Portugal: ACM, Apr. 2018.

Other publications. Additionally, we published articles whose contributions are not part of this thesis. In [Hub+15], we propose a secure architecture for OS-level virtualization for mobile devices. Our approach allows multiple Linux-based OS instances, such as Android, to run simultaneously on a single physical device. We isolate those instances from each other by restricting them to their minimally required functionality and permitting communication only through well-defined channels. Our architecture is able to keep user data secure even if large parts of the rest of the system are compromised.

In [ZH18], we propose a novel compiler-based approach to protect programs written in unsafe languages, such as C and C++, against attacks corrupting return addresses. By storing return addresses on a second, *safe stack*, we effectively prevent various attacks, ranging from simple code injections to advanced techniques like ROP. In contrast to other *dual stack* schemes, our safe stack is protected against information disclosure attacks. With an average overhead of no more than 2.7%, our approach is well usable in real-world scenarios.

In [MHH19], we propose a concept for locating memory pages containing valuable secrets, such as keys, in encrypted virtual machines, only by observing page faults and certain I/O events. In our prototype implementation, we combine this knowledge with our method for extracting data from encrypted memory presented in this thesis, to efficiently extract valuable keys from an encrypted virtual machine.

[Hub+15] Manuel Huber, Julian Horsch, Michael Velten, Michael Weiß, and Sascha Wessel. “A Secure Architecture for Operating System-Level Virtualization on Mobile Devices”. In: *Information Security and Cryptology - 11th International Conference, Inscrypt 2015, Beijing, China, November 1-3, 2015, Revised Selected Papers*. Vol. 9589. Lecture Notes in Computer Science. Springer, 2015, pp. 430–450.

[ZH18] Philipp Zieris and Julian Horsch. “A Leak-Resilient Dual Stack Scheme for Backward-Edge Control-Flow Integrity”. In: *Proceedings of the 2018 ACM Asia Conference on Computer and Communications Security*. ASIA CCS ’18. Incheon, Republic of Korea: ACM, June 2018.

[MHH19] Mathias Morbitzer, Manuel Huber, and Julian Horsch. “Extracting Secrets from Encrypted Virtual Machines”. In: *Proceedings of the 9th ACM Conference on Data and Application Security and Privacy*. CODASPY ’19. Richardson, Texas, USA: ACM, Mar. 2019.

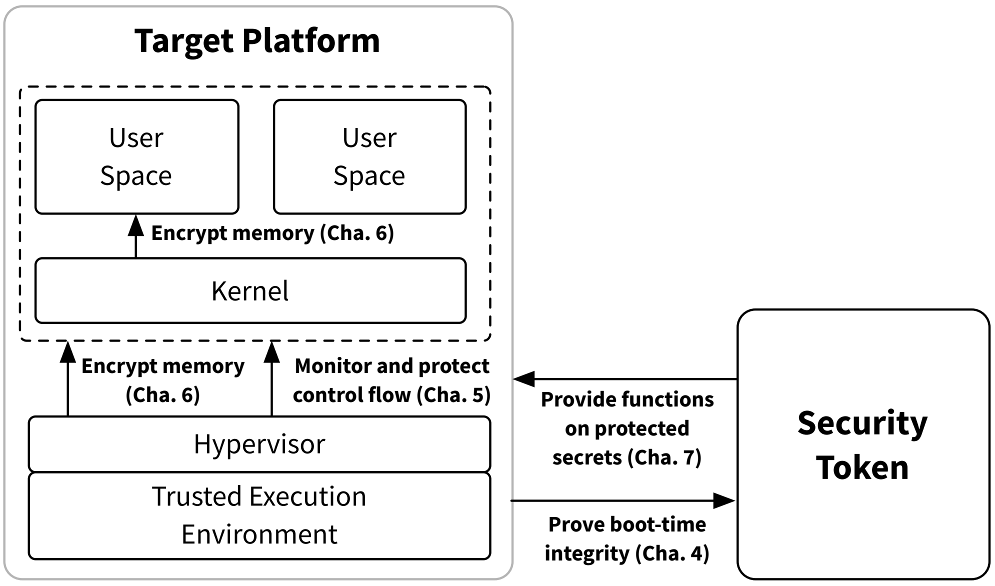

Target Platform
User Space
User Space
Encrypt memory (Cha. 6)
Kernel
Encrypt memory (Cha. 6)
Monitor and protect control flow (Cha. 5)
Hypervisor
Trusted Execution Environment
Provide functions on protected secrets (Cha. 7)
Prove boot-time integrity (Cha. 4)
Security Token

Figure 1.2: Visual outline of the thesis.

1.4 Outline

The remainder of the thesis is organized as follows. In Chapter 2, we introduce existing technologies and concepts that are necessary or helpful as background for understanding our contributions. First, we discuss fundamentals of the ARM architecture, the predominant CPU architecture for mobile devices. Then, we cover virtualization concepts, especially platform virtualization as logical separation in modern CPUs. Afterwards, we introduce TEEs as hardened logical CPU separation and boot integrity mechanisms, i.e., secure and measured boot.

In Chapter 3, we first introduce our generic system architecture used throughout the thesis. We discuss both physical entities of the architecture and the hardware requirements imposed on them by our different security concepts. We conclude the chapter introducing a generic attacker model building the basis for the specific attacker models in the following chapters.

Chapters 4, 5, 6 and 7 build the main contributions of the thesis. Figure 1.2 visualizes the main topics of those chapters in the context of our system architecture. In Chapter 4, we introduce our software-based trusted boot process which allows the target platform to prove its boot-time integrity towards the token without requiring shared key material. The mechanism is based on SobTrA, our primitive for externally verifiable execution on ARM, which we present as main part of the chapter. After introducing a specific attacker model and discussing related work, we present the SobTrA design, our implementation on Cortex-A8 CPUs and experimental results from our prototype. The proof of the target platform's boot time integrity guarantees the token that the following security concepts on the target platform are active at run-time.

In Chapter 5, we present our transparent, virtualization-based framework for page-granular tracing execution of kernel and user space on our target platform. As framework application with the goal to protect the run-time integrity of our target platform, we propose a concept for enforcing a particular page-granular control flow. After introducing typical attacks and related work, we present conceptual and implementation details of our tracing framework. Then, we introduce our run-time integrity application and its implementation. Finally, we conclude the chapter with a detailed performance analysis based on our prototype implementation.

In Chapter 6, we introduce our concepts to protect the confidentiality of the target platform's main memory. We first discuss memory attacks, able to extract data from main memory, and, based on them, specify an attacker model. Then, we introduce TransCrypt, our virtualization-based approach for encrypting the main memory of kernel and user space at run-time. We describe conceptual details of TransCrypt and its prototype implementation and, afterwards, evaluate the prototype regarding security and performance. Then, we introduce our Linux-based suspend-time encryption scheme and discuss the combination of both our run-time and suspend-time approaches. Finally, we conclude the chapter with a security analysis of AMD SEV, a hardware-based mechanism for memory encryption, presenting SEVered, an attack with which we are able to subvert SEV.

In Chapter 7, we propose two concepts that externalize highly valuable secrets from the target platform to the token to protect them against attacks breaking logical separations in the target platform's CPU. After detailing those attacks, we first introduce CoKey, our approach for protecting symmetric keys by forcing the target platform and token to cooperate during encryption and decryption. Then, we present TrustID, our concept for securing user identities with the token. For both approaches, we cover conceptual details and the respective prototype implementations before discussing the security gained through each concept.

We conclude the thesis in Chapter 8, where we summarize and assess our contributions before sketching possible future research topics in the field of mobile security.

Background 2

In this chapter, we introduce basic technologies and concepts important for understanding the rest of the thesis. While some, more specific aspects of the topics are introduced later where needed, the information given in this chapter is typically useful for more than one chapter of the thesis.

2.1 ARM Architecture

The ARM architecture is a widely used Reduced Instruction Set Computing (RISC) design for CPUs. ARM processors provide some characteristic features typical for RISC architectures:

- Many general-purpose registers, which can, in most cases, be freely used in three-operand instructions.
- Memory contents are only read and written using explicit load and store instructions (*Load-Store* architecture).
- Simple addressing modes for load/store where addresses are only generated based on register contents and immediates.

The ARM architecture is versioned, with ARMv8 being the latest version at the time of writing. Furthermore, ARM differentiates between three architecture *profiles*:

A – Application. Profile for traditional application processors. Provides support for virtual memory with an Memory Management Unit (MMU).

R – Real-time. Profile for processors in safety-critical environments. Provides support for memory protection with a Memory Protection Unit (MPU).

M – Microcontroller. Profile for microcontrollers, optimized for low-latency interrupt handling. Supports a variant of the R-profile MPU.

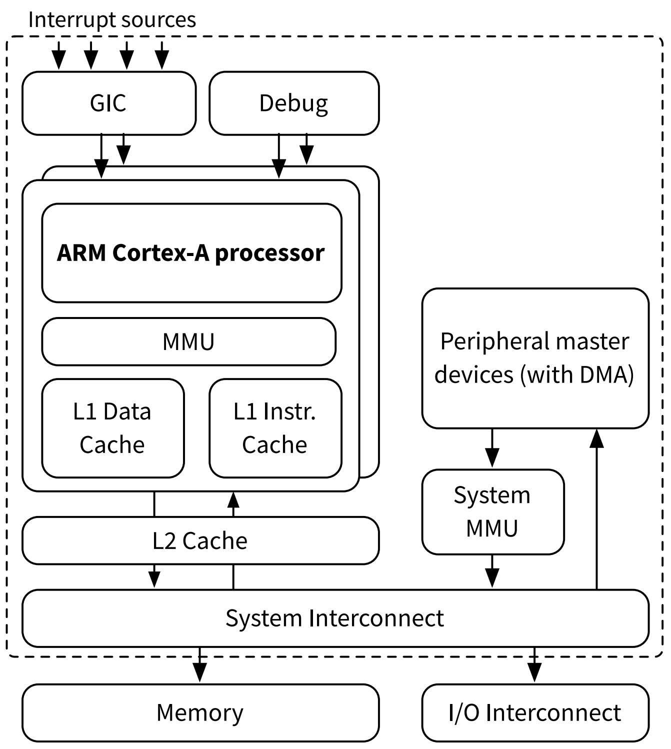


Figure 2.1: Architecture of a typical ARM SoC.

In this work, we focus on the two latest versions of the A-profile architecture, namely ARMv8-A [ARM17] and ARMv7-A [ARM14]. ARM, furthermore, specifies extensions to the architecture, for example, the ARMv7-A Virtualization Extensions [ARM14] and ARMv8 Cryptographic Extensions. For ARMv8, ARM also specifies updates to the architecture in form of sub-versions, e.g., ARMv8.3-A [ARM17]. The following descriptions refer to ARMv8-A without extensions and are based on information from the ARMv8-A reference manual [ARM17], if not stated otherwise.

Aside from specifying the ARM architecture, ARM develops designs for actual CPUs implementing the architecture. Those *Cortex* processors typically follow a simple naming scheme including the architecture profile and a number. For example, the Cortex-A53 processor implements the ARMv8-A architecture. An ARM Cortex-A CPU is typically part of an embedded SoC, containing not only multiple CPU cores but also bus systems, co-processors, caches, peripherals, Graphics Processing Units (GPUs) and other components in a single die. Figure 2.1 depicts a typical, yet simplified SoC architecture. In the following, we discuss the basic architectural concepts of an ARM application processor core.

2.1.1 Execution States and Instruction Sets

ARMv8-A [ARM17] defines two *execution states* for a CPU: AArch64 and AArch32. While AArch32 is a 32-bit state backward-compatible to ARMv7-A, AArch64 is a complete 64-bit redesign of the architecture. Since AArch32, apart from some additions for interprocessing with AArch64, basically equals ARMv7-A, we use both terms interchangeably. The execution state determines several important properties of the CPU's execution environment:

- The supported Instruction Set Architectures (ISAs) and register set.
- Significant aspects of the exception, programmers' and memory model.

The ARMv8-A execution states support different *instruction sets*, i.e., ISAs. While the backward-compatible AArch32 state supports *A32* and *T32*, which are basically the ARM and Thumb-2 ISAs specified by ARMv7, AArch64 only supports the new *A64* ISA.

A32. The A32 instruction set uses fixed-length 32-bit wide instructions and 32-bit registers. It is compatible to the ARMv7 ARM instruction set.

T32. The T32 instruction set uses the same 32-bit registers as the A32 ISA but a mixture of 32- and 16-bit wide instructions to increase code density. T32 is compatible to the ARMv7 Thumb-2 instruction set.

A64. The A64 instruction set uses fixed-length 32-bit wide instructions and, primarily, 64-bit registers.

The AArch32 state and its ISAs use a register set that contains thirteen 32-bit general-purpose registers and some special registers, such as *Program Counter (PC)*, *Stack Pointer (SP)* and *Link Register (LR)*. The AArch64 state uses 31 64-bit general-purpose registers, a PC, and several SPs and LRs. In AArch64, components of the program state (`PSTATE`), such as the arithmetic flags, are directly and independently modified using special instructions. AArch32 additionally supports a backward-compatible mechanism for modifying the entire program state by accessing the *Current Program Status Register (CPSR)*. Furthermore, while in ARMv7 and AArch32 system registers, such as the *System Control Register (SCTLR)*, are accessed with co-processor instructions, AArch64 allows named access to those.

The exception and memory models of the ARMv8-A execution states are discussed in the following.

2.1.2 Privilege and Exception Model

ARM application processors implement multiple privilege levels for logical separation of different types of software running on the CPU. Figure 2.2 shows the logical separations inside an ARMv8-A processor. ARMv8-A defines four Exception Levels (ELs), EL0-3. The EL determines the privilege of the software running in it, i.e., its Privilege Level (PL).

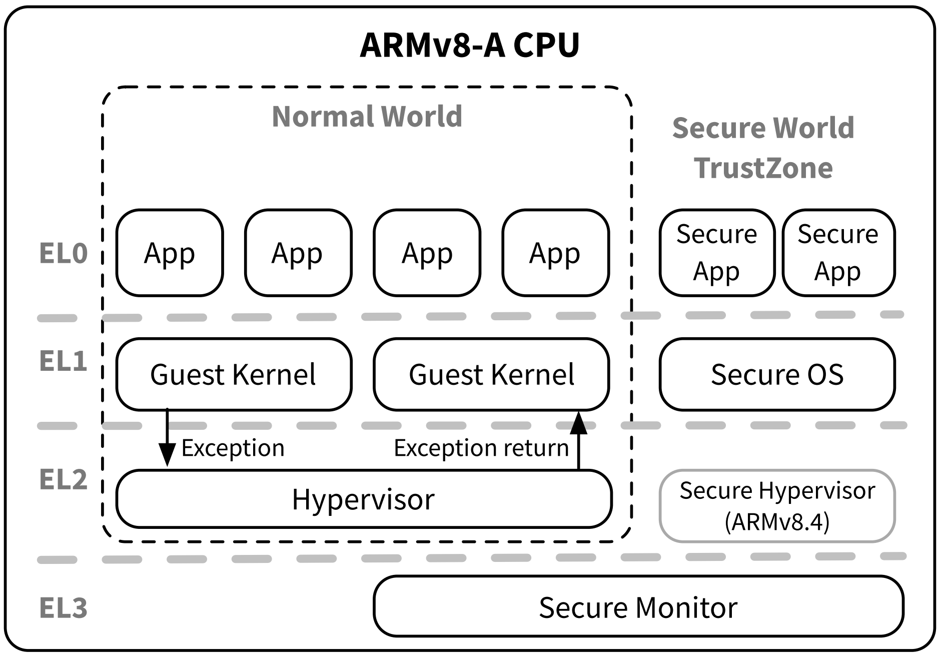


Figure 2.2: Exception and privilege levels of an ARMv8-A CPU.

The higher the EL, the more privileged, with EL3 being the highest privilege and EL0 the lowest, also known as *unprivileged* execution. EL0-2 are mirrored in normal and secure state versions. As shown in Figure 2.2, applications typically run in EL0, OS kernels in EL1 and hypervisors or Virtual Machine Monitors (VMMs) in EL2. EL3 typically houses firmware and code for switching between normal and secure world, building ARM's TEE TrustZone, discussed later in this chapter. Both, EL2 and EL3, are architecturally optional and may be omitted in an ARMv8-A processor.

Transfers. Transfers between ELs happen via exceptions. When an exception occurs, the execution is transferred from, i.e., *taken from*, the current EL *to* the same or a higher EL to a code location previously specified in the *exception vector table* of the target EL. After handling the exception, the execution is usually returned to the location and EL it was taken from. In a multi-core system, each core executes and transfers between the ELs independently. ARM differentiates between synchronous and asynchronous exceptions. A *synchronous exception* is a direct result of an (attempted) execution of an instruction. Such an exception is triggered, for example, when an unprivileged application running in EL0 tries to execute a privileged instruction or issues a system call using `SVC`. *Asynchronous exceptions* or *interrupts* are not the direct result of an instruction but other, often external, events, such as peripheral devices signaling incoming data.

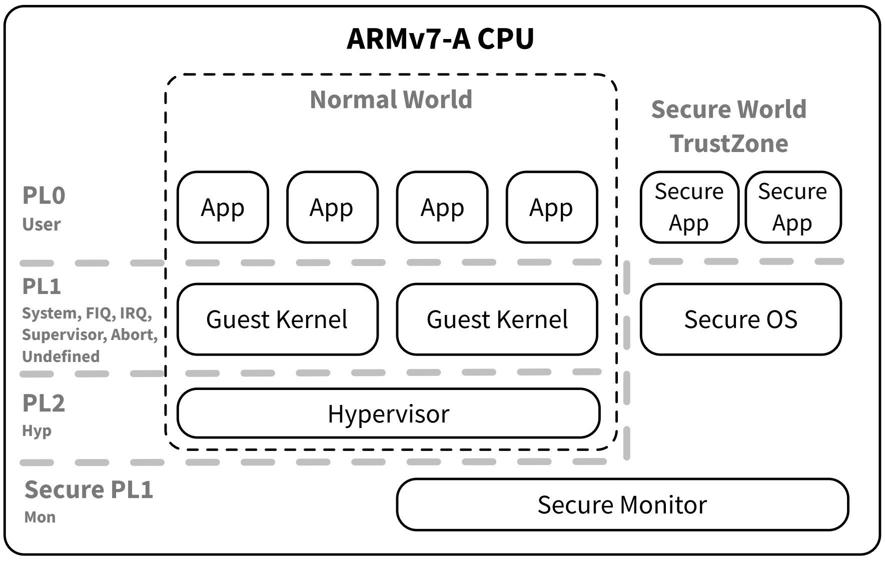


Figure 2.3: Exception and privilege levels of an ARMv7-A CPU.

Interprocessing. ELs *use* execution states, i.e., AArch32 or AArch64. An ARMv8-A CPU can run ELs with different execution states and change them when changing the EL, which is called *interprocessing*. Interprocessing may only change state from AArch32 to AArch64 when taking an exception to a higher EL and from AArch64 to AArch32 when returning from an exception into a lower EL. For example, an OS kernel running in AArch64 might be able to support AArch32 applications, but an AArch32 kernel only runs AArch32 applications. This also means that running AArch32 EL3 software forces all ELs to use AArch32 resulting in a CPU that is completely compatible to an ARMv7-A CPU.

ARMv7-A exception model. Figure 2.3 shows the exception and privilege model of an ARMv7-A CPU or an ARMv8-A CPU running AArch32 EL3 software. Here, the CPU supports different *modes*, which are mapped to privilege levels PL0-2. While the other PLs each only contain one mode, PL1 provides a mode for each type of exception it handles. The ARMv7-A model does *not* contain an extra privilege level for the secure monitor. Instead, the corresponding Monitor mode shares the Secure PL1 domain with the Secure OS, both handling different exceptions using distinct exception vector tables.

2.1.3 Memory Model

ARM application processors use virtual memory. All instructions use Virtual Addresses (VAs) translated by one or more logical MMUs into Physical Addresses (PAs) before being used for accessing the system interconnect, for example, to load a value from volatile

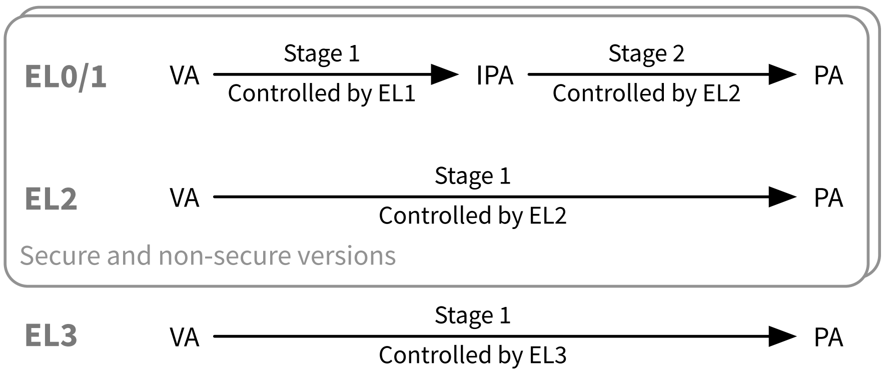


Figure 2.4: Address translation regimes in an ARMv8.4-A processor.

memory. The main difference between AArch64 and AArch32 is the size of the VAs used. AArch64 uses 64-bit and AArch32 32-bit virtual addressing.

Translation regimes. Depending on the EL a VA is accessed from, the VA is translated using one of multiple *translation regimes*. The translations regimes of an ARMv8.4-A processor are visualized in Figure 2.4. Translation regimes are configured using translation tables, which are multi-level memory structures allowing page-granular mapping of input to output addresses. The CPU provides separate regimes for EL0/1 and EL2 in both secure and non-secure versions and a single regime for EL3. This means that the same VA can be translated to five different PAs based on in which of these contexts the accessing software is running.

A translation regime can have one or two stages. A stage 1 translation translates from Virtual Addresses (VAs) to *either* Physical Addresses (PAs) *or Intermediate Physical Addresses (IPAs)*[1]. A stage 2 translation *always* translates from IPAs to PAs. In virtualization terms, a stage 2 translation is a *Second Level Address Translation (SLAT)*. As such, it allows a hypervisor to run multiple Virtual Machines (VMs) in the same guest-physical address space, as detailed in Section 2.2.2. Each stage of a regime is configured independently and from a specific EL. Stage 1 translations are normally controlled by the EL the regime belongs to, whereas stage 2 translations are controlled by a higher privileged EL.

The shown ARMv8.4-A model provides a superset of the regimes of previous architecture versions. Previous versions of ARMv8-A do not provide a secure EL2. Hence, the corresponding EL2 regime and the secure EL0/1 stage 2 regime are not present there. In ARMv7-A, additionally, EL3 and its translation regime are omitted.

[1]Intermediate Physical Address (IPA) is the term ARM uses for what is commonly referred to as *Guest-Physical Address (GPA)* in an abstract virtualization context. We use both terms interchangeably.

Attributes and permissions. Translations are configured using translation tables. Aside from the target address, a Page Table Entry (PTE), controlling the translation of a single page, is configured with attributes and access permissions for the underlying memory. With its *attributes*, the controlling entity can specify if a page should be cacheable and if it is shared by multiple cores. *Access permissions* determine which types of accesses are allowed to the memory of a page. Besides the typical *read* and *write* permissions, the execution of a page can be restricted using the Execute Never (XN) flag. Furthermore, execution can be restricted based on the executing context using the Privileged Execute Never (PXN) and Unprivileged Execute Never (UXN) permissions.

The secure and non-secure worlds can have different *physical* address spaces. For example, some of the system's memory might only be accessible to the secure world. Therefore, each translation is additionally associated with a Non-Secure (NS) flag with which secure ELs may specify if the corresponding memory should be accessed as secure memory. For non-secure translations, the flag is ignored and accesses are always non-secure.

In translation regimes with two stages, attributes and permissions of both stages are combined so that always the more restrictive setting applies. For example, this allows a hypervisor to prevent a guest from executing memory the guest marked as executable.

2.2 Virtualization

The basic idea of *virtualization* is to create multiple virtual instances of *one* physical resource. One of the most widely used forms of virtualization is *virtual memory*. In modern, preemptive OSs, each application process has its own full-sized *virtual address space*, despite the fact that the actual physical memory of the platform is typically much smaller. A controlling entity, in this case the OS kernel, manages the virtualization, i.e., the translation from virtual to physical memory and the allocation of actual physical memory for each process. Today, the term virtualization is mostly used to describe the virtualization of a full computing platform and its hardware resources, i.e., *platform virtualization*. Virtualization improves flexibility and allows for optimal usage of the virtualized resource. Furthermore, virtualization can improve security by isolating software from direct access to a physical resource.

2.2.1 Forms of Virtualization

There are two basic roles when virtualizing a resource: A *host* controlling the virtualization of the physical resource and one or more *guests* using the created virtual instances of the resource. Virtualization can be used on different *layers* in a computing system. Figure 2.5 shows a system employing multiple forms of virtualization on different layers:

Hardware. On this layer, actual hardware resources, such as the CPU and storage devices, are virtualized. Figure 2.5 shows *platform virtualization*, a specific form of hardware

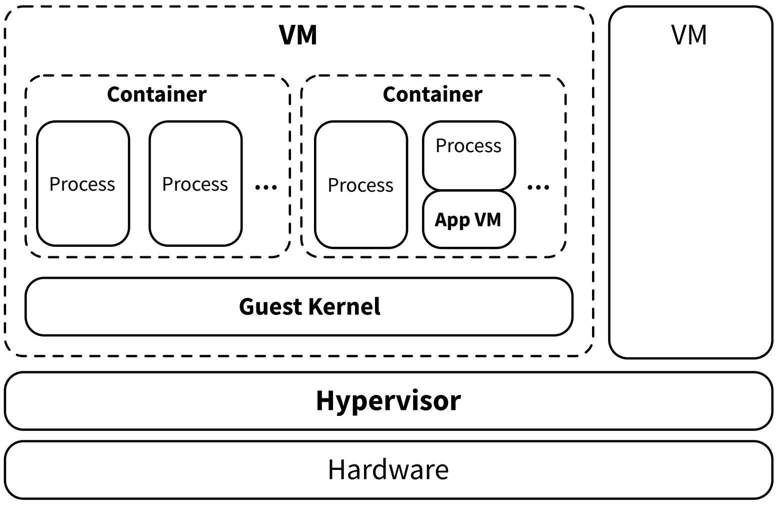


Figure 2.5: Different layers of virtualization in a single system.

virtualization in which the entire hardware platform is virtualized. The virtualization host is a hypervisor or VMM[1] creating one or more VMs for guest OSs to run in. Hypervisors for platform virtualization can be included in a host OS (Type-2) or can be standalone bare-metal software (Type-1).

Operating System. On this layer, the OS itself is virtualized, creating multiple *containers* with the same guest OS as the host OS. Each container consists of multiple processes for which OS resources such as Process Identifiers (PIDs), mounts, networks and users are separated from other containers and processes. For the container's processes, this creates the impression of running solely on the OS kernel.

Application. Applications are virtualized by a user space runtime, such as the Java Virtual Machine (JVM), abstracting and translating all accesses to the actual OS. Virtualized applications can run on all systems for which the corresponding runtime environment is available.

Especially in the context of hardware virtualization, we differentiate between two basic *types* of virtualization:

Full virtualization. A resource is fully virtualized, meaning that the possibly unaware guest uses its virtual resource as if it was the actual physical resource. Consequently, this type of virtualization does *not* require changes to the guest. It is *transparent* to the guest.

[1]We use the terms hypervisor and VMM interchangeably.

Paravirtualization. The guest is aware of the virtualization and uses an interface provided by the host to interact with its virtual resource. This type of virtualization requires guest software adapted to use the virtualization interface.

In the context of this thesis, primarily the hardware virtualization layer is of interest. Full platform virtualization, supporting full-fledged VMs with unmodified guest OSs, poses difficult challenges. In a fully virtualized platform, VMs provide an execution environment in which guest software runs as if it was running solely on the actual hardware. For this to be *efficient*, a guest must be able to execute most if its instructions natively, i.e., directly on the hardware, without interfering with the hypervisor or other VMs. Popek and Goldberg formulated [PG74] a basic requirement for a CPU's ISA in order to support *efficient* platform virtualization. They define two groups of special instructions in an ISA: *Privileged* instructions, which can only be executed by a privileged mode and trap if executed by an unprivileged mode, and *sensitive* instructions, which are influenced by or influence the configuration of system resources. According to Popek and Goldberg, an ISA is efficiently virtualizable if the set of sensitive instructions is a subset of the set of privileged instructions. For such an ISA, a hypervisor can be built that allows the guest to run on the actual hardware while still preventing it from accessing or modifying system resources by trapping the according, sensitive instructions. On an ISA that does *not* provide this property, virtualization requires complex and expensive techniques, such as (dynamic) binary translation.

An ISA fulfilling the Popek and Goldberg requirement does not necessarily *actively* support virtualization. A hypervisor built only based on this single requirement still has to provide a lot of complex virtualization functionality, for example, to realize memory virtualization. Furthermore, the virtualization possibly requires a vast amount of context switches for trapping sensitive instructions. Both aspects greatly reduce the performance of the virtualization. Therefore, modern CPUs and platforms provide hardware support for virtualization, reducing traps and hypervisor complexity, which we discuss in the following.

2.2.2 Hardware-assisted Virtualization

The performance of platform virtualization depends on the time spent in the hypervisor. Therefore, in order to reduce the virtualization overhead, modern platforms contain a multitude of virtualization features in hardware, minimizing the number of hypervisor traps and the necessary hypervisor functionality. In the following, we discuss such *hardware-assisted* virtualization of different components of a computing platform.

CPU. The basic CPU virtualization feature is the addition of another execution mode for the hypervisor to run in. This mode is higher privileged than the traditional, privileged kernel mode, which remains available for the guest kernel inside a VM. As discussed in Section 2.1.2 and shown in Figure 2.2, ARMv8-A processors provide such a mode in form of EL2. The guest kernel can execute the subset of sensitive instructions that involves the configuration of its own VM, i.e., *guest-sensitive* instructions, independently, without requiring traps into the hypervisor. The CPU typically provides a special instruction for calls to the hypervisor from VMs, the Hypervisor Call (HVC) instruction in case of ARM, and optional, configurable traps for guest-sensitive instructions.

Memory. As discussed in the beginning of this section, memory in modern CPUs is already virtualized in the form of *virtual memory*. This virtualization is used to provide each user space process with a full, distinct virtual address space. In order to realize this feature, the MMU of those CPUs translates VAs to PAs as configured by the OS kernel for the running process.

In CPUs with hardware-assisted virtualization, another translation stage, the *Second Level Address Translation (SLAT)*, is added. The guest still controls the first stage of translation, which now translates into Guest-Physical Addresses (GPAs) (i.e., IPAs on ARM). The SLAT is configured by the hypervisor and translates GPAs to actual PAs. The SLAT takes the same role for VMs as the first stage translation for user space processes and allows the hypervisor to run multiple VMs using the same guest-physical address space. As discussed in Section 2.1.3 and visualized in Figure 2.4, ARM processors support SLAT in form of the stage 2 EL1/0 translations. Additionally, virtualization support often also introduces *Virtual Machine Identifiers (VMIDs)*. Analogously to *Address Space Identifiers (ASIDs)* for user space processes, VMIDs can be set by the hypervisor and are then used by the hardware to tag cache and Translation Lookaside Buffer (TLB) entries allowing for fast and efficient maintenance operations on these shared resources.

Devices. Peripheral devices can be virtualized, i.e., shared between multiple VMs, using different software techniques. First, they can be *emulated* by the hypervisor, requiring expensive traps on each guest access to the device. Second, devices can be *paravirtualized*, requiring special drivers in the guest and the hypervisor. Finally, devices can be virtualized in hardware or assigned exclusively to a single VM, both of which requires the *direct pass-through* of the device into a VM. CPUs with hardware-assisted virtualization typically provide features for improving the performance of device emulation and especially of device pass-through.

To speed up device emulation, CPUs provide detailed analysis information for traps from VMs. For example, with ARM virtualization, when a guest accesses memory mapped by the hypervisor as an emulated device, the trap that is triggered by the permission fault in the SLAT typically provides information about the type of access and the exact access address.

For device pass-through, CPUs with hardware-assisted virtualization typically provide means to directly assign corresponding interrupts to the targeted VM. This means that, in the best case, no interference by the hypervisor is necessary anymore during normal device operation. For DMA devices, *Input-Output Memory Management Units (IOMMUs)* with hardware-assisted virtualization provide a SLAT, analogously to the SLAT on the main CPU.

2.2.3 AMD SEV and SME

In the strongly hierarchical privilege model of virtualized system, higher layers inherently have to trust the lower layers managing them. For example, an owner of a VM has to trust his server operator providing the hypervisor managing his VM, as the operator can, theoretically, access all the VM's data. In order to alleviate this problem, AMD introduced Secure Encrypted Virtualization (SEV) [KPW16], a technology for encrypting the memory of VMs, protecting it from accesses by other VMs and even privileged system software. AMD SEV is based on AMD's hardware-assisted virtualization support and Secure Memory Encryption (SME) [KPW16], another new feature, enabling transparent full memory encryption. In the following, we shortly introduce both technologies and the supplemental SEV Encrypted State (SEV-ES) feature.

AMD SME. SME is a hardware extension for full main memory encryption. Its goal is to protect the data in DRAM of the entire system against a strong physical attacker trying to extract data, for example, via a cold boot attack. SME introduces an additional bit, the *C-bit*, in the CPU's address translation table entries indicating that a page should be encrypted. Pages marked with the C-bit are encrypted by the DRAM controller using a single key randomly generated at system startup. The key is managed by a special security co-processor, the AMD Secure Processor (AMD-SP), and never exposed to software on the CPU. Transparent SME (TSME) is an additional operation mode of SME that encrypts all memory pages regardless of their C-bit value. TSME can be used to encrypt systems with legacy system software not supporting the C-bit management.

AMD SEV. SME uses a single key and hence only protects against a *physical* attacker. AMD SEV encrypts VMs and the hypervisor with different keys, in order to protect them from each other via cryptographic isolation. SEV is an extension of AMD's virtualization technology AMD-V. AMD-V ensures separation inside the CPU by tagging code and data with an VM ASID. As with SME, data is encrypted by the DRAM controller when entering or leaving DRAM but using different keys based on the VM ASID. As for SME, the encryption is managed on a per-page basis using the C-bit in the translation tables. With AMD-V's SLAT, a guest manages its own translations and can use SEV's C-bit to mark pages for encryption independently of the hypervisor. By disabling the C-bit in the VM translations, a guest can create memory regions shared with the hypervisor. If the hypervisor enables encryption of these regions in the SLAT, they are still encrypted in memory but can be accessed by both, VM and hypervisor. SEV does not prevent a hypervisor from switching pages belonging to the same VM in the SLAT or from replaying old pages of the same VM.

Keys are managed by the SEV firmware on the AMD-SP co-processor and, as for SME, are never exposed to software running on the main CPU. The SEV firmware offers an interface to the hypervisor for managing VMs. Aside from basic functions, such as launching and running VMs, the interface provides functionality for proving its authenticity and for attesting the secure, SEV-enabled launch of a guest VM towards a remote party, such as the VM owner. For both, the AMD-SP contains key material from AMD and the platform owner. For the launch attestation, the VM owner can provide a guest image which gets loaded and measured together with other SEV-related guest state by the SEV firmware before being launched. The SEV firmware then generates a signature over the measurement, with which the VM owner can ensure the secure launch of his VM before sending security-critical data, such as the VM's disk encryption key.

AMD SEV-ES. Like SME, SEV, in its base form, only encrypts data in DRAM and leaves other important information, such as the VM's registers state, unencrypted and therefore available to a possibly malicious hypervisor. SEV-ES [Adv18] is an extension to SEV, encrypting all information about a VM's state not required by the hypervisor to function properly. For this, the contents of the Virtual Machine Control Block (VMCB) are partitioned into an unencrypted *Control Area* and an encrypted *Safe Area*.

2.3 Trusted Execution Environments

A Trusted Execution Environment (TEE) is a secure area in a system, which is logically separated from the system's normal operation state. The normal operation state is then often referred to as *non-secure* or *normal world* [ARM17] or Rich Execution Environment (REE) [Glo17], as opposed to the *secure world* provided by the TEE. The separation mainly affects the system's main CPU, but typically also extends to other resources in the system, for instance, to the system's main memory. On the system's CPU, the TEE runs in parallel but isolated from the CPU's normal privilege modes, including even the most privileged ones such as a hypervisor mode or the Intel Secure Management Mode (SMM).

TEEs and their applications provide security-critical functions to normal world applications. For example, a TEE can be used to handle cryptographic operations for Digital Rights Management (DRM), so that the corresponding key material is never exposed to the normal world. Regarding its usage model, a TEE is comparable to a dedicated security device, such as a Secure Element (SE), in that it is typically used in a request-response fashion. Apart from that, there are some important differences between a dedicated security device and a TEE. A TEE is less secure, without real physical separation and without special hardening against hardware attacks. For example, TEEs often require an intact SoC package to keep their security guarantees. On the other hand, a TEE, with its access to the host CPU, is functionally more capable than an SE and can, for instance, use the full performance of the CPU and implement high-performance communication with the normal world.

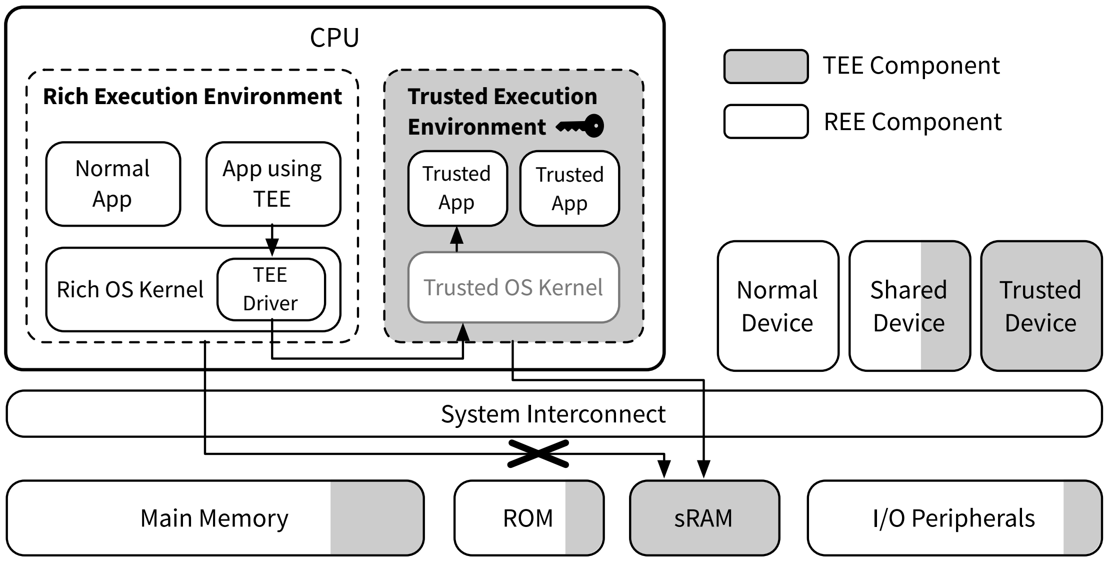


Figure 2.6: Typical architecture of a system with TEE.

Figure 2.6 shows a typical architecture of a system employing a TEE. The shown architecture is roughly based on the TEE specifications [Glo17] by GlobalPlatform. Apart from the logical separation inside the CPU, a TEE also requires a concept for the security of other resources in the system. In the shown architecture, the TEE's isolation is extended to system resources in a generic way. Resources can be *normal*, *trusted* or *shared* between TEE and REE. Trusted resources and trusted parts of shared resources can *only* be accessed by TEE components, i.e., the TEE modes in the CPU and other trusted bus master devices. The isolation of TEE components can be realized using physical isolation, hardware logic isolation, such as an additional "secure" signal on the interconnect, or cryptographic isolation using unique, device-specific key material only accessible to the TEE.

A TEE typically provides means to protect the integrity and confidentiality of its code and data. Load-time integrity is often established using code signatures in combination with a secure boot or a similar mechanism. Run-time integrity and confidentiality is ensured by the TEE's isolation mechanisms, preventing access of normal components in the system to TEE memory and storage resources. In our typical architecture shown in Figure 2.6, the Static Random-access Memory (SRAM) of the system, which is a special volatile memory type that is part of the processor die, is trusted and exclusively assigned to the TEE. Hence, it cannot be accessed by normal world components and can be used to securely store code and data of the TEE. As part of the SoC die, it is protected against hardware attacks targeting, typically external, RAM modules. The isolation of SRAM is normally realized using a hardware logic that prevents accesses to a resource based on a special, additional signal on the interconnect. TEE data in shared RAM and persistent storage devices is often protected using cryptographic isolation. For this, the TEE uses its unique and secret key material to encrypt data from its secure memory into memory or persistent storage shared with the normal world. While the normal world can access the encrypted data, it cannot decrypt it, establishing a cryptographic isolation.

Some TEEs provide support for software attestation. With this, a TEE or one of its components can prove its integrity towards a third party, the *verifier*. Attestation typically requires a shared secret or a certified key pair trusted by the verifier, whose availability is bound to the TEE and its integrity by hardware means. This key material is typically *provisioned* to the TEE or the hardware itself before deploying the device for productive use. A TEE might support provisioning and attestation in hardware or in software. For the latter, the provisioned attestation key can be protected using cryptographic isolation as described before.

In the following, we shortly introduce the two currently most important TEE implementations in modern CPUs, namely the ARM TrustZone and Intel SGX.

2.3.1 ARM TrustZone

The TrustZone [AF04; ARM17] is ARM's TEE implementation. It closely matches the generic TEE concept described in the previous section, introducing a secure logical separation inside the CPU and a mechanism extending the separation to system resources.

Inside the ARM CPU, TrustZone introduces the *secure world*, mainly consisting of a set of new modes or ELs. Those modes have already been introduced as part of the ARM architecture privilege model in Section 2.1.2 and are shown in Figures 2.2 and 2.3. The TrustZone secure world basically mirrors the normal world, providing the ability to run a secure OS with secure applications in parallel to the normal world. When executing in the secure world, the CPU allows access to secure resources, such as secure configuration registers. Additionally, the TrustZone introduces a monitor mode (ARMv7-A) or EL3 (ARMv8-A) to host a secure monitor component responsible for switching between normal and secure world. This mode is *always* secure but may change its view on the system freely, allowing it to access both worlds for realizing the world switch. A world switch is typically initiated by a secure interrupt or a request from the normal world for a function offered by the secure world via an Secure Memory Card (SMC) call.

The TrustZone extends its separation to the system interconnect by introducing an additional signal on the system bus, indicating if an access is secure. As explained in Section 2.1.3 and shown in Figure 2.4, the TrustZone secure world provides its own, independent address translation regimes. The TrustZone, furthermore, introduces the *Non-Secure (NS) bit* for translations, indicating if the underlying memory should be accessed securely, i.e., with the corresponding secure signal set. *Only* secure world modes are able to unset the NS bit in their translations resulting in a secure access to the system bus. All normal world translations always have the NS bit set, independently of the value chosen by their managing software. This mechanism builds the basis for partitioning the physical address space into secure and non-secure versions. A resource mapped into the physical address space can be TrustZone-aware, permitting or preventing accesses based on their security state. Bus masters other than the CPU can be TrustZone-aware and access the secure address space, but non-aware masters default to the non-secure address space. ARM provides several hardware blocks to implement the partitioning of hardware resources between secure and non-secure address spaces. The TrustZone Memory Adapter (TZMA) can be used to partition on-SoC Read-Only Memory (ROM) or SRAM into a secure and a non-secure region. The TrustZone Address Space Controller (TZASC) can be used to partition an address range into multiple secure and non-secure regions, e.g., to share one RAM device between worlds. For other bus master devices, ARM provides TrustZone-aware DMA controllers that can be configured to reject accesses to secure address regions.

Entries in TLBs and cache lines are tagged with their security state, so that no flushes are required when switching worlds. For memory shared between worlds, this creates a possible aliasing problem as a secure and a non-secure cache line can co-exist for the same memory. Hence, secure world software must ensure that it accesses shared memory with the NS bit set. While explicit cache flushes from the normal world cannot flush secure cache lines, cache line eviction as basis for cache timing attacks is still possible.

The ARM architecture requires a CPU to always start in its highest available privilege level (typically EL3). Furthermore, the privilege level can only be increased by triggering an exception, transferring execution to a code location previously specified by the targeted EL. Together with a secure boot process (see Section 2.4.1), this privilege model can ensure the integrity of TrustZone software. TrustZone-enabled ARM platforms typically provide some kind of secret device-specific key that is only available after a successful secure boot. This key can be used by TrustZone software in conjunction with secure SRAM to implement cryptographic isolation for persistent secrets, as described before.

The ARM TrustZone does not provide built-in architectural support for software attestation. Nonetheless, it typically provides the means for implementing an attestation mechanism. In a provisioning step, an attestation key pair or shared secret can be created in the TrustZone and be protected with cryptographic isolation, as described before. This key is then certified by or shared with the verifier. In its productive environment, the key can only be decrypted by the TrustZone software after a successful secure boot. The TrustZone software can then take measurements of the attested software and sign the measurements with its verifier-trusted key to attest the platform's software integrity.

2.3.2 Intel SGX

Intel SGX [McK+13; Int18; CD16] is a *user-level* TEE introduced first in the Skylake generation Intel Core processors. In contrast to the ARM TrustZone, Intel SGX deviates significantly from the classic GlobalPlatform TEE architecture, implementing typical tasks of a TEE secure OS and monitor component directly in hardware. Figure 2.7 shows an overview of an Intel SGX-enabled system. User mode (Ring 3) applications can move their

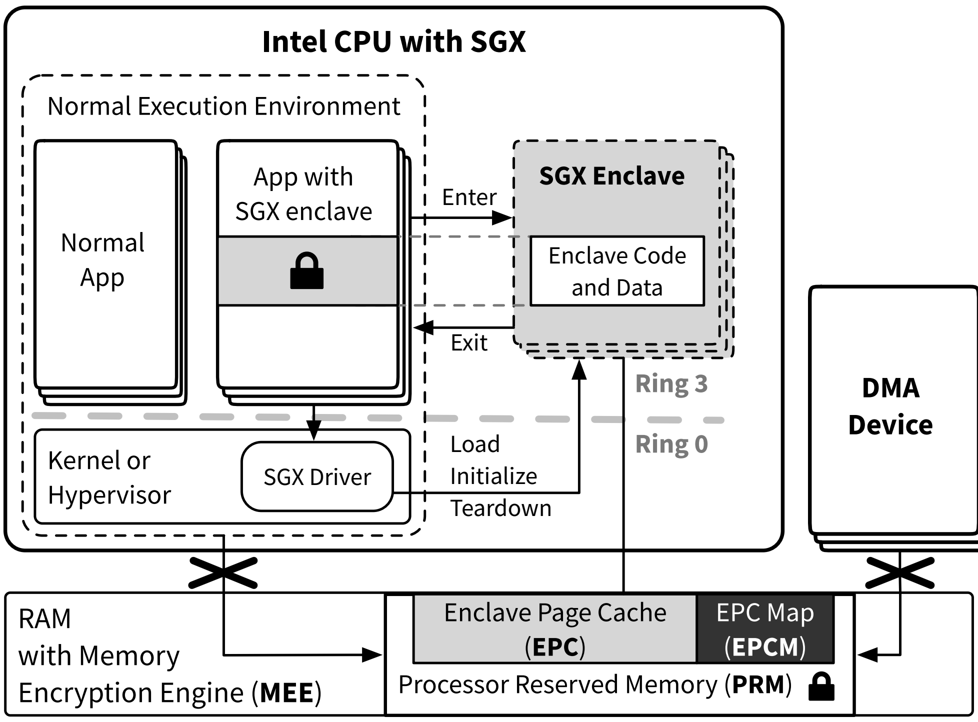


Figure 2.7: Architecture of an SGX-enabled Intel system.

security-critical components into SGX *enclaves*, which are integrity- and confidentiality-protected from each other, from other applications and from normal privileged *system software*, such as the kernel or hypervisor. Enclaves are created, initialized and torn down by system software, but can only be entered by user space applications. For all those operations, SGX introduces special, new processor instructions. Furthermore, as shown in Figure 2.7, Intel SGX introduces several new structures on the main memory level:

Processor Reserved Memory (PRM). The PRM is a configurable part of the main memory protected against all normal world accesses. As shown in Figure 2.7, this also includes accesses from DMA devices.

Enclave Page Cache (EPC). The EPC is a sub-region of the PRM storing the data and code of the system's enclaves. Pages in the EPC are encrypted by the Memory Encryption Engine (MEE) and only accessible by the enclave they belong to. Pages in the EPC are allocated to enclaves by privileged normal world software.

Enclave Page Cache Map (EPCM). The EPCM is a CPU-private memory region, inaccessible to any kind of software. SGX uses the EPCM to store metadata for EPC pages, such as the page's validity and type, owning enclave, access permissions and VA.

A new enclave is initialized by system software, executing special CPU instructions that create the enclave and fill it by copying data from a given location in the normal memory of the requesting application. During its creation, an enclave's contents and configurations are measured by SGX using a cryptographic hash function. After the initialization of an enclave, no further pages can be added and the measurement hash is finalized, establishing the *Enclave Identity*. Every enclave must provide a certificate and signature of its author. The certificate includes a measurement hash, a product ID, the enclave version, a vendor value for identifying special privileged Intel enclaves, and required enclave attributes. Before finalizing the initialization of an enclave, its signature is checked by SGX against the measurement hash. After a successful check, the signature and certificate establish the enclave's *Signing Identity*.

Enclaves can only be entered from unprivileged Ring 3 applications and run with the same user mode privileges as their application. In contrast to the ARM TrustZone, SGX enclaves do *not* provide independent address translations and, consequentially, use the same translations as their host applications, controlled and configured by the underlying privileged system software, i.e., the kernel and hypervisor. This is necessary for the system software in order to be able to manage EPC page allocations. Nonetheless, SGX stores the page access address and permissions as defined by the enclave author in the EPCM and uses those values to override possibly malicious address translations by the system software. Additionally, SGX prevents system software from mapping the same EPC page to multiple enclaves and ensures that the virtual address space of an enclave is exclusively mapped into the EPC. Furthermore, SGX offers functions to let the system software securely evict pages from EPC to normal memory. SGX encrypts evicted pages and uses nonces to protect the EPC from replay attacks by the system software. In contrast to the ARM TrustZone, SGX does not provide a generic mechanism to extend its secure state to hardware resources in the system. While SGX protects enclave data against DMA devices, it does not enable the implementation of SGX-aware devices other than the CPU and RAM.

An enclave can use a special instruction to request the generation of different keys based on its identity. Those keys are derived from a CPU-specific secret and can therefore not be generated on a different CPU. For example, a key can be requested that is bound to the specific enclave instance using its Enclave Identity. This allows an enclave to realize cryptographic isolation, securely storing its secrets in shared (persistent) memory. The key generation can also use the enclave's Signing Identity, allowing, for example, the sharing of keys between version updates or between different enclaves of the same author.

SGX provides an integrated mechanism for the attestation of an enclave's integrity. In a *local attestation*, an enclave uses a special instruction to prove its integrity towards another enclave on the same CPU by generating a signed report which can be checked by the receiving enclave. Furthermore, SGX offers a remote attestation allowing an enclave author to remotely verify an enclave's integrity, for example, before providing specific key material. The remote attestation is realized using a local attestation towards a special, privileged

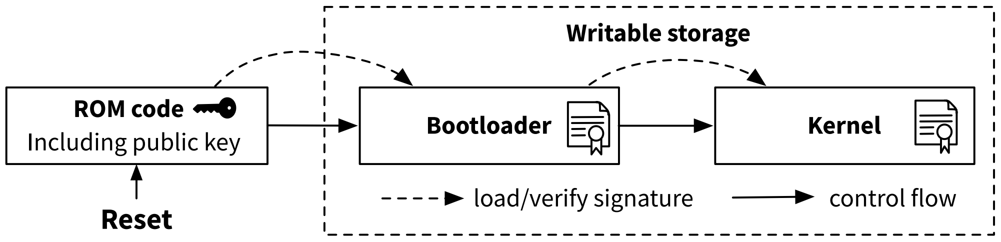


Figure 2.8: Basic concept of a secure boot.

Quoting Enclave by Intel and a CPU-specific *Provisioning Secret* shared with Intel, which is used by another privileged enclave, the *Provisioning Enclave*, to generate attestation keys.

2.4 Boot Integrity and Attestation

Ensuring the integrity of firmware and software loaded during a system's boot process is fundamental in many application scenarios. In the following we discuss two boot integrity concepts with different goals. The *secure boot* enforces that only signed software can be loaded on a protected platform. The *measured boot* stores measurements of boot stages in a secure location, for example, to prove the platform's boot integrity to a remote party afterwards.

2.4.1 Secure Boot

The *secure boot* [AFS97] is a technique widely used on mobile and embedded devices, such as smartphones, ensuring that only software with a valid signature can be loaded during the boot process. The basic concept is illustrated in Figure 2.8, depicting a simplified three-stage secure boot process. As basis, the secure boot requires an immutable piece of code that is guaranteed to be executed directly after device reset, before any other code. This component, which we call *trust anchor* in the following, is typically realized using ROM, as depicted in Figure 2.8. Additionally, the secure boot requires one or more public keys stored immutably in hardware for verifying code signatures. This is normally realized with One-Time Password (OTP) fuses that can be programmed in a provisioning step to store the device owner's keys[1].

The secure boot starts with the trust anchor loading the first mutable boot stage and verifying its signature with the platform's fused keys before handing over control. The loaded boot stage then, in turn, uses the platform's or own keys to verify the signature of the next stage, building up a *chain of trust*.

On ARM platforms, the secure boot is strongly connected to the initialization of the different privilege levels, the ELs. Figure 2.9 illustrates a typical secure boot sequence on

[1]Typically, only hashes of the public keys are stored to save costly OTP fuses.

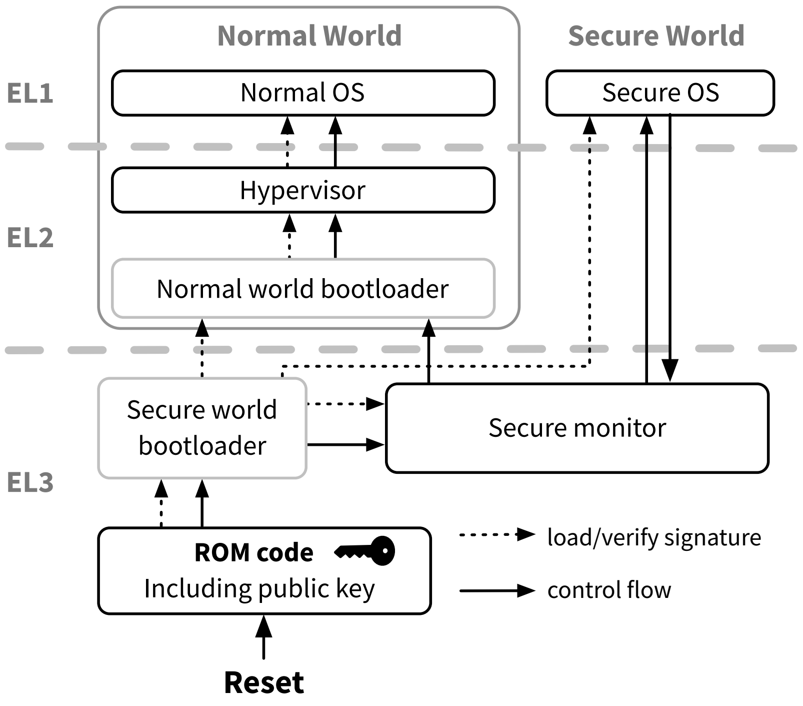


Figure 2.9: Typical secure boot on an ARMv8-A CPU.

an ARMv8-A CPU. The CPU always starts its execution in the highest available EL, which is EL3 in CPUs with TrustZone support. The trust anchor ROM code securely loads a signed secure world bootloader which loads and checks the integrity of a normal world bootloader, the secure monitor and the secure OS kernel. Then, it passes control to the secure monitor, which, among others, initializes the EL3 exception vector table for handling SMCs and the world switch functionality. Afterwards, the monitor uses an exception return (`eret`) to drop to the secure OS in secure EL1, which initializes itself, including the secure EL1 exception vectors. After finishing its initialization, the secure OS passes control back to the secure monitor using a specific SMC. With this step, the secure world boot is finished and the monitor uses an exception return to pass control to a normal world bootloader, such as *u-boot* or a Unified Extensible Firmware Interface (UEFI) bootloader. The normal world bootloader finishes the secure boot by loading an (optional) hypervisor and the normal OS kernel, both of which initialize their respective EL exception vector tables.

2.4.2 Measured Boot

In contrast to a secure boot, a *measured boot*, sometimes also called *trusted* or *authenticated* boot, does not enforce that only signed software is booted on a device but enforces that measurements, i.e., hash digests, of the loaded boot stages are stored into a *secure component* before executing them. The secure component must be separated from the normal execution environment, either physically or logically, and ensure that measurements can only be written but not removed. A typical example for a physically separated component is a Trusted Platform Module (TPM) [Tru11] connected to the system, whereas a Firmware Trusted Platform Module (fTPM) [Raj+16] in the TrustZone is an example for a logically separated secure component. Similar to the secure boot, the measured boot requires specific *trust anchor* code that is immutable and guaranteed to be executed as the first stage of the boot process. The trust anchor, often called Core Root of Trust for Measurement (CRTM) in the context of TPMs, starts the measured boot by measuring the following boot stage and storing the measurement in the system's secure component. Then it passes control to the measured stage, which, in turn, measures the following stage. After finishing the boot chain, the secure component contains a secure log of the platform's boot process, which cannot be modified retroactively. In contrast to the secure boot, the measured boot does not prevent a maliciously modified stage from being started, but ensures that the modification is securely reflected in measurements in the secure component. The measured boot is a basic mechanism that enables different use cases, two of which we discuss in the following:

Platform attestation. In a platform attestation, the measurements taken during the measured boot are used to prove the platform's integrity towards a (remote) *verifier*. As for other forms of software attestation, the attestation requires a secret key in the attesting entity, i.e., the secure component, that is trusted by or shared with the verifier. This key is used by the secure component to sign the measurements received during the measured boot, proving the platform's boot integrity towards the verifier.

Sealing. Depending on the measurements received during the measured boot, the secure component can provide or prevent access to its secrets or secure functions. For example, it can combine measurements with its own secrets and provide cryptographic functions using the resulting keys to its host platform. With this, data can be encrypted so that it can only be decrypted again if the host platform is in the same state, i.e., delivered the same measurements to the secure component during the boot process. In the context of the TPM, the described mechanism is called *sealing*.

System Architecture 3

In this chapter, we introduce a generic, mobile system architecture that serves as a common basis for the security concepts we propose throughout the thesis. As shortly discussed in our introduction in Section 1.2, the main design goal for our system architecture is the implementation of multiple, increasingly trusted layers of separation, isolating security-critical components and functionality from the less or non-trusted majority of the system's components. Figure 3.1 shows an overview of our generic system architecture. It consists of two main physical entities: the target platform and the security token. The *target platform* is a high-performance, general-purpose computing device. It contains a CPU that provides several levels of logical separation, building the foundation for our target platform architecture. The *token* is a small, low-performance special-purpose security device attached to the target platform. As physically independent device, it provides the strongest separation and is used for protecting the most valuable secrets in the system.

In the following, we discuss the architecture and the hardware characteristics of both target platform and token. Afterwards, we introduce our generic attacker model, building the basis for the specific attacker and threat models of the concepts introduced in the following chapters.

The architecture presented in this chapter is the result of combining several architectures that have been published with our corresponding security concepts [HW15; HHW17a; Hor+14b; HWE16; Hor+14a].

3.1 Target Platform

The target platform is the device whose secrets and integrity should be protected by our architecture and concepts. Conceptually, it can be any device that requires strong protection because it is highly exposed to possible attacks and contains valuable data. Such a device could, for instance, be a smartphone or a laptop.

Figure 3.1 illustrates the basic architecture of the target platform on the left, depicting the protection mechanisms introduced in this thesis and their location in the separation

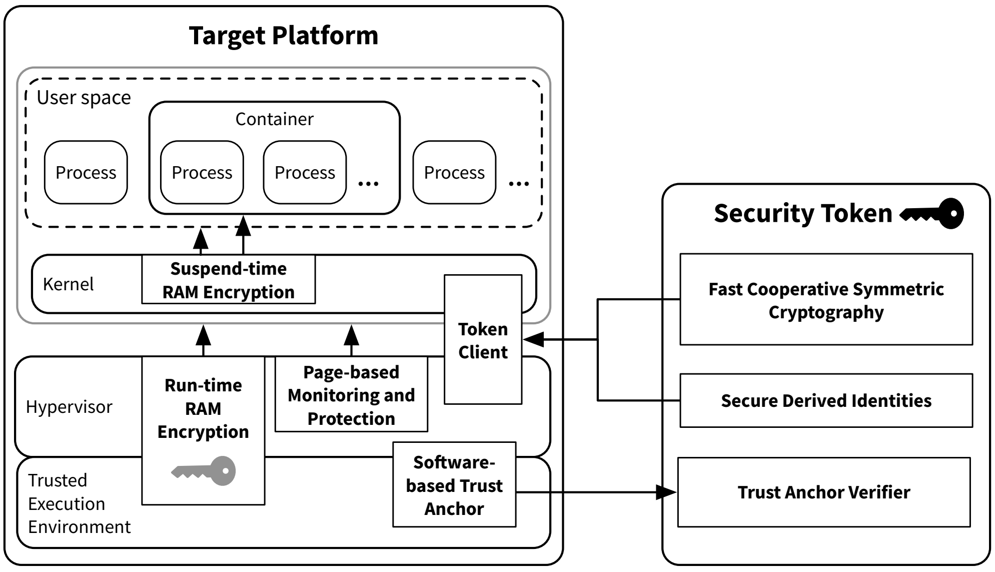

Target Platform
User space
Container
Process
Process
Process
Process
...
...
Kernel
Suspend-time RAM Encryption
Token Client
Hypervisor
Run-time RAM Encryption
Page-based Monitoring and Protection
Trusted Execution Environment
Software-based Trust Anchor
Security Token
Fast Cooperative Symmetric Cryptography
Secure Derived Identities
Trust Anchor Verifier

Figure 3.1: Generic system architecture.

layers of the target platform. The architecture is generic in the sense that some of the concepts presented in this thesis require specific hardware features. Most of the concepts are orthogonal so that a subset of them can be implemented if the target platform's hardware does not support a specific feature. In the following, we further specify the target platform by discussing its layers, components and corresponding hardware requirements:

TEE. The TEE (see also Section 2.3) represents the most secure layer inside the target platform. It provides secure key storage for our run-time memory encryption (see Section 6.3) and houses SobTrA, proving the system integrity towards the token (see Chapter 4) in order to unlock its security functions. SobTrA is currently specific to ARM Cortex-A processors and can therefore only be used on a target platform employing such a processor.

Our architecture assumes a target platform with a classic GlobalPlatform-like TEE, such as ARM TrustZone (see Section 2.3.1). A user-level TEE, such as Intel SGX (see Section 2.3.2), requires a different architecture with a user space daemon using an enclave for the memory encryption key and SobTrA in the highest available normal privilege mode.

While our target platform architecture uses a TEE, it is not strictly necessary. If the target platform does *not* provide a TEE, SobTrA can be run from the next highest privilege level. Section 6.5.3 describes alternative key storage concepts without TEE for our run-time memory encryption.

Hypervisor. The hypervisor layer contains the components for run-time memory encryption (see Section 6.3) and page-based execution monitoring (see Chapter 5). The first protects a guest's confidentiality, the second its run-time integrity. Both require basic hardware-assisted CPU virtualization features (see Section 2.2.2), more specifically SLAT memory management and trapping of specific instructions. In our target platform, these functions are abstracted for both components by a minimal, custom hypervisor which supports only a single guest. The hypervisor layer and its functions are completely transparent, without an explicit guest interface. Hence, the layer has a minimal TCB and attack surface. Conceptually, both security concepts could also be integrated into a full-fledged hypervisor supporting multiple guests.

Since hardware-assisted virtualization is supported by all major CPU architectures, no specific CPU is required by our components in this layer. As both of our security concepts on this layer require frequent context switches, ARM CPUs are preferable because of their lightweight virtualization.

Optionally, the hypervisor layer may contain a client component for accessing security functionality offered by the token.

Kernel. The kernel layer implements our suspend-time memory encryption (see Section 6.6) and the token client. The latter enables kernel and user space to access security functions offered by the token (see Chapter 7), such as external symmetric cryptography (see Section 7.2), and is used, for example, to realize secure FDE.

The kernel layer does not require any specific hardware features. Since our suspend-time memory encryption is realized as a Linux-specific design, the target platform must run a Linux kernel in order to use it.

User space. The user space contains several parallel processes, realizing the actual, use case-specific functionality of the target platform. Using kernel functionality, processes can be grouped into isolated, functional groups called *containers*, as described in detail in [Hub+15].

Our kernel-based security concepts leverage process and container abstractions. The suspend-time memory encryption can either encrypt the whole guest user space when the target platform is fully suspended or groups of processes by suspending containers. Our token-based external cryptography uses user space secrets to isolate its operations for different user space contexts.

The modularity of our generic architecture allows for different concept combinations, depending on the actual target platform's hardware. While a target platform with an Intel CPU supports a subset of the concepts, our typical target platform is an ARM application processor with virtualization support and TrustZone TEE. In order to be useful in an actual application scenario, the target platform, furthermore, provides different Input/Output (I/O) devices, such as a display, user input devices, and a network interface.

3.2 Security Token

The security token is a small, hardened special-purpose device connected to the target platform, as shown in Figure 3.1 on the right. Being physically separated, the token can act as an additional, strongly isolated layer storing the system's most valuable secrets and providing functions based on them to the target platform.

There are no hard constraints regarding the exact hardware of the token and its physical connection to the target platform. The token can be an internal or an external device, optionally removable, and can be connected to the target platform using a variety of interfaces, such as Universal Serial Bus (USB), Serial Peripheral Interface (SPI) or Secure Digital Input Output (SDIO). In some cases, for example, if the token is an internal co-processor, it might even be connected directly to the target platform's system bus. Examples for actual token devices are smart cards in different form-factors and USB devices, such as the USB armory [Inv].

Since the token can have different hardware and many different physical forms, our system architecture, as depicted in Figure 3.1, does not define a specific token software architecture. Instead, we define specific requirements to the token's hardware imposed by the three token-based security concepts we propose in the thesis:

Hardened. The token must provide means to protect its integrity and confidentiality. The actual level of hardening depends on the application scenario. In a basic form, the token's boot-time integrity must be protected against manipulations, e.g., by a secure boot (see Section 2.4.1), and the token must encrypt its persistent data.

Independent timer. The token must provide an independent time source in order to be able to measure the time the target platform needs to calculate the SobTrA checksum, proving its integrity (see Chapter 4).

Cryptography. The token must be able to perform cryptographic operations. For our external cooperative cryptography concept (see Section 7.2), symmetric cryptography must be supported. For our derived identity storage concept (see Section 7.3), asymmetric cryptography is required. Optimally, the token provides hardware acceleration for these operations.

Device secret. For our external cooperative cryptography concept (see Section 7.2), the token must provide a secret, device-specific symmetric key. Optimally, this key is fused into the hardware, cannot be read but only used, and is only available if the boot-time integrity of the token is ensured.

Unlock input. The token might optionally provide independent user input, such as a Personal Identification Number (PIN) pad or fingerprint reader, for unlocking and, thus, protecting its security functions. While such a feature improves security of most of the concepts (cf. Section 7.2.5), it is optional considering that many token form factors do not allow it. Typically, the number of tries for unlocking the token should be limited, in order to prevent brute forcing, for example, when using a PIN.

LED. The token might optionally contain a Light-Emitting Diode (LED) for indicating specific activity to the user. In our external cooperative cryptography concept (see Section 7.2), this optional token feature is used to improve security against certain attacks.

As for the target platform, the security functions on the token are orthogonal and can be combined differently, depending on the actual token's hardware and the requirements it meets. This modular architecture allows for a variety of devices to be used as tokens, implementing all or a subset of the proposed functions.

3.3 Generic Attacker Model

In the following, we introduce different attackers with varying capabilities to our architecture. Our attacker model is generic in the sense that it does not determine a specific attacker but provides the building blocks for the specific attacker models defined for each of our security concepts in the following chapters. We show that our generic system architecture is suited to build a system secure against those specific attackers in separate security discussions for each of our security concepts. Regarding concrete attacks, this section only gives a coarse overview, deferring detailed attack descriptions to the chapters introducing our corresponding defense mechanisms.

The security concepts proposed in this thesis try to protect the integrity and confidentiality of different assets in the target platform. The general *goal* of the attacker always is to break those protections and to gain control of the target platform and/or to extract the protected assets. The specific attacker goal, e.g., the actual target asset, depends on the concrete security concept discussed.

We define two *types* of attackers. A *remote attacker* who has no physical access to the system's hardware, attacking the system via its remotely accessible interfaces, and a *physical* attacker with direct access to the hardware, conducting hardware attacks or attacking the system via its local interfaces. Figure 3.2 depicts both attacker types in the context of our system architecture. The figure furthermore shows the increasing trust levels of the system's components, beginning with untrusted user space processes and ending with the fully trusted token. Attackers have *capabilities* based on their type and their level of expertise. The more capable an attacker, the higher the trust level of the system's components he is able to attack successfully, using his type-specific attack vectors. In the following, we introduce the specific attack vectors of our attacker types before discussing their common capabilities.

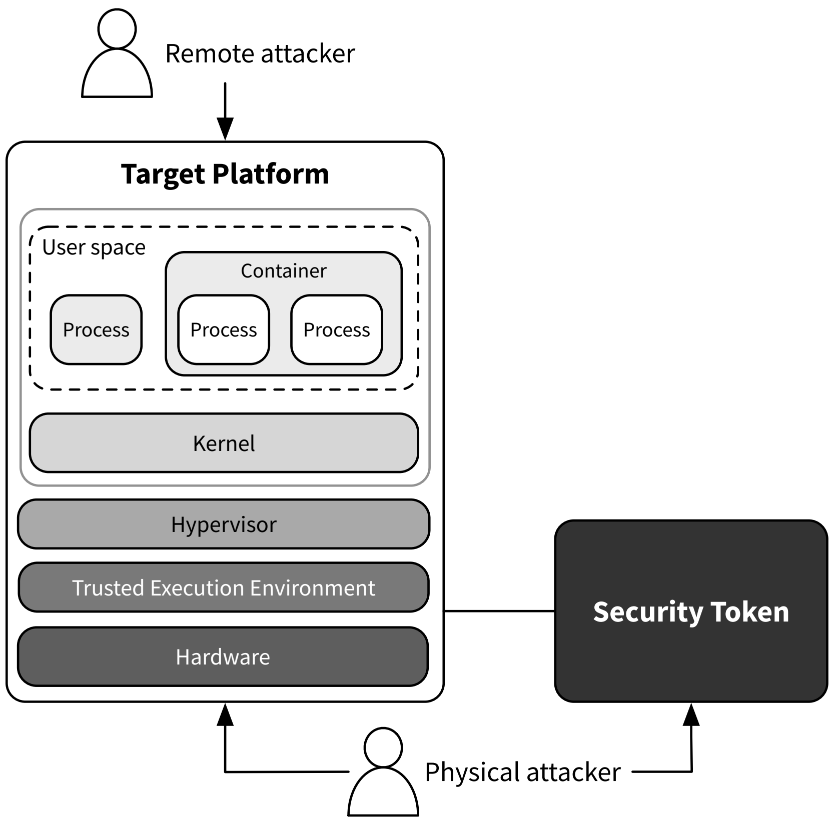


Figure 3.2: Generic attacker model.

Remote attacker. The remote attacker attacks the system remotely, for example, via a network connection. He has *no physical access* to the system's hardware. Since only the target platform offers remote communication connections, in order to gain access to higher trust levels and ultimately the token, the remote attacker typically has to work his way through the target platform's software stack, as depicted in Figure 3.2, starting on the top with an attack on one of its outward-facing user space processes. As described in detail in Section 5.1, typical remote attacks try to achieve RCE, i.e., take over a user process, for instance, by exploiting memory corruption bugs in unsafe languages, such as C and C++. Afterwards, depending on his skill level, the remote attacker might be able to use the controlled process to attack higher privileged, logically separated layers in the target platform. For example, he might be able to exploit further bugs and gain control of privileged software as described in Section 5.1, or extract secrets from bug-free privileged software using microarchitectural attacks, breaking logical separation as described in detail in Section 7.1. The remote attacker has no direct access to the token and has to gain control over the target platform in order to communicate with the token.

Physical attacker. The *physical* attacker has physical access to the system's hardware components. Depending on the specific scenario, he might have access to the target platform, the token, or both. The physical attacker uses his direct hardware access for attacking the system. He typically attacks local physical interfaces, such as debug or serial ports, or directly targets the hardware, for example, via cold boot or bus sniffing attacks, as described in detail in Section 6.1. Using those attack vectors, the physical attacker might be able to take control of the target platform at run-time, for example, with a writing DMA attack or by modifying software on secondary storage and rebooting. If successful, the physical attacker basically gains the same privileges as a remote attacker in the layer it compromises. But typically, our physical attacker conducts attacks that extract a snapshot of the target platform's state at the time of the attack. Our most important physical attacker, as described in detail in Section 6.1, is able to execute *memory attacks* against the target platform, extracting a full dump of its main memory without having control over privileged software on the target platform. The physical attacker has access to the token and may therefore directly communicate with the interface it exposes to the target platform. But, as the remote attacker, he is *not* able to compromise the software running on the token.

Both attackers. Both attackers have full access to infrastructure components surrounding our two main entities, token and target platform. For example, the remote attacker has a privileged position in the network the target platform is connected to. Both attackers are unable to break cryptographic primitives. Furthermore, they are unable to compromise the token's software and especially unable to extract secrets, such as keys, from the token. This assumption is reasonable, since the token is a special-purpose device with a minimalist interface for a defined set of functions, comparable to an SE. As such, it can be hardened effectively. If the token is equipped with a secure unlocking mechanism, as described in Section 3.2, none of the attackers is able to unlock a locked token. If the token is unlocked via the target platform, both attackers might be able to gain access to the unlock secret if they control software on the target platform.

Despite using different initial attack vectors, both attackers can be *equally capable* in some cases and achieve the same attack goals. For example, a physical attacker might execute a writing DMA attack on the target platform's kernel to achieve privileged code execution. A remote attacker might exploit a bug in a syscall implementation to achieve the same goal. As another example, a remote attacker might be able to remotely take over a peripheral device in the target platform, such as a network controller, and extract a main memory dump via DMA. A physical attacker might achieve the same goal using a cold boot attack.

Depending on the specific security concept and its protection goal, a specific attacker model defined in one of the following chapters might include one or both types of attackers with different capability levels. Strongly connected to the capability level of the attacker is the Trusted Computing Base (TCB) we assume for a concept, i.e., the software layers that have to be uncompromised for the mechanism to provide its security guarantees. The more capable an attacker in a specific attacker model, the smaller the TCB we assume for the corresponding concept. The token and its software components are always part of the TCB and are sufficient as TCB for the token-based concepts in Chapter 4 and Chapter 7. Our hypervisor-based monitoring concept in Chapter 5 adds the hypervisor to the TCB. Finally, our memory encryption mechanisms in Chapter 6 additionally require the kernel layer to be part of the TCB.

3.4 Summary

In this chapter, we introduced our system architecture and generic attacker model. Our system consists of two physical entities, the target platform and the security token. For the target platform, we specified a coarse software architecture, defining several layers of logical separation and our corresponding security mechanisms together with their hardware requirements. Since the token may take many physical forms, we omitted the definition of a token software architecture and instead defined functional blocks and corresponding hardware requirements. Defining orthogonal hardware requirements for the different security mechanisms makes our architecture modular and applicable to different platforms, possibly implementing only a subset of features matching the targeted hardware.

For our generic attacker model, we defined two types of attackers with different capabilities, using different attack vectors. The remote attacker does not have access to the hardware and attacks the system via one of its remotely accessible interfaces. The physical attacker has physical access to the system and conducts hardware attacks or attacks the system via one of its local interfaces. Our generic model provides the building blocks for the specific attacker models defined for each security concept in the following chapters.

4 Establishing Trusted Boot-time Integrity

As one main part of this thesis, we want to explore ways to leverage physical separation for key protection. To this end, our system architecture provides the security token as secure, logically separated storage location for the most valuable secrets of the target platform. Apart from protecting the token with a user-known secret, such as a PIN, the token should also be able to protect itself from a target platform whose code integrity is not guaranteed at startup.

Different approaches are imaginable to achieve this goal and let the token gain trust in the boot-time integrity of the target platform, building the basis for the security of our two-part system architecture during its run-time. Several solutions could be constructed with traditional software attestation mechanisms we outlined in Chapter 2. Software attestation always requires key material trusted by the verifier, i.e., the token, and a mechanism to bind the availability of those keys to the attested platform's boot-integrity. This could, for example, be achieved using a secure boot, a TEE and an associated key protection mechanism, as described in Section 2.3.1. Furthermore, a target platform with a (firmware) TPM could implement a measured boot, as described in Section 2.4.2. At run-time, in both cases, an attestation protocol would be used to provide the token with proof about the integrity of the boot process of the target platform. While both approaches rely on different hardware features in the target platform, they share some significant downsides for our use case scenario. First, both require the token's trust in the initial stage of the target platform's boot process, i.e., the *root of trust* or *trust anchor*, typically implemented as ROM code on mobile devices. Second, there is no way to dynamically pair a target platform with a token. The target platform has to be provisioned with pre-shared key material or a private key trusted by the token. Lastly, relying on specific hardware features, both mechanisms cannot be retrofitted to legacy devices or patched if the ROM code has a vulnerability or the attestation key is compromised.

To overcome these downsides, in this chapter, we propose a *software-based trusted boot*, allowing the token to gain trust in the boot-time integrity of the target platform without

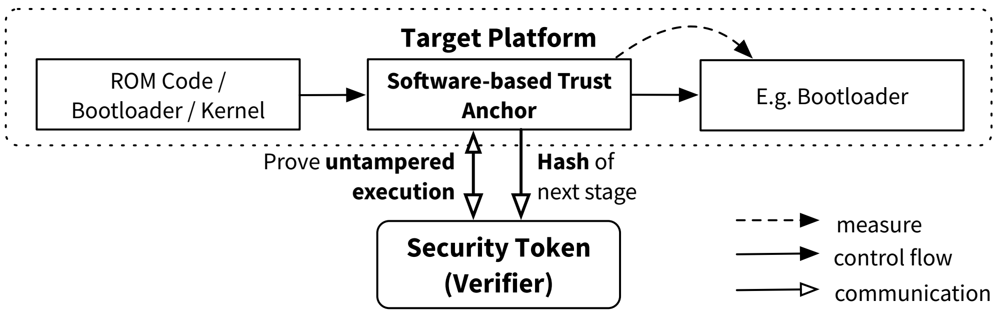

Target Platform
ROM Code / Bootloader / Kernel
Software-based Trust Anchor
E.g. Bootloader
Prove untampered execution
Hash of next stage
Security Token (Verifier)
measure
control flow
communication

Figure 4.1: Software-based trusted boot process.

relying on any pre-shared secrets or pre-existing trust relationship. As core of our new boot process we introduce a *Software-based Trust Anchor (SobTrA)*, a primitive for externally verifiable execution on ARM Cortex-A processors. In a timing-based protocol, SobTrA is able to provide the token with the guarantee that a piece of code runs untampered on the target platform. With SobTrA used as a replacement or supplement for the traditional hardware-based trust anchor, the software-based trusted boot can found a chain of trust on the *initially untrusted* target platform. Externally verifiable execution is strongly dependent on the underlying hardware and while primitives have been proposed for several architectures [KJ03; Ses+04; Ses+05; Ses+06], SobTrA is first of its kind for an ARM Cortex-A CPU, a complex and highly relevant architecture for mobile devices.

The chapter is organized as follows. First, we discuss the basic concept of our software-based trusted boot. Then, we refine our general attacker model from Section 3.3 to the SobTrA scenario and introduce related work providing similar guarantees as SobTrA on other hardware platforms. Afterwards, we introduce the general design of SobTrA and present details of a SobTrA realization on an ARM Cortex-A8 target platform including an actual prototype implementation. Finally, we show the practicability of the approach on the basis of experimental results acquired with our prototype.

Parts of this chapter have been published in [Hor+14b] and are based on results from the author's master thesis [Hor12]. More specifically, the basic SobTrA design, implementation and experimental evaluation are results of the latter.

4.1 Software-based Trusted Boot

The goal of our software-based trusted boot process is to bootstrap the target platform while proving the integrity of the booted components to the token, without requiring traditional hardware features for attestation. Figure 4.1 illustrates the process. The boot process involves both entities of our system architecture, i.e., the *initially untrusted target platform* and the token. In terms of software attestation, the token takes the role of the *verifier* and we use both terms interchangeably in the following.

The target platform starts its boot process normally, i.e., without any additions or modifications, and, at an arbitrary point, starts the SobTrA code and protocol. The basic idea of SobTrA is to let the token measure the time the target platform needs to perform a self-checksumming calculation initialized by a random challenge. If the verifier, i.e., the token, receives the correct result in a time frame that does not exceed some previously determined upper bound, the target platform is verified successfully. Details of this process are explained in the remainder of the chapter.

After successfully finishing SobTrA, the token can be sure that the SobTrA code was executed untampered by the target platform. As final steps of the boot protocol, the target platform calculates and sends a cryptographic hash of the following boot stage to the token, which compares it to the expected value. If the hash is correct, the token allows access to its security functionality, if not, those functions are disabled at least until the next reboot. Alternatively, the token can also require measurements of multiple boot components before unlocking its functions, similar to a measured boot with the token taking the role of a TPM.

In our system architecture, the boot stage following SobTrA in the boot process is running in the highest privilege layer on our target platform. On ARM platforms, this is typically the firmware running in the TrustZone TEE as described in Section 2.4.1. This stage can include a public key as basis for the chain of trust built upon SobTrA. The integrity of the key is guaranteed alongside the integrity of the boot stage itself. Note that the target platform might additionally employ a traditional, hardware-based secure boot, which, orthogonal to our mechanism, enforces that only signed software may be booted on the target platform.

4.2 Attacker Model and Assumptions

In Section 3.3, we defined a remote and a physical attacker depending on his position in the architecture. For SobTrA we consider both physical and remote attackers, who are able to *modify any software which is loaded during boot on the target platform*. The remote attacker might achieve this, for example, by exploiting a user space process in the target, privilege escalation and modifying the kernel image. A physical attacker, on the other hand, could, for example, try to directly modify the secondary storage of the target platform. The attacker is not able to attack the token itself, especially not the cryptographic secrets stored on it. The attacker is not able to modify memory of the target platform without involving the main CPU, e.g., via DMA or a Joint Test Action Group (JTAG) debug interface. Note that such an attacker would also be able to circumvent a traditional measured boot. We assume that the attacker is *unable* to modify the physical connection between token and target platform or the hardware of the target platform itself. From this basic assumption, three specific assumptions follow:

- The exact hardware configuration of the initially untrusted target platform is known to the verifier, including CPU type (e.g., ARM Cortex-A8) and clock speed.

- The verifier receives messages directly from the target platform, excluding MitM attacks.
- No proxy or relay attack is possible. These are attacks where the target platform uses the help of another connected device to fulfill the challenge of the verifier.

Especially in the context of tightly integrated embedded target platform, such as smartphones, these assumptions can be considered reasonable.

4.3 Related Work

In its core, SobTrA is a primitive for so-called *externally verifiable untampered execution*. Such a primitive mainly consists of a function that calculates a checksum based on a challenge by the verifier, its own memory layout, and different platform-specific inputs. The verifier measures the time the target requires for calculating the correct result and assumes that the code was executed untampered if a pre-determined time threshold is not exceeded.

As one of the first approaches towards externally verifiable execution, Kennel and Jamieson proposed *Genuinity* [KJ03], a *self-checksumming* function for x86 platforms, calculating a checksum over its own code and a challenge from the verifier. Additionally to its own code and the challenge, *Genuinity* uses hardware performance counters as input to the checksum calculation, making the result dependent on the specific x86 platform used and impeding the reproduction of the checksum on other platforms. Unfortunately, the instructions used for accessing performance counters are slow, making the approach vulnerable to *substitution attacks* [SCT04], in which an attacker replaces parts of the checksum function with his own code (see Section 4.4.3).

The authors of *SWATT* [Ses+04], an approach targeting an Atmel 8-bit microcontroller, carefully design their checksum function around a single tight loop, minimizing the execution time of single loop iterations. Instead of performance counters, the *SWATT* checksum function uses fast-accessible CPU state, such as the *program counter* and *status register*, as additional input. Similar to microbenchmarking scenarios, the design amplifies absolute execution time overhead introduced by attacker's code inside the main loop with an increasing number of loop iterations. Another advantage of *SWATT* is that the execution time of its checksum function does not vary with varying challenges but only with the number of loop iterations, considerably improving the reliability and usability of the verification process. A checksum function following the *SWATT* design must be designed specifically for the targeted hardware architecture. *Pioneer* [Ses+05; Ses09] and *ICE* [Ses+06] refine and adapt the *SWATT* design and implementation to x86 and TI MSP430 hardware platforms, respectively.

Later publications introduce further concepts to implement checksum functions with similar properties. Either by generating the checksum function itself as the challenge

[MPB10] or by introducing a memory bottleneck during the checksum calculation to slow down the attacker's code [GGR09; JJ11]. As a downside, the first approach shows a higher variance for the checksum function execution time than the *SWATT* design. Downsides of the latter approach are that it takes more time to execute and is harder to be used during system run-time, requiring massive memory swapping.

As first primitive for externally verifiable execution on an ARM Cortex-A platform, SobTrA adapts and refines the tight loop checksum function design of the *SWATT*-like approaches [Ses+04; Ses+05; Ses+06; Ses09] to the ARMv7-A architecture, resulting in substantial changes to the underlying algorithm design and implementation.

4.4 Software-based Trust Anchor

The core of the proposed software-based trusted boot, as shown in Figure 4.1, is the software-based trust anchor SobTrA. In the following, we cover basic conceptual aspects of SobTrA which apply to all ARMv7-A targets before we introduce an ARM Cortex-A8-specific realization.First, we discuss the different components of SobTrA and how SobTrA can be integrated in our general system architecture introduced in Chapter 3. Then, we discuss the SobTrA software architecture and protocol, general checksum function design aspects and common attack types.

4.4.1 Hardware Architecture

General aspects of our system architecture have already been discussed in Chapter 3. For SobTrA we further specify the actual hardware properties of the architecture components as follows. SobTrA in its current form is a primitive whose concepts are specific to ARM Cortex-A processors using the ARMv7-A architecture (Section 2.1). Therefore, a specific system architecture that uses SobTrA employs an ARMv7-A target platform as the initially untrusted device. Furthermore, our SobTrA checksum function implementation is specific to the ARMv7-A Cortex-A8 SoC. SobTrA does not pose any requirements regarding the architecture of the verifier, i.e., the token in our architecture, but it must provide a component to measure time independently of the target platform. An optional feature for the token is the addition of some kind of unlocking mechanism as described in Section 3.2. The unlocking mechanism can be used to support the assumption from our attacker model in Section 4.2 that the hardware is unmodified. In such a scenario only the legitimate user is able to unlock the token and does so only if there are no visible hardware modifications.

There are no specific requirements regarding the connection between token and target platform for SobTrA except that its latency should be as low as possible. The connection can be temporary, for example, via USB or SDIO, or fixed, e.g., via SPI. As discussed in Section 4.7, our prototype uses an SPI connection.

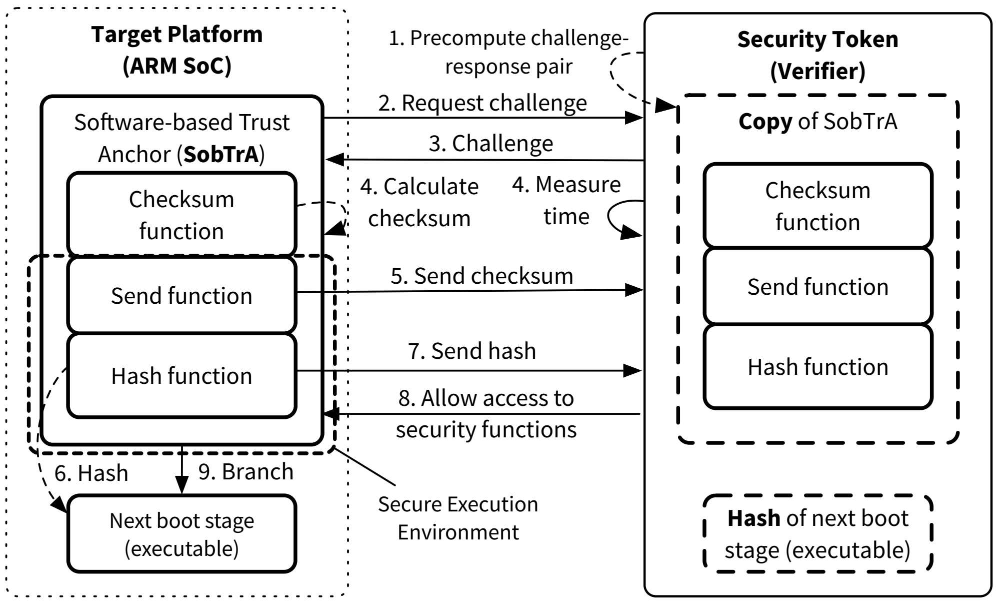

Target Platform (ARM SoC)
Software-based Trust Anchor (SobTrA)
Checksum function
Send function
Hash function
6. Hash
9. Branch
Next boot stage (executable)
1. Precompute challenge-response pair
2. Request challenge
3. Challenge
4. Calculate checksum
4. Measure time
5. Send checksum
7. Send hash
8. Allow access to security functions
Secure Execution Environment
Security Token (Verifier)
Copy of SobTrA
Checksum function
Send function
Hash function
Hash of next boot stage (executable)

Figure 4.2: Architecture and protocol of SobTrA and the software-based trusted boot process.

4.4.2 Software Architecture and Protocol

An overview of the SobTrA architecture and the steps of the verification and trusted boot protocol are depicted in Figure 4.2. SobTrA runs on the target platform and consists of three main parts, which are the checksum, send and hash functions. The *checksum function* is the core of SobTrA and is mainly responsible for calculating a checksum over the code itself based on a challenge by the verifier. The *send function* sends the resulting checksum to the verifier. The *hash function* calculates a hash digest over the next boot stage.

The following steps are executed during the execution of SobTrA. In the end of the process, the verifier, i.e., the token, has the guarantee that the target platform continues its boot process with untampered software components. In the first communication step, SobTrA requests a challenge (2.). The verifier either pre-computes (1.) challenge-response pairs to accelerate the protocol or computes one on demand directly before sending the challenge to the target platform (3.). SobTrA uses the challenge to initialize the checksum function, which calculates a checksum (4.) over itself and the other two main parts of SobTrA while the verifier measures the time. Afterwards, SobTrA initializes an *uninterruptible execution environment* for the rest of the trust anchor to run in, preventing an attacker from gaining control via exceptions (see Section 4.6). Then, the *send function* sends the resulting checksum back to the verifier (5.). The verifier stops its time measurement as soon as it receives the checksum. If the time is below a certain threshold and the checksum is correct, the verifier can be certain that SobTrA runs untampered on the target platform. Meanwhile, the hash function in SobTrA computes (6.) a hash digest of the next

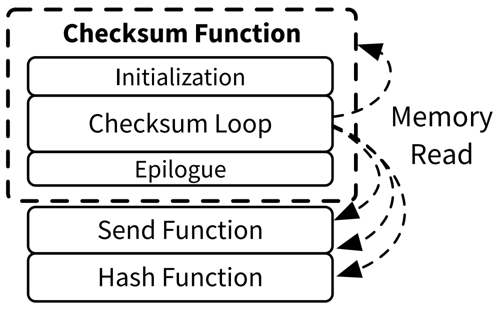


Figure 4.3: Structural overview of the SobTrA checksum function.

stage in the boot process. The steps that follow after this point depend on the specific application scenario. In our scenario, the verifier in form of the token wants to ensure that the boot process on the target platform is untampered before allowing access to its security functionality mainly detailed in Chapter 7. Hence, the target platform sends the hash digest to the verifier (7.) who checks if the hash is correct and only allows access to functionality (8.) if the verification is successful as described before and the hash is correct. The unlocked functionality could, for example, be the access to keys used to protect the target platform's file system as described in Section 7.2. All parts of SobTrA must be self-contained, i.e., they must not call code outside the checksummed region and must not cause exceptions.

The verifier contains a copy of the SobTrA binary image deployed on the target platform. It is therefore able to calculate the correct checksum to validate the checksum received from the target platform. The number of loop iterations, sent by the verifier alongside the challenge, determines the maximal time allowed for the target platform to return the correct checksum. This maximum time must be pre-measured *once* on a trusted device identical to the target platform and is valid for *all* challenges with the same iteration number.

4.4.3 Checksum Function Design

The core of SobTrA is the so-called checksum function, which calculates the checksum based on a challenge and an iteration count. The checksum function must show the following *basic property*: A *tampered* checksum function must produce a *wrong checksum* or execute with a *measurable time overhead*. Figure 4.3 shows the basic structure of the checksum function in the context of the other SobTrA parts. The first step towards the basic property is to make the checksum function *self-checksumming*, incorporating its own code and all other parts of SobTrA into the checksum. With this, changes to the SobTrA code are reflected in the checksum, forcing an attacker to actively hide code manipulations from the self-checksumming. This, in turn, typically increases computation complexity, thus, generating a measurable execution time overhead. The checksum function mainly

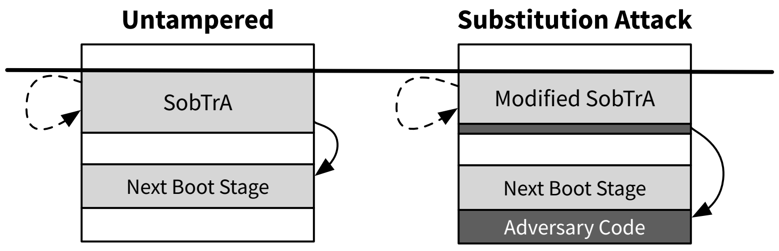


Figure 4.4: Substitution attack example: The final branch is substituted to redirect the control flow to adversary code.

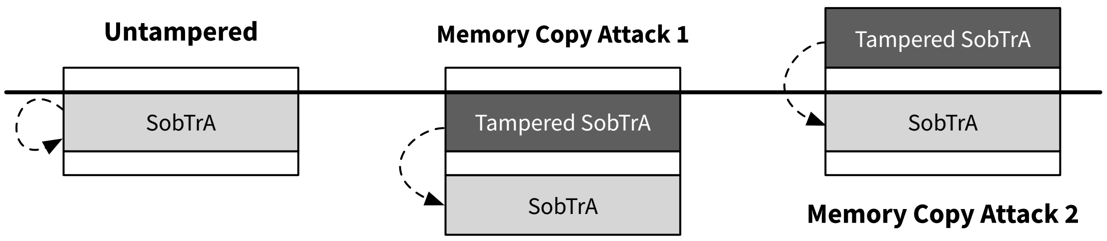


Figure 4.5: Two types of memory copy attacks.

consists of one tight *main loop*, in which a word from memory is read, transformed and included into the checksum together with CPU state input. In order to circumvent this self-checksumming mechanism, an attacker has to add instructions inside the main loop, a manipulation that is measurable when choosing an appropriate number of loop iterations. The epilogue code is responsible for initializing an uninterruptible execution environment for the rest of the trust anchor to securely execute in, as discussed in Section 4.6.

Common Attack Types. Attacks against the checksum function can be grouped into two basic types: Substitution attacks and memory copy attacks. In a *substitution attack*, the attacker modifies parts of the trust anchor's code, for example, the final branch instruction of the epilogue as visualized in Figure 4.4. He then tries to circumvent the self-checksumming mechanism specifically for the manipulated instructions. In a *memory copy attack*, the attacker maintains an untampered copy of the trust anchor and, while executing his manipulated trust anchor code, redirects all self-checksumming memory reads to this untampered image. Depending on the memory location of the copies in relation to the originally intended trust anchor location, we can differentiate between two basic forms of memory copy attacks, as illustrated in Figure 4.5.

Design Goals. Seshadri et al. [Ses+04; Ses+05] describe properties of checksum functions showing the aforementioned basic property of either producing a wrong checksum or causing a measurable overhead when maliciously modified. Based on these, we define the following *design goals* for the SobTrA checksum function:

1. The checksummed memory must be *traversed in a pseudorandom manner* to prevent substitution attacks.

2. The checksum function must be *initialized by a random challenge* to prevent pre-computation.

3. The checksum function must include *CPU state inputs*, e.g., PC and Status Register (SR), into the checksum, mainly to prevent memory copy attacks.

4. To slow down code added by an adversary (register spilling), the checksum function must *use all available CPU registers*.

5. The checksum function should be *strongly ordered and non-parallelizable* to prevent addition of malicious instructions without overhead.

6. The body of the checksum function's main loop should be *as small as possible* to allow the measurement of small attacker overheads and to facilitate code optimization.

7. The checksum function implementation should be *time optimal*. Since optimality is hard to guarantee, this property should be approximated by carefully designing the critical main loop on machine code level. In the future, it might be possible to ensure optimality using tools like Denali [JNR02].

8. The checksum function execution time should have a *low variance*, only depending on the number of loop iterations and not on the challenge.

In the following, we describe a checksum function for ARM Cortex-A8 processors which we designed based on these goals.

4.5 Cortex-A8 Checksum Function

A checksum function implementation *must* be designed very specifically for a processor architecture to provide the introduced properties. The SobTrA checksum function is constructed specifically for ARMv7-A Cortex-A8 processors. Nonetheless, the checksum function design can provide a basis for an implementation on other ARM cores, especially ones with a similar pipeline design, e.g., other in-order pipeline processors, such as the newer Cortex-A53. SobTrA currently only supports single-core processors but could be combined with the generic approach by Yan et al. [Yan+11] to extend its functionality to multi-core processors.

4.5.1 Hardware Characteristics

We discussed details of the ARM architecture in general and the ARMv7-A architecture version in Section 2.1. The ARMv7-A architecture has some special characteristics which must be considered in the checksum function design:

Special purpose registers. The Program Counter (PC) is represented as an explicit register while the Status Register (SR) is *not* accessible like a normal register and flags must be set explicitly using the flag-setting versions of data processing instructions.

Load/Store architecture. Memory is accessed *only* via dedicated load/store instructions.

Free rotation/shift. Many instructions allow the rotation/shifting of the second operand without an additional instruction (and without cost at least on Cortex-A8).

Two instruction Sets. There are two different instruction sets with different instruction encodings: ARM and Thumb-2. The latter mixes 16 and 32 bit wide instructions for higher code density.

Different processor modes. An ARMv7-A processor provides different processor modes most of which are dedicated modes for handling the different exceptions that can occur in the system. The mode the CPU is running in is encoded in the CPSR.

4.5.2 Hardware Initialization

It is crucial that the hardware is configured to provide the maximum possible performance for running the checksum function. Every unused optimization could potentially be abused by an attacker to hide execution time overhead introduced by his code. Therefore, the CPU must be configured to run with the highest clock speed configurable in software, and branch prediction and all caches must be activated. Latter requires the MMU to be *enabled* [ARM14].

Besides performance initializations, it is also necessary to temporarily disable all *interrupts* and run in the most privileged processor mode available to prevent an attacker from gaining control during or shortly after the checksum function. This is discussed in detail in Section 4.6. In our overall system architecture (see Chapter 3) this means that the checksum function on the target platform runs in Monitor mode, which normally controls switches between TrustZone secure world and normal world, as described in Section 2.3.1. If the Monitor mode is not available, for example, because it is controlled exclusively by the manufacturer of the target platform, we assume that it is also not available to a possible attacker. In this case, the checksum function can also be executed in another privileged mode such as Supervisor Mode with the same security against the described attacker.

Listing 4.1 SobTrA checksum function algorithm.

```
 1: Input:
        $y$: number of iterations of the function
        $challenge$: value initializing $C$, $rand$ and $r13$
 2: Output: Checksum $C$
 3: Variables:
        [$start\_code$, $end\_code$]: checksummed memory area
        $daddr$: address of current memory access
        $rand$: current pseudorandom number
        $r13$: intermediate result from previous iteration
        $t$: variable to save parallel processing results
        $j$: index of current checksum part
        $l$: loop counter
        $SR$: status (flags) register
        $PC$: program counter
 4: for $l = y$ to 0 do
 5:     $rand \longleftarrow rand + (rand^2 \vee 5) \bmod 2^{32}$        ▹ T-Function [KS04] updates $rand$
 6:     $daddr \longleftarrow ((C_{j-1} \oplus rand) \wedge MASK) + start$        ▹ Update memory address with $rand$
 7:     $C_j \longleftarrow C_j + PC$        ▹ Begin: Update checksum part with index $j$
 8:     $t \longleftarrow \text{mem}[daddr]$
 9:     $C_j \longleftarrow t \oplus \text{rotate}(C_j)$
10:     $t \longleftarrow t + r13$
11:     $r13 \longleftarrow r13 + C_j$
12:     $t \longleftarrow t \oplus l + C_{j-1} \oplus rand + daddr \oplus C_{j-2}$
13:     $C_j \longleftarrow C_j + PC \oplus l +_c C_{j-1} \oplus rand +_c daddr \oplus C_{j-2} +_c t$
14:     $C_j \longleftarrow SR \oplus \text{rotate}(C_j)$        ▹ End: Update checksum part with index $j$
15:     $j \longleftarrow (j + 1) \bmod 10$        ▹ Update index $j$
16: end for
```

4.5.3 Basic Algorithm

In the following, we discuss the basic design of the SobTrA checksum function, before introducing details of its implementation. Listing 4.1 shows pseudocode for the checksum function main loop. The loop body can be structured into three parts:

1. **Pseudorandom Update.** Calculation of a pseudorandom number used for address generation and for updating the checksum (Line 5).

2. **Data Address Update.** Calculation of the address of the next memory word to be incorporated into the checksum based on the pseudorandom number from the previous step (Line 6).

3. **Checksum Update.** Update of the current checksum part (Lines 7 to 14). The checksum part is combined with a memory value read from the SobTrA code, other checksum parts, the current pseudorandom value, an intermediate result from the previous iteration (in `r13`), the SR, and the PC.

These three parts form a *basic block* or simply *block* of the checksum function. The checksum update sequence is highly architecture-specific. Hence, some of the design ideas will be explained later together with the actual implementation. The first five of our design goals defined in Section 4.4.3 are obviously fulfilled right away by the algorithm. The pseudorandom memory traversal by the algorithm (design goal 1) ensures that an attacker cannot predict when his code modification will be checksummed, forcing him to introduce, for example, an `if`-like statement into the main loop for a substitution attack. The function is initialized by a challenge (design goal 2), prohibiting an attacker from computing a checksum ahead of time. Including the PC (design goal 3) into the checksum with instructions that, because of their PC-usage, cannot be reordered or repositioned, makes the result of the function dependent on the memory location it executes from and, hence, prevents memory copy attacks. Incorporating the SR (design goal 3) ensures that the processor mode the checksum is calculated in is reflected in the checksum, as required for protection against malicious exceptions (see Section 4.6). In order to deprive an attacker from usable registers for his code, the checksum is split into several parts C_j, filling all available registers (design goal 4), and each iteration of the algorithm loop updates one part C_j of the checksum. By making blocks dependent on previous results, creating interdependencies between subparts of the loop body and interleaving add (+), add-with-carry ($+_c$) and xor ($\oplus$) operations for updating the checksum parts, it is ensured that the function is strongly ordered and non-parallelizable (design goal 5).

Design goals 6 and 7, namely the minimization and optimization of the checksum function, are fulfilled by carefully designing the main loop body on machine code level, as shown in the next section. The experimental evaluation of our prototype implementation in Section 4.8 confirms that the checksum function's execution time has a very low variance (design goal 8), enabling robust SobTrA protocol runs and the measurement of very small overheads introduced by attacks.

4.5.4 Main Loop Implementation

In the following, we first introduce some general aspects of our checksum function implementation for ARM Cortex-A8 before discussing details of the actual implementation of the three structural parts of the checksum function shown in Listings 4.2 to 4.4. As discussed in detail later in Section 4.6, the checksum function implementation should run in the most privileged CPU mode available on the target platform.

Our assembler code uses named registers, as specified in Figure 4.6, to increase readability. There is one dedicated register for each running variable in the pseudocode algorithm (Listing 4.1) and an additional scratch register named `rs` to hold different temporary values, especially the t variable, throughout a loop iteration. Ten registers (`chk0` to `chk9`) store the checksum parts resulting in a 320 bit wide checksum. To make most efficient use of available registers, the main loop in the pseudocode algorithm is *unrolled* into ten checksum part-specific blocks in the actual implementation. Hence, the index j in the

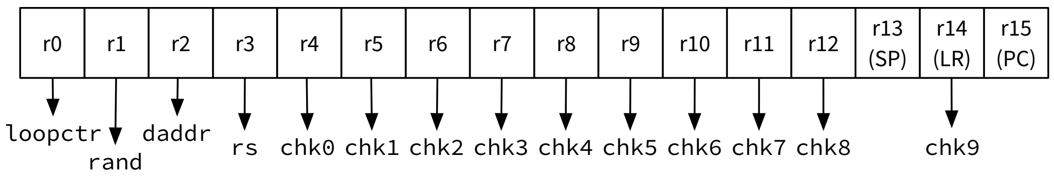


Figure 4.6: Register allocation in the SobTrA checksum function.

pseudocode has no counterpart in our implementation. The SobTrA checksum function implementation *must* be encoded using the Thumb-2 instruction set because the checksum function leverages the mixed instruction size (16 and 32 bits) to prevent a specific form of memory copy attack, as discussed in [Hor+14b]. In Thumb-2 encoding, the registers `r13` (Stack Pointer) and `r15` (PC) are *only usable* in some instructions. Register `r13` can therefore *not* be used as a checksum register or working register. To fulfill the property of using all registers, we occupy `r13` with an intermediate result to be included into the checksum in the following block, as already shown in the algorithm pseudocode in Listing 4.1.

To reduce the possibilities for an attacker to insert instructions without overhead, we manually analyzed the cycle-exact execution of the checksum function implementation for the Cortex-A8 dual-issue, in-order pipeline using information provided by [ARM10]. With our analysis, whose results we verified in practical tests on our prototype using the ARM architectural performance counters, we made sure that our implementation makes highly efficient use of the pipeline features. The most important optimization is based on the dual-issue feature of the pipeline. This feature allows the parallel execution of two consecutive instructions but *not* if both instructions write to the same target register. Hence, for a naïve implementation consecutively updating the current checksum register, the dual-issue rate would be very low. To increase this rate and make the dual-issue feature unavailable to a possible attacker, our implementation uses a second value *t* stored in `rs`, which is updated in parallel to the checksum register using dual-issued instructions and combined with the it at the end of the block. Our implementation of a basic block consists of 28 instructions of which only six are not dual-issued. Furthermore, the dual-issue rate cannot be increased by re-ordering the instructions. A basic block executes in only 31 Cortex-A8 processor cycles. In the following, we discuss further implementation details of the initialization and the three checksum function parts.

Initialization. The checksum function is initialized with a 320 bit wide challenge filling all checksum registers with a starting value. The seed for the random number generator is derived from the challenge by xoring their parts. Besides that, also `r13` is initialized with a value derived from `rand`.

Listing 4.2 SobTrA pseudorandom update implementation.

```
mul rs, rand, rand        @ $x^2$
orr rs, rs, #0x5          @ $x^2 \vee 5$
add rand, rs, rand        @ $(x^2 \vee 5) + x$
```

Listing 4.3 SobTrA data address update implementation (first block).

```
eors daddr, chk9, rand         @ derive random value
ldr  rs, =0x7ff<<2             @ load mask in scratch register
and  daddr, daddr, rs          @ mask random to get an offset
adr  rs, start_code            @ gen. start address PC-relative
add  daddr, daddr, rs          @ add offset to start address
```

Pseudorandom Update. Like other approaches [Ses+04; MPB10; Ses+05], SobTrA uses a T-Function [KS04] for the pseudorandom update (Listing 4.1, Line 5), mainly because it can be implemented with only few instructions as shown in Listing 4.2. Therefore, an optimal implementation can be ensured more easily.

Data Address Update. The SobTrA data address update implementation for the first checksum function block is shown in Listing 4.3. For optimal performance, only word-aligned addresses are generated. The mask used for extracting offset bits from the pseudorandom value determines that eight KiB starting from the `start_code` label are inside the self-checksumming mechanism's reach. While this is enough to cover all SobTrA parts in our prototype, the checksummed region can be easily increased by modifying the mask value, for example, to accommodate a larger hash or send function (see Section 4.4.2).

Checksum Update. Listing 4.4 shows the SobTrA implementation of the checksum update sequence for the first checksum part register. To keep the scratch register `rs`, used for loading the memory value (Line 10), alive and unavailable to an attacker, SobTrA processes it with a sequence of operations parallel to the checksum register and incorporates it into the checksum right before it is needed to load the status register CPSR. As discussed before, these operations fill the Cortex-A8 dual-issue pipeline optimally and therefore cause almost no overhead. Some operations are used in their flag-setting variants (mnemonic suffixed with "s") to impede the forgery of the CPSR. This is particularly important for the addition immediately before the `mrs` instruction, which reads the CPSR, since it prevents reordering of latter instruction to an upper position. The flags in the CPSR set by the flag-setting instructions prevent an attacker from removing the relatively expensive `mrs` instruction, costing about eight CPU cycles on Cortex-A8 [ARM10]. As discussed before, the SP register `r13` is not usable in all instructions. Hence, our SobTrA implementation occupies `r13` with an intermediate checksum result that is combined with the following checksum part in the next block (Lines 12 and 13).

Listing 4.4 SobTrA checksum update implementation (first block).

```
 9 add  chk0, chk0, pc           @ $C_0 \leftarrow C_0 + PC$
10 ldr  rs, [daddr]              @ $t \leftarrow \text{mem}[daddr]$
11 eor  chk0, rs, chk0, ror #1   @ $C_0 \leftarrow t \oplus \text{rotate}(C_0)$
12 add  rs, rs, r13              @ $t \leftarrow t + r13$
13 add  r13, chk0                @ $r13 \leftarrow r13 + C_0$
14 add  chk0, chk0, pc           @ $C_0 \leftarrow C_0 + PC$
15 eors rs, rs, loopctr          @ $t \leftarrow t \oplus l$
16 eors chk0, chk0, loopctr      @ $C_0 \leftarrow C_0 \oplus l$
17 add  rs, rs, chk9             @ $t \leftarrow t + C_9$
18 adcs chk0, chk0, chk9         @ $C_0 \leftarrow C_0 +_c C_9$
19 eors rs, rs, rand             @ $t \leftarrow t \oplus rand$
20 eors chk0, chk0, rand         @ $C_0 \leftarrow C_0 \oplus rand$
21 add  rs, rs, daddr            @ $t \leftarrow t + daddr$
22 adcs chk0, chk0, daddr        @ $C_0 \leftarrow C_0 +_c daddr$
23 eors rs, rs, chk8             @ $t \leftarrow t \oplus C_8$
24 eors chk0, chk0, chk8         @ $C_0 \leftarrow C_0 \oplus C_8$
25 adcs chk0, chk0, rs           @ $C_0 \leftarrow C_0 +_c t$
26 mrs  rs, cpsr                 @ load the $SR$
27 eor  chk0, rs, chk0, ror #1   @ $C_0 \leftarrow SR \oplus \text{rotate}(C_0)$
28 sub  loopctr, loopctr, #1     @ decrement $l$
```

Further details of our checksum function implementation are discussed in [Hor+14b]. There, we show that memory copy and substitution attacks, as introduced in Section 4.4.3, in basic and advanced forms produce at least *one CPU cycle overhead* per SobTrA checksum function block. Among others, we explain in detail how the use of the PC register (Lines 9 and 14) hardens the checksum function against memory copy attacks and how specific substitution attacks [Cas+09] are prevented with add-with-carry (`adcs`) operations (as suggested in [PD10]) and rotations (Line 11 and 27). Furthermore, we discuss the desynchronization of Harvard-style TLBs [WOS05] for an advanced memory copy attack and the corresponding SobTrA defense using self-modifying code, as originally proposed by Giffin et al. [GCK05].

By ensuring the checksum function's resistance to substitution and memory copy attcks, we ensure that an adversary cannot attack the function directly, i.e., from within the same execution context. In the next section, we discuss how SobTrA prevents attacks from *other* execution contexts, triggered through exceptions.

4.6 Exception Protection

SobTrA must ensure that an attacker cannot gain control after a successful checksum calculation. Otherwise, an attacker could use the valid checksum for a successful verification and run arbitrary code afterwards. While SobTrA ensures that only checksummed or hashed code runs following the normal execution flow within the same execution context, an attacker might try to gain control by triggering an exception. In the following, we shortly discuss relevant aspects of the ARM exception model supplementing the general information given in Section 2.1.2. Then, we discuss how SobTrA protects itself against exceptions after (Section 4.6.2) and during (Section 4.6.3) calculation of the checksum. While the checksum function presented in the previous section is mostly designed specifically for Cortex-A8 processors, the concepts for exception prevention are applicable to all ARMv7-A processors and, in parts, also to ARMv8-A processors.

4.6.1 ARM Exception Handling

As discussed in Section 2.1.2, ARM application processors always contain multiple execution modes with different privilege levels. While ARMv7-A defines several Privilege Levels (PLs) and modes that run in those PLs, ARMv8-A only defines Exception Levels (ELs), which are privilege level and mode at the same time. The mode an exception is *taken from* is the mode in which the processor runs while an exception occurs. The mode an exception is *taken to* is the mode the processor switches into to handle the exception. There are some rules for mode switches when taking exceptions [ARM14; ARM17] that are important for SobTrA:

- Exceptions can only be *taken to* a higher or equal privilege level.
- Exceptions can only *return to* a lower or equal privilege level.
- Privilege levels are defined independently in each security state, e.g., an exception in non-secure PL0/EL0 is never taken to secure PL1/EL1 and vice versa. Since Monitor/EL3 mode is *always* secure, this also implies that an exception can never be taken from a secure to a non-secure mode.
- (ARMv7-A) Exceptions can be taken from any non-secure mode to Monitor mode, which is the only way to enter the secure world.
- (ARMv7-A) A switch into non-secure state can be executed from all secure PL1 modes.

ARMv7-A requires additional rules for defining the transition between secure world and non-secure world since PLs in secure world only go up to PL1. Monitor mode shares PL1 with the other secure PL1 modes. This means that, following the first rule, exceptions can be taken from Monitor mode to other secure PL1 modes. The normal way to exit secure state is by an exception return from Monitor mode but ARMv7-A additionally offers the deprecated possibility to execute the switch from any secure PL1 mode as expressed in the last rule.

ARMv8-A defines a cleaner exception and privilege model as it introduces a separate privilege level EL3 as a replacement for Monitor mode while still providing secure EL1 for the secure OS. Therefore, the first three rules are sufficient to implicate that a switch into secure state on ARMv8-A is only possible via an exception to EL3 and a switch from secure state back into non-secure state is only possible via an exception return from EL3.

When taking an exception, several actions are automatically executed by the processor. The steps that are important for SobTrA are roughly summarized in the following without a specific order:

- The CPSR of the current mode is saved in the Saved Program Status Register (SPSR) of the target mode.
- The preferred return address is saved to the Link Register (LR) of the target mode.
- The CPSR is changed to reflect the new mode, i.e., among others, the mode bits are modified and asynchronous exceptions are masked.
- The PC is changed to the address specified in the appropriate entry of the exception vector table of the target mode.

After completing these steps, execution is continued in target mode. Both, ARMv7-A and ARMv8-A, use four exception vector tables. One for non-secure PL1/EL1, one for PL2/EL2, one for Monitor/EL3 and one for secure PL1/EL1.

4.6.2 Uninterruptible Execution Environment

It is guaranteed that only checksummed and unmodified code is executed after a successful checksum function run. Hence, an exception is the only way remaining for an attacker to possibly gain control of the target platform's execution. In the previous section, we discussed the handling of exceptions in ARM application processors. In the following, we use this knowledge to develop a concept for an *uninterruptible execution environment* protecting execution of SobTrA against malicious exceptions *after completing the checksum function*. Figure 4.7 visualizes the parts of SobTrA executed protected by the uninterruptible execution environment.

As explained before, SobTrA must always run in the most privileged mode available for programming in the target platform. This is also the most privileged mode we assume an attacker can run code in. If an attacker is able to execute code in a higher privilege level, he will probably be able to trigger an exception at some point after verification in order to take control of the target's execution. In a typical target platform in our scenario, the mode SobTrA executes in is Monitor mode, since it has the highest privilege level in the system. But in a system where the manufacturer exclusively controls the secure state, SobTrA can also be run securely from non-secure PL2 or PL1, assuming that the manufacturer is not the attacker.

The mode in which SobTrA executes must be known to the verifier and can be verified as it is incorporated into the checksum via the CPSR. This means that executing the checksum

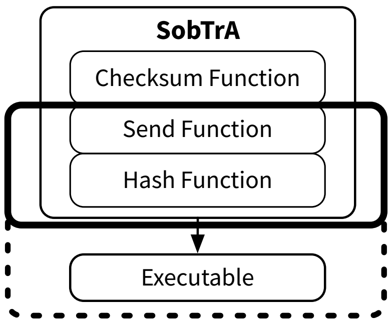


Figure 4.7: Uninterruptible execution environment in the context of the SobTrA structure.

function in a different mode yields a different checksum. In secure state, modes from PL0 and PL1 are mirrored from the non-secure modes in the same privilege levels. They are indistinguishable from their non-secure counterparts only by looking at the CPSR. The security state of the system is reflected by the Non-Secure bit in the Secure Configuration Register (SCR), which can only be written in Monitor/EL3 mode. For example, the verifier cannot distinguish if a checksum was calculated in secure or in non-secure Supervisor mode. Monitor mode, on the other hand, as the only always-secure mode, is uniquely identifiable using the CPSR. Hence, in a system in which all privilege levels are available, SobTrA must run from Monitor mode. Note that, as mentioned before, in a system where secure state is not available for programming we assume that it is the case also for an attacker. Therefore, SobTrA can securely run from the highest available non-secure PL.

The basic idea of the uninterruptible environment is as follows. Immediately *after* calculating the checksum and *before* sending the result, SobTrA establishes an uninterruptible execution environment by replacing all exception vector tables in its privilege level with own tables. In our typical scenario, where SobTrA runs in Monitor mode, this means that both exception vector tables belonging to secure PL1, i.e., the Monitor table and the secure OS table, must be replaced by SobTrA. It is important that those tables and all the code they point to are *inside* the region checksummed by the checksum function. Code inside the checksummed region must not call unmeasured or unchecksummed code. In the simplest case, the SobTrA exception vector tables contain dead loops. In this case, code inside the checksummed region must additionally not cause any exceptions.

As discussed in the previous section, exceptions can only be taken to the same or a higher PL. This ensures that an attacker controlling a lower PL cannot trigger an exception and hijack execution. With SobTrA verifiably running in the highest PL available to a possible attacker, it is furthermore ensured that an attacker cannot gain control by triggering an exception from a more privileged mode.

Summarizing, the uninterruptible execution environment ensures that after sending the checksum, the SobTrA code runs uninterrupted and can safely initialize a following boot stage. The following boot stage will either be a secure world firmware, a hypervisor or a kernel, taking control of the corresponding exception vector tables without offering an opening to attackers to hijack execution.

4.6.3 Protection during Checksum Calculation

We must also ensure that an attacker cannot use exceptions to gain control during and shortly after the checksum function execution, i.e., *after* the checksum calculation but *before* the uninterruptible execution environment is established. Before we discuss each exception type and why SobTrA can be considered secure against it, we first discuss three *properties* of SobTrA that make exploiting exceptions during checksum calculation generally difficult:

1. **Privileged execution.** As discussed before, SobTrA runs in the most privileged mode available. This can be ensured by the verifier as it influences the result of the checksum calculation via the CPSR. With the same arguments as before, we can exclude attacks using exceptions from lower or higher PLs. Consequently, an attacker could, at most, try to exploit an exception into the PL SobTrA itself runs in.

2. **Use of LR.** When handling an exception, the processor automatically writes the preferred return address into a link register of the target mode. Since SobTrA uses LR for storing part of the checksum, this can, depending on the PL, lead to a corruption of the checksum.

3. **Unpredictability.** An attacker might not be able to control when exactly the exception is triggered, for example, due to SobTrA's pseudorandom memory traversal. An exception triggered during checksum function calculation is of no particular use to an attacker who might therefore have to trigger the exception multiple times. This, in turn, quickly introduces a measurable overhead to the calculation.

If SobTrA runs in a mode, such as non-secure Supervisor mode, that uses the same LR as SobTrA for exception return addresses and handles all its exceptions by itself, Property 1 and 2 ensure security against exceptions during checksum calculation.

In our typical scenario, where SobTrA runs in Monitor mode, i.e., secure PL1, Property 2 only offers partial protection. Monitor mode provides the LR and SobTrA uses it for storing a part of the checksum so that the checksum is corrupted as soon as an attacker tries to trigger an exception into Monitor mode. Nonetheless, Monitor mode shares the secure PL1 with the privileged modes meant for running a secure OS. Therefore, exceptions might be triggered that are within the PL but are handled in one of the other secure PL1 modes, e.g., a prefetch abort.

In order to ensure the secure calculation of the checksum even in such cases where SobTrA runs in a mode for which Property 2 does not provide security, i.e., exceptions might be triggered without corrupting the checksum, we analyze the different exception types. As explained before, it can be assumed that the checksum function itself executes uncompromised, i.e., it is guaranteed that the SobTrA code runs untampered:

Synchronous exceptions. Those exceptions are triggered by executing specific instructions including an Undefined Instruction, Supervisor Call, Hypervisor Call and Secure Monitor Call. Since the SobTrA code runs untampered and does not contain any of these instructions, an attacker cannot abuse those.

Reset exception. A reset is triggered by setting the corresponding CPU pin. Hence, we consider this a hardware attack that is out-of-scope for SobTrA.

IRQ, FIQ and external Abort. External interrupts are disabled by SobTrA, which is reflected in the checksum incorporating the CPSR and its respective masking bits. Together with Property 1, this ensures protection against this type of exception, as higher level interrupts, which might not be maskable by SobTrA, are not available to an attacker. Furthermore, in Monitor mode, SobTrA is additionally protected by Property 3. Note that in some systems there might be a non-maskable FIQ, enabled by setting an input pin at boot time. As for the reset exception, an attacker would require a hardware attack, which we consider out-of-scope for SobTrA.

Prefetch Abort. These are triggered by the MMU when fetching instructions from invalid memory addresses. The checksum function and uninterruptible execution environment establishment code reside on the same minimal page preventing Prefetch Aborts and excluding them as attack vector.

Data Abort. These are triggered by the MMU when accessing invalid data addresses. For the current prototype, the checksummed region is about 8 KiB in size, which means that a Data Abort might be caused by an attacker. Since data access follows a pseudorandom pattern, SobTrA is protected by Property 3. Furthermore, the checksummed region could probably be space optimized to fit into a single page in the future.

Summarizing, SobTrA is able to protect itself against exceptions in all critical phases of its execution. This is the case for all target platform configurations and SobTrA running in Supervisor (EL1), Hypervisor (EL2) or Monitor (EL3) mode.

4.7 Prototype

We implemented a prototype able to execute a complete software-based trusted boot using SobTrA. The prototype hardware consists of a Beagleboard (Rev. C3) [Bea09] as the target platform and an mbed LPC1768 microcontroller [NXP16] as verifier, i.e., token, interconnected via SPI. The Beagleboard employs an OMAP3530 SoC (ARM Cortex-A8, 600 MHz) by Texas Instruments [Tex12] with Harvard-style 16 KiB level 1 caches and 256

KiB unified level 2 cache. The mbed NXP LPC1768 is an ARM Cortex-M3 microcontroller by NXP clocked at 96 MHz. It contains 512 KB Flash storage and 32 KB SRAM. As discussed in Sections 3.2 and 4.4.1, except for an independent timing source, there are no specific requirements regarding the verifier device hardware. While our general architecture might suggest using a more capable token device, we intentionally chose one from the lower end of the spectrum to show that the SobTrA verifier code can run on almost any device.

SobTrA on the Beagleboard is implemented as bare-metal, standalone binary image containing the checksum function, an SPI driver for communicating with the target platform, a minimalist SHA-1 implementation and a very basic ARM Linux kernel bootloader for starting a trusted kernel. In the prototype boot sequence, SobTrA is loaded as an additional boot stage on the Beagleboard and initiates the protocol as SPI-Master. After successful completion of the protocol, the verifier can be sure that the kernel it received a hash for is started on the target platform. In case the SobTrA protocol fails or the hash is not correct, the token can refuse any subsequent access to security functionality.

For the verifier side, we implemented a SobTrA simulator which is able to compute challenge-response pairs without running on the actual target platform hardware. On the verifier, timing is not an issue, and we are therefore able to use a high-level language, namely C, for the implementation. This gives us maximum flexibility in the choice of the verifier hardware, requiring only recompilation and minor platform adaptions. As discussed before, the time the target platform is allowed to take to complete the checksum calculation must be pre-measured on the actual hardware and cannot be done using the simulator. Note that the execution time only varies with the number of iterations of the checksum function and *not* with the particular challenge. Therefore, after determining the number of iterations to be used (see below), the corresponding time threshold has to be measured only *once* on the target hardware and can be re-used for an arbitrary number of SobTrA executions, i.e., boot processes, afterwards.

4.8 Experimental Results

We conducted two main experiments with our prototype. In the first, we wanted to determine how reliable time measurements for checksum calculations can be reproduced to ensure that an attacker's SobTrA execution can be reliably distinguished from an untampered execution. In the second experiment, we analyzed how the iterations of the checksum function influence the execution time overhead for an attacker.

As discussed in Sections 4.4 and 4.5 and shown in more detail in [Hor+14b], an attack on our checksum function produces at least *one CPU cycle overhead* per checksum function block. Hence, for the first experiment, we repeatedly measured the SobTrA execution time using a fixed number of iterations (1.5 million) for both untampered execution and an ideal attacker's execution with exactly one additional CPU cycle per checksum function block. For both cases, we differentiated between pure execution time and time

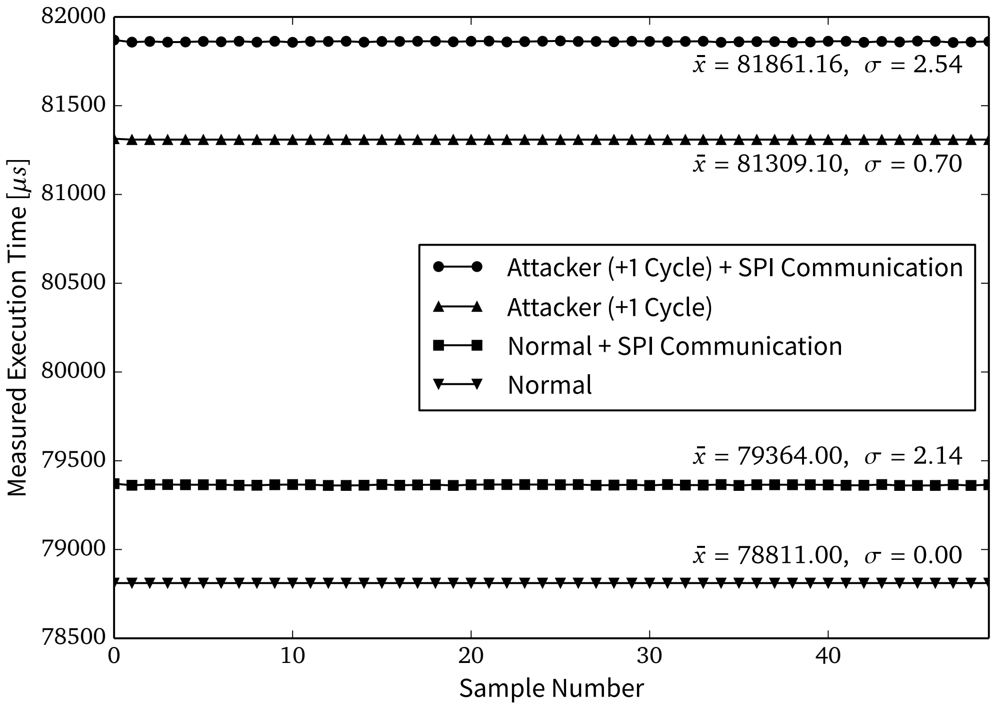


Figure 4.8: Consecutive measurements of SobTrA execution time for 1.5 million iterations.

measured by the verifier via SPI at 3 MHz. The resulting plot is depicted in Figure 4.8. It shows 50 samples with measurements for normal and attacker's execution with and without SPI communication. For each of the measurements, the plot additionally shows the mean value ($\bar{x}$) and standard deviation (σ) over all samples. With an overhead of $^{81861}/_{79364} - 1 \approx 3.15\%$ in the real world case including the communication, even attacker overheads smaller than one cycle per checksum function block are measurable quite well. Even more importantly, the samples show a very low standard deviation, which ensures that measurements can always be reliably evaluated. First, this shows that SobTrA itself is able to provide a solid basis for all kinds of physical communication interfaces between target platform and verifier, as it introduces almost no time variance at all. Second, it shows that the SPI communication has a low latency.

In the second experiment, we measured the execution time, including communication overheads, for an increasing number of 1000 to 100000 iterations with a step size of 1000. Again, we took samples of the normal untampered execution and an attacker's execution with *one* cycle overhead. Figure 4.9 shows the overhead generated by the attacker, i.e., the difference between normal and tampered execution time. The plot allows us to identify the number of iterations necessary to generate a specific attacker's overhead. On the average, about 24000 iterations are needed to generate 40μs overhead for the attacker. An overhead

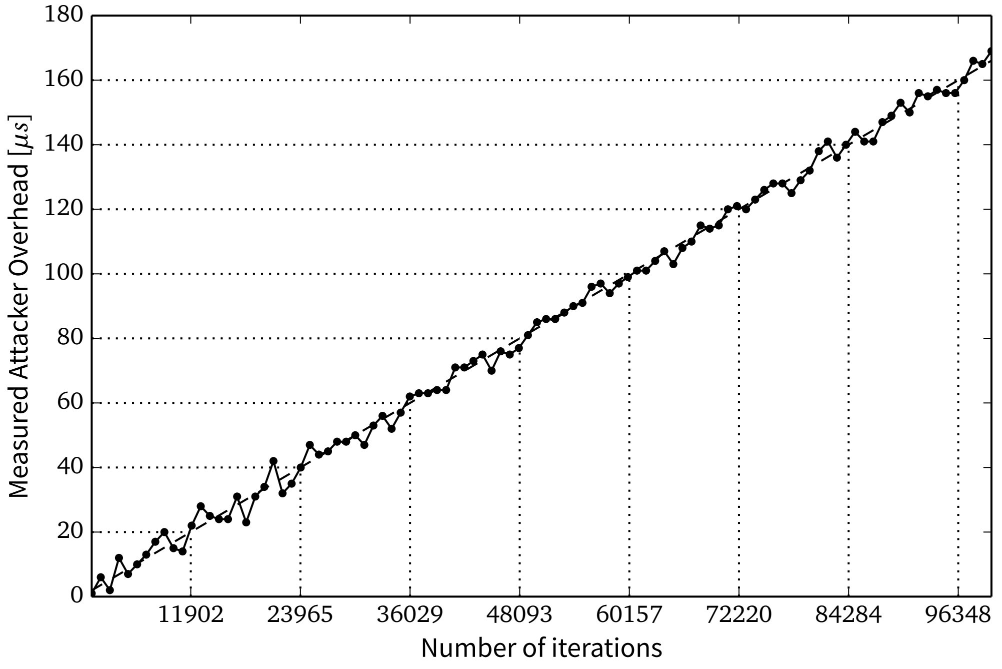


Figure 4.9: Measured attacker overhead (+1 cycle) for 1000 to 100000 iterations.

of 40μs is already reliably measurable and the 24000 iterations are also sufficient to cover SobTrA's 8 KiB in the random memory traversal, as discussed in [Hor+14b]. The SobTrA execution time in this case is about 1.8 ms. The results from both experiments indicate that it might even be possible to derive a target platform-dependent linear function for calculating the execution time based on the number of iterations. Then, the iterations could be varied dynamically without having to pre-measure for each specific iteration count.

Summarizing, the experiments with our prototype confirm that the verifier is able to repeatedly, reliably and quickly distinguish between tampered and untampered execution of SobTrA on the target platform.

4.9 Summary

In this chapter, we discussed how the token can establish trust in the code integrity of the target platform at startup. This is an important achievement as it allows the token to protect its secrets and security functionality from a target platform initially controlled by an attacker.

To be independent of specific software attestation features in the target platform's hardware and from shared or trusted key material between the token and the target platform, we introduced SobTrA, a software-based trust anchor for ARM Cortex-A processors.

With this, a verifier device, such as the token, obtains the guarantee that the SobTrA code itself executes untampered on an attached, initially untrusted target platform using a self-checksumming calculation that the target platform must complete within a certain timeframe. Based on SobTrA, we developed the concept of a software-based trusted boot in which the verifier can obtain the guarantee that a specific bootloader, firmware, hypervisor or kernel identified by its hash value is started.

We implemented a Cortex-A8-specific checksum function resistant against known attacks. Furthermore, we analyzed ARM exception handling in detail and designed an uninterruptible execution environment ensuring that an attacker cannot gain control of the execution through exceptions during or after the checksum function execution. Our prototype is able to run a software-based trusted boot up to a Linux kernel. In our experiments, the approach showed very robust timing, enabling the token to clearly distinguish untampered and tampered execution.

In terms of our general architecture (see Chapter 3), after execution of SobTrA, the token has the guarantee that the boot process on the initially untrusted target platform runs unmodified. In our architecture, SobTrA can either be executed in the TEE or in hypervisor mode. Based on the established trust and the fact that the token only provides access to its functionality after the successful software-based trusted boot, we can build a chain of trust for following boot stages and start extending them with additional security functions.

5 Monitoring and Protecting Run-time Integrity

In the previous chapter, we introduced the software-based trusted boot as a mechanism to establish the token's trust in the boot-time integrity of privileged software components on the target platform. With a chain of trust built upon this boot process, the token's trust can be extended up to the level of user land code but only at *load-time*. With an additional secure boot process, the integrity of all code in our target platform can be ensured but also only at *load-time*. Large code bases typically contain a lot of vulnerabilities that can be exploited at run-time by software attacks. To harden the target platform against such attacks and, hence, be able to maintain the initial trust of the token, it is crucial to protect the run-time execution of code after its integrity was ensured at load-time.

Protection of run-time code execution is a huge research field. Most of the protections are built into the kernel or the target binary itself. Sometimes protections additionally rely on special hardware features. For example, Data Execution Prevention (DEP) prevents simple code injection attacks and requires hardware and OS to provide support for non-executable memory. In this chapter, in order to improve compartmentalization and reduce the TCB of a run-time integrity-protected target platform, we want to explore novel ways to leverage logical, in-CPU separation for monitoring and protecting code execution. More specifically, we want to explore protection mechanisms that work completely from a higher privilege level, namely the hypervisor. The idea is depicted in Figure 5.1. With our approach, the hypervisor should be able to transparently gather information about the Control Flow (CF) of the lower privilege levels, analyze it and react in case of a detected anomaly.

On x86 platforms, the approach to move security software to the VMM of the system, i.e., to the hypervisor, is well-known and thoroughly researched achieving different goals depending on the use case. For malware detection and analysis, being located in the hypervisor allows the security software to be highly transparent while at the same time having a detailed view on the system state [VY05; GR03; Den+12; Ngu+09; Din+08; Sha+09; Sri+11; JWX07; DZX13]. For run-time integrity protection, the additional privilege layer offers protection against malware in the kernel layer and reduces the

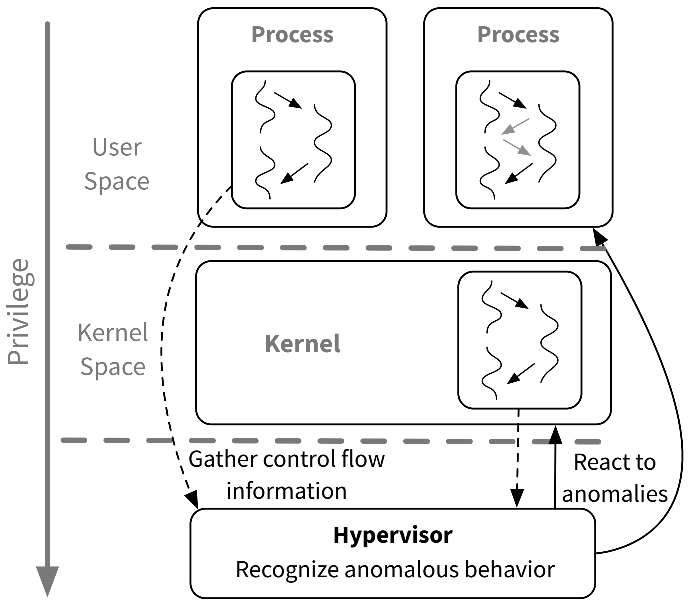


Figure 5.1: Hypervisor-based control flow integrity protection.

TCB [Rhe+09; XTL11; SG11; Jia+11; HS12; RLX11; Ses+07]. The introduction of the virtualization extensions for the ARM architecture [MN11; ARM14; ARM17] allows us to follow a similar approach for an ARM-based target platform.

We first introduce and discuss a framework for monitoring, i.e., tracing, code execution on the target platform from a minimal custom ARM hypervisor. Our framework uses the Second Level Address Translation (SLAT) provided by hardware-assisted virtualization in recent ARM CPUs (see Section 2.2.2) to transparently restrict the set of executable pages of the guest. This causes traps at control flow transfers between pages giving the hypervisor insights on the guest's execution. Besides providing a small TCB, our framework is able to transparently trace not only userland code but *also kernel code*, does not require any modifications to the guest and is agnostic to the specific guest OS in use. With these properties, the framework can be valuable for a variety of use cases including malware detection, malware analysis and run-time integrity protection. The evaluation of our prototype implementation shows that despite the vast number of hypervisor context switches such a concept involves, the performance is surprisingly good. This not only makes the framework a valuable basic tool for realizing existing x86 VMM-based security concepts on ARM, but also shows that the overhead of hypervisor interaction on ARM is very small allowing for a whole new class of VMM-based security applications. Those applications rely on very frequent switches between the layers, typically incurring too much overhead on other architectures, such as Intel x86. As an example for such an application, we developed a security concept that uses the framework to enforce a previously determined page-granular integrity protected control flow in the guest to protect it from Control Flow (CF) hijacking attacks.

The chapter is organized as follows. We first motivate the approach by discussing an attacker model in Section 5.1 and related work in Section 5.2. Then we introduce the tracing framework in Sections 5.3, 5.4, 5.5 and 5.6 and the CF protection we built upon it in Section 5.7. Finally, we discuss the performance of our prototype implementation in Section 5.8.

Parts of this chapter have been published in [HW15].

5.1 Attacker Model and Attacks on Control Flow

In Section 3.3, we defined a *remote* and a *physical* attacker depending on the attacker's position in the architecture. In this chapter, we consider a *remote* attacker who attacks the target platform via the communication channels it deliberately provides. Such an attacker is depicted in Figure 3.2 on the top. We assume that one or more of the processes handling external communication on the target platform are written in memory unsafe programming languages, such as C and C++, and contain memory corruption bugs, e.g., exploitable buffer overflows. Our attacker is able to divert the CF and take control of an exposed process by exploiting a memory corruption vulnerability. In a second step, our attacker is able to use the overtaken process to attack the kernel running on the target platform. The attacker is *not* able to access local hardware interfaces of the target platform, especially no hardware debug interfaces, such as via JTAG.

In the following we discuss the attacks our attacker is able to execute. Memory corruption bugs in memory unsafe languages [Sze+13] are diverse and so are the attacks our adversary can use to gain initial control of an outward-facing process on the target platform. Typical memory corruption bugs include buffer overflows and dangling pointers. Simple attacks exploit those bugs to introduce new, malicious program code into the victim process (*code injection*) or to modify present code (*code corruption*). Afterwards, the attacker diverts the CF of the program to the malicious code. A typical example is the *stack code injection*, where the attacker overflows a stack-based buffer to inject malicious code and at the same time modifies the return address of the current function stack frame to execute it when the function returns. Code corruption and injection attacks rely on the ability of the process to introduce new code, either by modifying or adding code memory regions. Modern processors and OSs prevent those attacks by using DEP and non-writable code memory for almost all processes. Exceptions are processes generating or modifying code, such as for Just-In-Time (JIT) compilation. More recent attacks circumvent DEP and similar restrictions by re-using code already present in the victim program. Those so-called Code-Reuse Attacks (CRAs) typically modify one or more code pointers, such as return addresses, to divert the CF of the attacked program to a location of their choosing in the

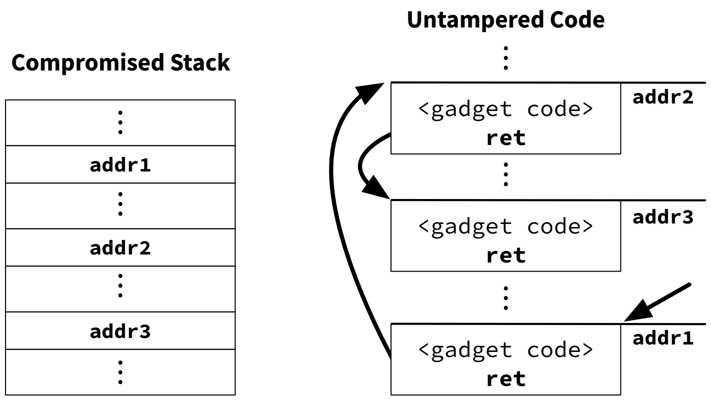


Figure 5.2: Basic idea and memory layout of a ROP attack.

original code. Simple CRAs, such as *ret2libc*, only change a single pointer and try to gain control by directly diverting control flow to an exploitable function, for example, to execute a command with a `system()` syscall. More advanced techniques, such as ROP [Sha07], stitch together multiple *gadgets*, i.e., small pieces of code with a CF transfer at their ends, to execute more complex attacks. Figure 5.2 visualizes the basic idea of a ROP attack. While this technique was initially implemented for the x86 architecture [Sha07], it has spread since then also to the ARM architecture [Kor10; Dav+10b; Wan+13].

As mentioned, in a second step our attacker might try to elevate the privileges of the process it controls by exploiting the kernel. While privilege escalation in general is a very diverse topic and a full discussion therefore out of scope, it should be noted that also here CF diversion is a common attack technique [KPK14].

Summarizing, all attacks our adversary is able to conduct rely on diverting and taking over the CF of the victim process. Consequently, we use the term *control-flow hijacking attack* to denote all those attacks in the following. Attacks not modifying code or code pointers, such as Data-Oriented Programming (DOP) [Hu+16], are excluded from our attacker model. In a DOP attack, the attacker corrupts variables influencing the CF, for example, loop counters, instead of code pointers. If specific, relatively rare DOP gadgets are present in the attacked code, a DOP attack can be mounted to realize arbitrary functionality, while still adhering to the legitimate Control Flow Graph (CFG) of the attacked program.

5.2 Related Work

VMM-based security concepts are well researched for x86 platforms. Many of these concepts are very x86-specific and Section 5.5 gives an overview how our framework can be used as a basis to realize them on ARM. In the following, we discuss concepts employing the same fundamental mechanisms as our framework and prior work that focuses on ARM or relates to the page hash-based CF protection example application described in Section 5.7.

MAVMM [Ngu+09] is a minimal, custom designed x86 hypervisor for malware analysis. Apart from the idea of using a specifically tailored, minimal hypervisor, the approach is very different from our framework, using different trapping points and tracing granularity. There are a lot of VMM-based concepts that use the SLAT page access permissions as a basic mechanism to achieve different goals. Rhee et al. [Rhe+09] use it to check the integrity of writes to kernel memory. Several publications [HS12; SG11; XTL11] use it to separate kernel modules from the kernel and from each other. *Secvisor* [Ses+07] uses SLAT memory protection to enforce that only approved kernel code is executed in kernel mode. IntroLib [Den+12] allows library call introspection via SLAT access permissions and Nitro [PSE11] uses the SLAT for system call tracing. Furthermore, different transparent breakpoint techniques [DZX13; VY05] rely on the feature. But none of the approaches is realized for ARM and is able to generate page-granular traces as our framework does, probably because of the performance issues such an extreme usage of the SLAT would impose on x86. To the best of our knowledge, we are the first to employ the mechanism on ARM and provide detailed a performance analysis, which can also be useful as a generic benchmark of real-world ARM virtualization.

Virtualization on ARM in general is a relatively new research subject compared to virtualization on x86. There are some publications that deal with general aspects of virtualization on embedded [Hei09] and mobile platforms [Yoo+08] or specifically on ARM [VH11; Din+12; DN14]. Dall et. al [Dal+16] provide a detailed performance analysis of ARM hardware virtualization confirming our result that switches from guest to hypervisor and vice versa are really fast at least for bare metal hypervisors.

DroidScope [YY12] is a QEMU-based framework, focusing on Android introspection and is complementary to our efforts. There are two publications [GVJ14; Aza+14] introducing architectures that use the ARM TrustZone to secure the kernel. Since the TrustZone is not a virtualization solution (see also Section 2.3.1), it lacks support for basic functions required by a hypervisor, especially necessary trapping functionality. SLAT as the basic building block of our framework is obviously not supported by the TrustZone. The authors of both papers propose instrumenting the kernel and replacing instructions with calls into the TrustZone to work around this limitation. Obviously, this approach is not transparent and relies on integrity of the guest kernel, which they try to ensure with certain initial requirements to the guest translations and a secure/trusted boot. Hence, in contrast to our framework, the mechanism cannot be used to implement page-granular, transparent tracing of user and especially guest kernel code.

Our example application (see Section 5.7) can prevent CF hijacking attacks in the guest (see Section 5.1) by enforcing a previously determined page-granular CF. As shortly discussed in Section 5.1, there is a huge body of work regarding memory corruption and CF hijacking attacks and their prevention. Szekeres et al. [Sze+13] provide a detailed analysis and categorization of memory corruption attacks and prevention techniques. The probably most-noticed attack technique is ROP and the majority of defensive concepts for ROP relies on compiler modifications [Tic+14; Ona+10], binary rewriting [Aba+05; ZS13] or dynamic binary instrumentation [KBA02; Vee+15; PBG15; DSW11] and not on VMM-based security. Many of the concepts involve Control Flow Integrity (CFI), originally introduced by Abadi et al. [Aba+05], enforcing certain rules to CF transfers on user or kernel level. Our approach can be used to implement page-granular CFI. Being VMM-based, it can provide an extra of attack resistance and transparency. Other VMM-based approaches typically only try to protect the guest kernel and kernel modules [SG11; XTL11; Ses+07] or only try to protect against code injection attacks [WS12], while our security application can be used for control flow protection for the guest kernel *and* user mode.

5.3 Hypervisor-based Monitoring Framework

In the following, we describe our tracing framework. First, we describe basic hardware requirements, then we discuss the framework's architecture, initialization and the fundamental tracing mechanism. Subsequently, we discuss the CF data that can be gathered with the framework.

5.3.1 Hardware Requirements

Our framework relies on some basic features of hardware-assisted virtualization to trace a guest efficiently. The basic principles of CPU support for virtualization have already been introduced in Section 2.2.2. In the following, we shortly discuss which of the features are important for the tracing framework.

The basic hardware-assisted virtualization feature that our framework uses for execution tracing is the hardware-supported SLAT, which gives the hypervisor the ability to map pages independently of and transparently to the guest kernel and user space. The SLAT basically adds a further stage of translation for all addresses accessed by a guest. This means that a VA inside a guest is first translated to a *Guest-Physical Address (GPA)* or *Intermediate Physical Address (IPA)* as ARM calls it, which is in turn translated to an actual PA. The first translation is controlled completely by the guest while the second mapping is controlled by the hypervisor and is transparent to the guest. This enables the hypervisor to run multiple guests with the same guest-physical address space while separating them in physical memory. The second important feature is the enforcement of separate access permissions on the SLAT, especially the XN bit, to be able to transparently prevent higher layers from executing or writing specific memory pages.

The main reason for requiring hardware-supported SLAT is performance. Since our concept relies on frequent switches between processor modes and changing of the second translation tables, a software implementation would not come near a usable solution. Furthermore, it would significantly increase the size of the hypervisor and therefore the TCB.

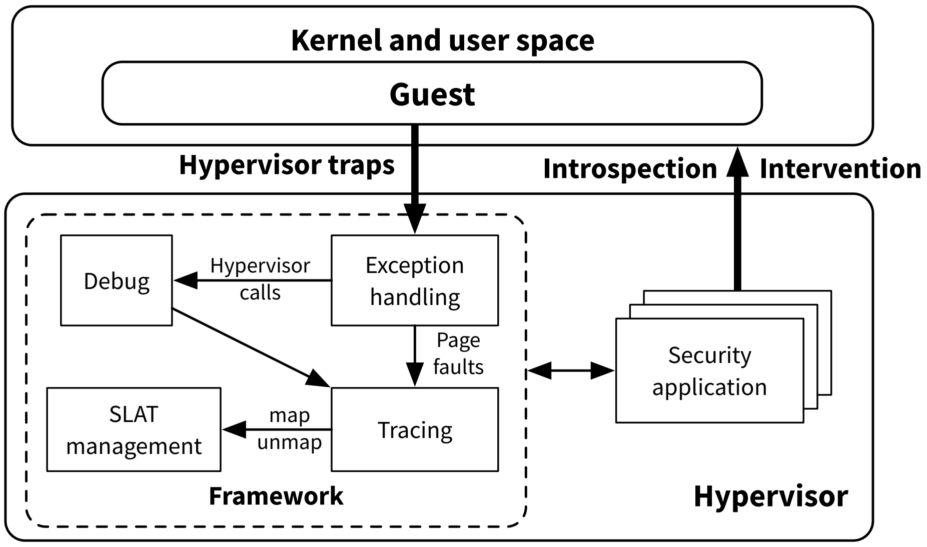


Figure 5.3: Monitoring framework architecture.

The described features, especially the SLAT, are part of the virtualization support of all major CPU architectures.[1] Therefore, major parts of the framework's concept are not specific to ARM. Nonetheless, since the target platform in our general system architecture (see Chapter 3) is typically ARM-based and the framework relies on frequent switches between hypervisor and guest, which are —as we will elaborate in our evaluation— particularly fast on ARM, we focus on the ARM architecture for our design and implementation. More specifically, we focus on the ARMv7-A [ARM14] architecture, but most of the concepts also apply directly to the more recent ARMv8-A [ARM17] architecture and differ only in terminology.

5.3.2 Architecture

The framework is built on top of a minimal, custom hypervisor on our target platform, which basically only supports memory management and is not able to run multiple guests. This restriction is not a conceptual limitation, but drastically reduces complexity and code size of the hypervisor and, hence, also reduces the risk for the framework to be detected or attacked. The hypervisor does not interfere with the guest's communication with the hardware. The hypervisor effectively protects its own memory not mapping it into the guest using the SLAT. For protection against DMA attacks from the guest, the System Memory Management Unit (SMMU) [MN11], ARM's version of an IOMMU, must be configured

[1]Intel calls its SLAT implementation "Extended Page Tables", AMD's implementation is named "Nested Page Tables".

accordingly. Physical attacks via debugging interfaces, e.g., using JTAG, are out of scope as discussed in Section 5.1.

The architecture of the tracing framework is depicted in Figure 5.3. It mainly consists of four components. The *exception handling module* receives all configured traps from the guest, i.e., from the guest's kernel and user mode. It dispatches HVCs to the debug module and all guest page faults (*abort exceptions* as ARM calls them) to the tracing module. The *debug module* contains a minimal serial driver to be able to provide information without relying on the guest. It furthermore receives HVCs to dynamically change parameters of the tracing. The *tracing module* uses the page fault information and the *SLAT module* to generate tracing data as described in Section 5.3.4. The resulting information is provided to *security applications* running alongside the framework in the hypervisor. These applications can use the data and their own introspection methods to determine an appropriate reaction to the guest's behavior and state if necessary. The details of this interaction depend on the particular security application. We present an example for such an application in Section 5.7.

As explained before, in terms of our general architecture (see Chapter 3), the tracing framework is completely located on the target platform. Its initial integrity is ensured by the token as described in Chapter 4.

As mentioned before, the basic idea is to minimize the code base of the hypervisor, i.e., the TCB, by implementing only the functionality required for tracing one guest. But the framework can also be used in conjunction with a full-fledged hypervisor supporting multiple guests. In this case the tracing data structures described in the following sections must be duplicated for each guest.

5.3.3 Initialization

During startup of the system and before the guest runs, the hypervisor initializes the SLAT to an identity mapping, which maps GPAs, called IPAs in the following, flat to PAs (except for the memory space where the hypervisor resides), preserving memory attributes such as cacheability and overlaying access permissions. By enabling the XN bit for all guest pages in the SLAT, the framework effectively prevents the guest from executing *any* page initially. The SLAT is furthermore configured to the smallest possible page size, which is 4 KiB, to make the tracing as fine granular as possible. The ARM architecture ensures that the access permissions provided by the guest translation and the hypervisor's SLAT are combined so that always the more restrictive setting applies. The framework initially does *not* restrict read or write accesses on the pages. On multi-core systems these initializations are executed for each core separately.

The framework furthermore initializes an empty list with a maximum size n per core which later contains its currently executable pages. As described later, the variable n, determining to the maximum number of executable pages, is an important parameter allowing us to adjust the accuracy of the tracing as a trade-off for performance.

5.3.4 Tracing Mechanism

With the start of execution of the guest, the following steps are repeatedly executed by the framework to gather control flow information:

1. The guest transfers execution from one page to another.

2. If the page is not in a set of n physical pages for which execution is currently allowed by the SLAT, the control transfer immediately triggers a page fault (a *prefetch abort*) into the hypervisor.

3. The framework can now analyze the prefetch abort and extract the trace data as described in Section 5.4.

4. To allow further execution of the guest, the hypervisor then makes the page executable before returning execution to the guest. In most cases a page from the set of n executable pages must be replaced according to a previously defined eviction strategy such as a simple CLOCK algorithm.

Note that all these steps are completely transparent for the guest and do not require any changes to the guest kernel or other software running in the guest. The choice of n has significant influence on the expressiveness of the gathered control flow data (see Section 5.4) since the framework allows the guest to transfer execution between the pages of the current set arbitrarily without triggering exceptions into the hypervisor. The source of the CF transfer in the described tracing mechanism can be anywhere within the currently executable set of pages. Consequently, a smaller set is desirable but reduces performance. The performance impact of the set size is discussed in detail in Section 5.8.

5.3.5 Multi-core Tracing

In a multi-core system, each core has to have its own set of executable pages. This means that our framework must manage distinct, per-core translation tables in the hypervisor. Otherwise, the cores would contest for the limited number of executable pages and produce a non-deterministic order of possibly completely unrelated page faults, which, in turn, would corrupt the resulting CF information. The translation tables of the cores must be identical in terms of actual mappings from IPA to PA, providing the same physical memory view to all cores. The only difference between the tables are the access permission flags determining which pages are currently executable on the specific core.

Figure 5.4 visualizes the tracing mechanism as a minimal example on a two-core system with three memory pages, all mapped executable in the guest. In the example, the tracing framework is configured to a maximum executable set size of $n = 2$. Figure 5.4 shows the system during a hypothetical CF transfer from the first to the third page in the guest. On both cores, the framework maps the three available pages flat from IPA to PA space and

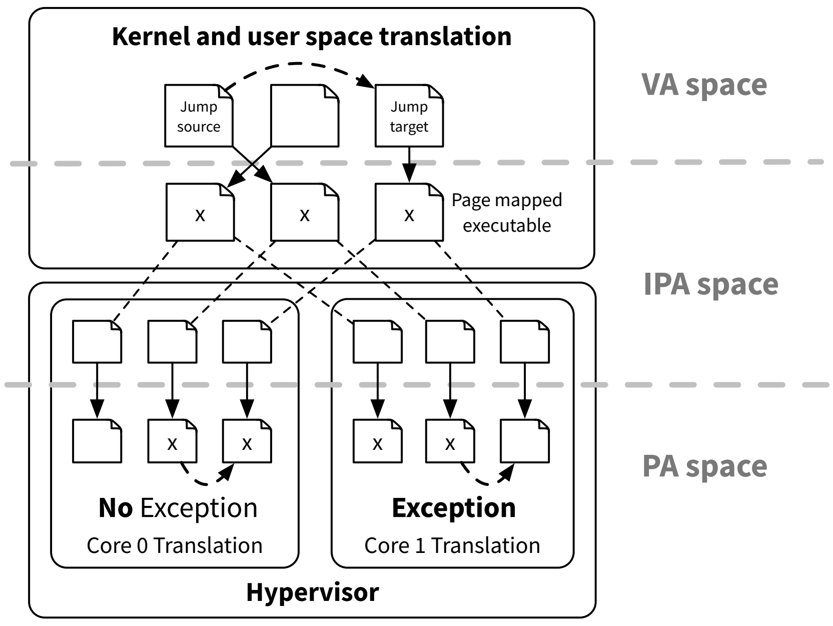


Figure 5.4: Gathering control flow data in a multi-core system.

restricts their access permissions. On the first core, the source and destination page of the guest's CF transfer are both in the set of executable pages. Hence, if executed on this core, the transfer can happen without triggering the tracing.

On the second core, only the source page is mapped executable. Thus, if executed on this core, the transfer triggers a prefetch abort exception into the hypervisor which the framework handles and analyzes to gather data regarding the jump in the guest. Afterwards, the framework maps the destination page executable while evicting the page not involved in the jump (assuming $n = 2$). Note that, although the example in the figure only shows the case of a *jump* between two pages, the framework is obviously also able to recognize the continuous CF transfer between two adjacent pages the same way.

In the example, both translation levels use the same page size for simplicity. The tracing granularity is only influenced by the page size in the SLAT. Hence, independently of the guest, the framework can use the smallest SLAT page size to ensure the maximum precision for the tracing. For a CPU architecture other than ARM that only supports SLAT page sizes larger than its first level pages, our tracing mechanism still works but loses precision as multiple first level pages map to one SLAT page and jumps between them cannot be intercepted by the hypervisor. Generally speaking, the smaller the SLAT pages can be configured, the higher the precision of the control flow data that can be gathered.

The example in Figure 5.4 intentionally makes no distinction between user and kernel space to illustrate the fact that it is irrelevant to the mechanism in which of both contexts a CF transfer happens to gather information about it. Note that the framework still can distinguish between kernel and user mode transfers and provides this information to an attached security application as described in Section 5.4.

5.3.6 Write XOR Execute Protection

The basic tracing mechanism introduced in the previous sections has one limitation: An attacker can use self-modifying code and a page that is readable and executable at the same time to execute arbitrary code without ever triggering an exception into the hypervisor. With this technique, the attacker can effectively circumvent the tracing. Hence, we extend the basic tracing with a strict W^X protection on the SLAT pages, preventing them from being mapped writable and executable at the same time. The W^X extension does *not* generally prevent the guest from executing self-modifying or JIT-compiled code but makes sure that access patterns for such code are reflected in the gathered CF information. Such a pattern could be, for example, repeated writes to a page followed by its execution.

To realize the W^X protection, we slightly modify the basic tracing mechanism. In the initialization phase, the framework now maps all pages as read-only. This means that now not only fetch but also data write accesses (a *data abort* exception) trigger exceptions into the hypervisor. Both types of exceptions, data and prefetch abort, must be handled differently. The prefetch abort is handled as described for the basic mechanism by evicting a page from the set of executable pages for the core that triggered the exception. But the framework must now additionally make sure that the page is not mapped *writable* on *any* core before that. In turn, when a data abort is triggered because of a write access to a read-only or executable page, the hypervisor must ensure that the page is not mapped *executable* on *any* core before mapping it writable for the core that triggered the exception. Otherwise, a scenario would be possible in which one core executes a page that is concurrently being modified by another core, effectively circumventing the W^X protection. Still, each core maintains its own translation table and its own set of executable pages with a maximum of n pages. There is no limit on the number of writable pages.

For the W^X protection, the SLAT page size should be the same or smaller than the guest-physical page size to avoid falsely recognizing a W^X violation for pages consisting of multiple pages in the guest. For the ARM virtualization extensions this is always possible and configuring the SLAT page size to 4 KiB avoids the problem altogether since 4 KiB is the minimal guest-physical page size.

Listing 5.1 shows the pseudo code for the two exception handlers in our framework implementing the tracing mechanism with W^X. All pages in the algorithm are guest-physical and flat mapped to physical memory with access permissions determined by the sets they are part of ($X(i)$ and $W(i)$). The algorithm identifies pages by their guest-physical addresses. Normally, the address of the guest-physical page p for which the exception is triggered is provided to the hypervisor by the hardware via the `HPFAR` register. In some cases, `HPFAR` is “unknown” [ARM14]. In such cases, the framework determines the guest-physical page by translating the faulting VA using the `ATS1C` architectural registers.

Listing 5.1 Tracing framework exception handlers.

```
State per core:
    $X(i)$: Set of executable pages of core $i$
    $W(i)$: Set of writable pages of core $i$
Exception Input:
    $p$: The page for which the abort was triggered
    $core$: The number of the core for which the abort was triggered
procedure HANDLE PREFETCH ABORT($p$, $core$)
    Gather control flow information (see Section 5.4)
    for $i = 0$ to $\#CORES$ do
        if $p \in W(i)$ then
            Remove $p$ from $W(i)$
        end if
    end for
    Replace a page from $X(core)$ with $p$ or insert it in an empty slot if available
end procedure
procedure HANDLE DATA ABORT($p$, $core$)
    for $i = 0$ to $\#CORES$ do
        if $p \in X(i)$ then
            Remove $p$ from $X(i)$
        end if
    end for
    Add $p$ to $W(core)$
end procedure
```

Since the different cores access not only their own translation tables and data structures for executable and writable sets of pages, accesses must be synchronized. Section 6.3.7 describes how to realize efficient synchronization for a different, but similar multi-core SLAT management scenario. Furthermore, the framework invalidates the page's guest TLB entry before returning to the guest.

5.4 Framework-provided Data

The tracing framework can provide different pieces of information about the guest execution and state to its security applications. As the hypervisor is entered on every SLAT page fault, the *frequency* with which security applications receive the information corresponds to the configurable number of concurrently executable pages n. For $n = 1$, security applications receive data for *every* control flow transfer from page to page in the guest. For $n > 1$, security applications receive data when the current set of n executable pages is left.

To understand the remainder of the section, a basic introduction in hypervisor trapping on ARM is useful. General aspects of the ARM architecture and virtualization have already been discussed in Sections 2.1.2 and 2.2.2. Hence, in the following, we only briefly summarize aspects specifically important for this section. An ARMv7-A processor with virtualization extensions provides nine different modes. One for handling each exception type, a system mode, and an unprivileged user mode. On a hypervisor trap exception, the processor switches from the current guest mode (PL0/1) to the Hyp mode (PL2), stores the preferred return address (depending on the trap type) in the ELR_hyp register and the CPSR in the SPSR of the Hyp mode (SPSR_hyp) and changes the PC to the address stored in the hypervisor trap exception vector. The hypervisor has an own SP register but shares *all* other registers with the user and system mode. After a trap, the hypervisor can determine the cause of the trap via the Hyp Syndrome Register (HSR). On a SLAT prefetch abort exception, the HIFAR holds the address of the instruction causing the fault. Furthermore, in most cases, the HPFAR holds the IPA of the faulting page.

On ARMv8-A processors, there are fewer processor modes as the processor only differentiates between Exception Levels (ELs) and does not provide a mode for each exception type. Those ELs also determine the privilege of the corresponding mode and therefore replace the ARMv7-A PLs. EL2 directly replaces Hyp mode and, when taking an exception to the hypervisor, the hardware provides the same information, differing only in some register names. For example, ELR_hyp becomes ELR_EL2 and SPSR_hyp becomes SPSR_EL2. Therefore, while the following discussion uses ARMv7-A terminology, the framework should be able to extract the same information on ARMv8-A.

The most important pieces of information the framework can extract for a guest control flow transfer are:

Control Flow Target. This includes the target VA the guest tries to execute and the corresponding IPA, from which the target PA and the physical page can be trivially inferred (flat mapping). The framework can extract the VA either from the HIFAR or the ELR_hyp register. The IPA can either be read from the HPFAR register or must be translated from the VA, e.g., using the ATS1C registers.

Control Flow Source. The CF transfer is guaranteed to originate from the set of currently executable pages. The smaller the sets size n, the more accurate the framework can determine the actual source. For $n = 1$, the framework can always provide the exact source guest-physical and physical (flat mapping) page. If the transfer occurred because of a branch-with-link instruction such as `blx` or `bl`, the LR of the mode the hypervisor was entered from can be used to further localize the exact source address.

Guest Context. The framework is able to provide information regarding the process context the guest's control flow transfer is happening in. This includes the ASID, used by the architecture to differentiate user space contexts, the active guest translation table and the CONTEXTIDR register, which the guest kernel can use to store the active PID.[1]

[1]The ARM Linux kernel can be configured to store the PID in the CONTEXTIDR.

Guest Execution State. The framework provides information which mode the guest is executing in and gives access to the guest's register set including the SP. It furthermore can extract the guest's CPSR from the `SPSR_hyp`.

The offset of the target VA into the page can give valuable hints regarding the cause of the CF transfer: If the offset is greater than zero, the transfer must be caused by a jump/branch instruction. If it is zero, the transfer is likely to be caused by continuous execution rolling over from the adjacent preceding page.

Note that the gathered information is independent of the specific kernel and user space software that is executing in the guest. Gathering further, introspection-based data is highly use case-specific and is therefore left to the security applications using the framework as described in the next section.

5.5 Framework Applications

As described with its architecture (see Section 5.3.2), our framework is designed to provide data to security applications running alongside (but not concurrently) in the hypervisor. When the framework provides data to an application, the application has the possibility to analyze the data, further inspect the guest's memory and, depending on the use case, react accordingly. Before sketching some use cases and the corresponding security applications, we first discuss how the framework can produce basic and (partly) introspection-based tracing data of different granularity, useful in several scenarios:

Function Call Tracing. Function calls can be traced if they span multiple pages using the framework provided CF source and target data. If the traced code can be recompiled, options like the `-ffunction-sections` (GCC) can be used to ensure that functions are page-aligned.

Library Call Tracing. With knowledge about the guest's memory layout, especially regarding the location of libraries in the virtual memory map of a process, the framework can be used to identify library function calls. Using the guest context information from the framework together with some basic introspection, this can be done for specific processes. In contrast to approaches like *IntroLib* [Den+12], our framework not only recognizes library calls, but realizes page-granular tracing, which contains the special case of library call tracing.

System Call Interposition. With page-granular tracing, the framework enables system call tracing for security applications similar to approaches for x86 [PSE11; Din+08]. For that, knowledge about the location of the exception vectors and/or other entrance points into the kernel is necessary to recognize the according pages when jumping to them. This can be very useful because direct trapping of user/kernel level exceptions to the hypervisor is only supported on ARM if the guest kernel does *not* enable the guest memory translation MMU and results in "unpredictable" behavior otherwise [ARM14]. If invasive debugging is available, the ARM vector catch feature could alternatively be used to trap all exceptions, including system calls. As this only captures system calls going over the exception vector, it might miss other kernel entry mechanisms, which our framework would still be able to recognize.

Instruction-level Tracing. The framework currently does not support instruction-level operation, i.e., single stepping or breakpoints for certain instructions. In contrast to x86, the ARMv7-A architecture does not provide a hardware mechanism to single-step a guest[1]. This makes it impossible to realize some x86-based concepts like *Stealth Breakpoints* [VY05] for ARMv7-A. The framework could be extended to support breakpoints similar to [DZX13], using `BRPT` instructions and splitting the data and code view for a page. If invasive debugging is enabled, which might not be possible[2], hardware breakpoints could be used and trapped to the hypervisor.

With support for these mechanisms, the framework can be useful in a variety of scenarios. It can, for example, be used as a basis for transparent *malware analysis* concepts often relying on system call interposition [Rie+11; Din+08], hooking OS API calls [Din+08], x86 branch tracing [Wil+12] or inserting hooks into the guest [Pay+08]. For malware analysis, complete transparency is the main, though unreachable [Gar+07] goal. Without components in the guest, the framework is already highly transparent. Nonetheless, for this use case the framework could be extended with full guest time virtualization, hiding latency caused by hypervisor traps as described in [Din+08].

The framework can also be used as a basis for porting x86 concepts in the field of *malware detection*. Many of these use system calls [HFS98; Kol+09; Bos+08; PG11] or OS API calls [Bos+08] as features for detection. But also concepts that detect malware based on periodic dynamic memory dumps [PH07] could be built for ARM using the framework.

Another application field is *run-time integrity protection*. Here, like for the malware analysis concepts, many x86-based approaches rely on system calls as interception and policy target [PG11; OOY08], which could be realized on ARM with the framework as a basis. For ROP prevention concepts relying on special x86 tracing features, like the Branch Trace Store (BTS) [PPK13] or the Last Branch Record (LBR) [Xia+12], it remains subject to future work to determine if the framework's page-based tracing is fine-granular enough to support them on ARM. Approaches that trap certain instructions [Che+09] might require the framework to be extended with breakpoint functionality (see above). Concepts like [Jia+11] inspecting the guest's stack for ROP indicators, could probably also be realized with the framework. Last but not least, the framework enables new run-time integrity approaches, one of which we present in Section 5.7.

[1]AArch64 in ARMv8-A introduces hardware single stepping.
[2]Invasive debugging requires the CPU to have the external signal `DBGEN` enabled.

5.6 Implementation

We implemented the framework on an Arndale board, a dual-core ARM Cortex-A15 developer board supporting the ARM Virtualization Extensions. Since the ARM Cortex-A15 is an ARMv7-A architecture CPU, the following description of our prototype implementation is focused on this architecture version. Nonetheless, as mentioned before, all the concepts and mechanisms basically apply also to ARMv8-A and its virtualization, mostly only differing in terminology.

As explained before and confirmed by our benchmarks (see Section 5.8) and by Dall et al. [Dal+16], hypervisor entries and exits on ARM are, in their most basic form, very lightweight. On x86, the `vmentry` and `vmexit` instructions implement a large part of the required VM state handling while on ARM only very basic operations are implemented in hardware as described in Section 5.4. Since our hypervisor does not require a complete VM state handling, this results in much improved performance.

The implementation is based on a minimal hypervisor we wrote from scratch implementing only memory management for a single guest. On top of this basic functionality we implemented our tracing framework. The hypervisor includes a minimal driver for serial I/O and supports Symmetric Multiprocessing (SMP). After startup and initialization, the hypervisor runs on both cores on the Arndale board. Hypervisor and framework together consist of about 3000 Lines of Code (LOC) (C and assembler) presenting a very small TCB.

5.6.1 Memory Management

Sections 2.1 and 2.2.2 gave an overview regarding ARM virtualization and the ARM architecture in general. Section 5.3.1 summarized hardware virtualization features important for the conceptual design of the framework. In the following, we briefly summarize details of the ARMv7-A virtualization crucial for our prototype and how we used them in our implementation.

The ARM Virtualization Extensions differentiate between Stage 1 and Stage 2 translations. The ARMv7-A architecture furthermore distinguishes between three different Privilege Levels (PLs). PL0 contains the user mode, PL1 the privileged kernel modes for exception handling (see also Section 5.4) and PL2 the hypervisor mode. Stage 1 translations take a VA as input and produce either an IPA or directly an PA depending on the context. Stage 2 translations take IPAs as input and translate them to PAs. Our hypervisor has to manage two different translations. First, it manages the translation of its own VA address space to PAs, i.e., the PL2 Stage 1 translation. For this, we simply employ a flat mapping to a physical memory region strictly isolated from the memory used for guests. The corresponding translation table is shared between both cores. Second, our hypervisor manages the SLAT from IPAs to PAs, i.e., the PL0/1 Stage 2 translation shown in Figure 5.5. The SLAT is the translation the framework uses to provide CF data. The SLAT mapping

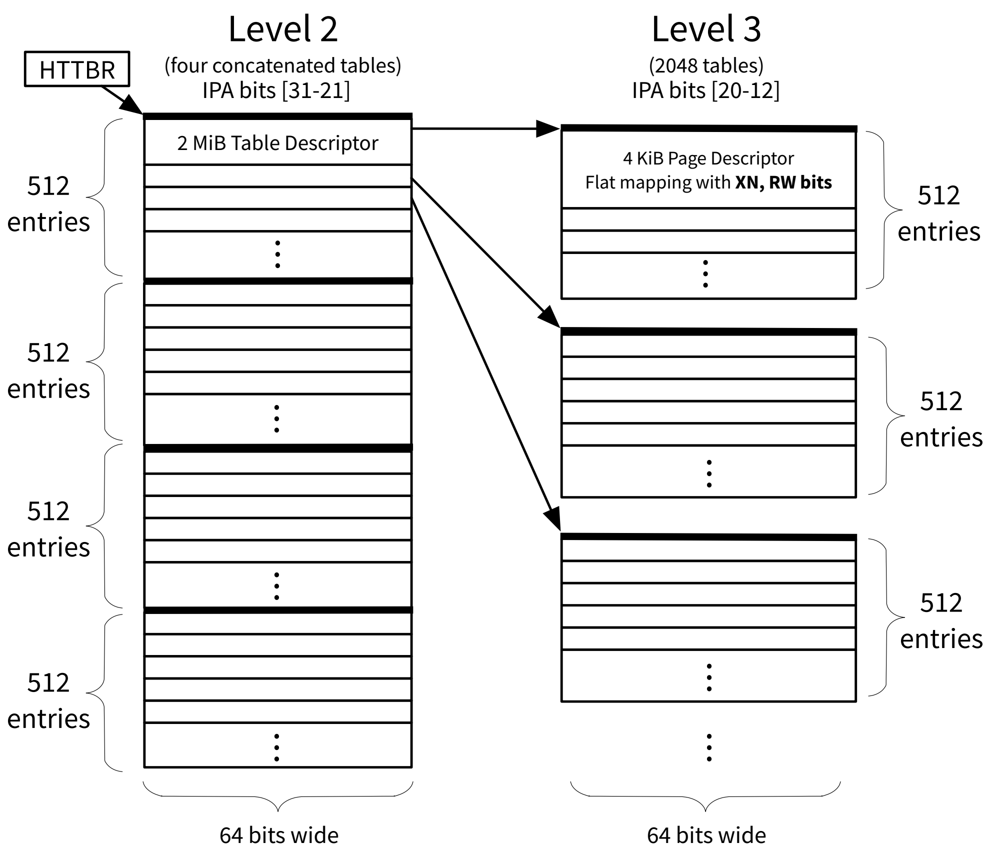


Figure 5.5: PL0/1 Stage 2 translation tables in the prototype implementation.

is configured by the hypervisor as described in Section 5.3.5 using a separate translation table for each core.

The PL0/1 Stage 2 translation architecturally supports input and output addresses of up to 40 bits length using the ARM Large Physical Address Extension (LPAE) and its long descriptor format. In our prototype, all addresses are 32 bits wide, which is sufficient since the Arndale board only provides two GiB of RAM. Additionally, using the same size for VAs, IPAs and PAs reduces the complexity of the implementation. Normally, the PL0/1 Stage 2 translation has three lookup levels using the finest granularity, i.e., 4 KiB pages. In our case with reduced input address size, the architecture allows us to skip the first level of translation, improving lookup performance. In this case, four second-level tables must be concatenated. Figure 5.5 shows the construction of the PL0/1 Stage 2 translation tables. The only parts modified after the initialization phase are the access bits and the XN bit in the level 3 page descriptors.

5.6.2 Tracing Framework Implementation

The ARM architecture allows the PL0/1 Stage 2 mapping to overlay all important properties of the PL0/1 Stage 1 translation. In most cases, this means that the more restrictive setting takes effect. Our tracing framework is only concerned with the access permissions, i.e., read/write permissions and the XN bit, and leaves all other attributes, like cacheability and shareability, to the guest's translation. For realizing the tracing mechanism, the framework explicitly stores a list of currently executable pages per core to be able to find them immediately in the translation table. The hypervisor starts the guest with a complete read-only translation table and successively fills and replaces the available slots for executable pages as described in Section 5.3.4 and shown in Listing 5.1. For testing purposes, we implemented a hypervisor call (`HVC` instruction) allowing us to change the number of executable pages dynamically.

We identified an interesting implementation problem originating from the two-part synchronization primitives `ldrex/strex` provided by the ARM architecture. If these two instructions happen to be on a page border with one of them on the first and the other one on the second page, running with only *one* executable page results in an infinite loop. This is because the hypervisor's page exception handler is repeatedly triggered and implicitly removes the lock acquired by the guest's `ldrex` with its own synchronization routines. Our prototype implements a heuristic to recognize such situations and handles them by allowing an additional executable page for a short time.

We verified the correct operation of the prototype by running a small deterministic bare metal application, the uboot bootloader, and checking if the execution transfers between pages for the known path of execution are recognized correctly.

5.7 Page Hash-based CF Protection

The page hash-based CF protection is an example application we developed using the presented tracing framework to enforce a particular, pre-determined page-granular control flow in the guest's kernel *and* user mode as a run-time integrity mechanism against control flow hijacking attacks described in Section 5.1.

The security application identifies pages by the hash digest of their contents and uses a *jump table* data structure to store all allowed page CF transfers as relations from a *source hash* to a *target hash*, as depicted in Figure 5.6. Since there might be multiple source hashes leading to the same target hash (when multiple pages have branches to the same target page), the jump table is constructed target hash-centered. The jump table uses a hash table array to speed up accesses by grouping hashes with the same prefix. The table stores target hashes as starting points for a list of source hashes from which the respective target page might be reached in the control flow. The jump table should normally be shared between all cores.

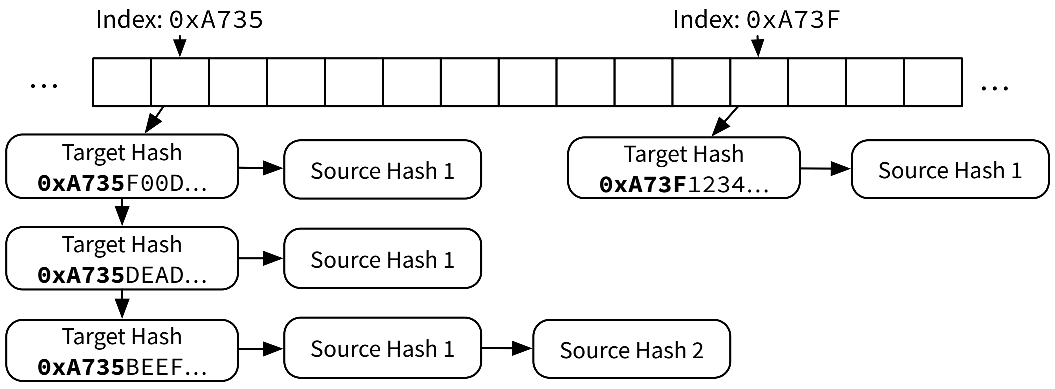

Index: 0xA735
Index: 0xA73F
...
...
Target Hash 0xA735F00D...
Source Hash 1
Target Hash 0xA735DEAD...
Source Hash 1
Target Hash 0xA735BEEF...
Source Hash 1
Source Hash 2
Target Hash 0xA73F1234...
Source Hash 1

Figure 5.6: Jump table data structure for storing allowed page-granular control flow transfers.

Our security application attaches to the tracing framework and, as soon as a page control flow transfer happens in the guest, the application retrieves the target page from the framework (see Section 5.4) and computes its hash. It then locates the hash in the jump table and checks if the allowed source hashes overlap with the source page(s) retrieved from the framework.

The application uses a *hash cache* to speed up operation by storing already computed hashes. As soon as a page is mapped writable, its cached hash is invalidated. The hash cache *must* be shared between all cores to make sure that the invalidation works correctly.

5.7.1 Jump Table Generation

A crucial challenge for the application is the initial creation of the jump table representing "normal behavior". One possibility is to employ some kind of *training phase* in which jumps are not checked against the data structure but *inserted* into it. For this, the tracing framework must be configured to run with only *one* executable page to always be able to determine the exact source page for a control flow transfer. Another method is to do a static pre-analysis of the software to be supervised and pre-generate the jump table accordingly. In both cases it can be reasonable to have different tables per in-guest process and for the kernel differentiating them using the framework provided guest context information (see Section 5.4).

The main purpose of our application is to show the usefulness of our tracing framework and measure its performance in a meaningful way. Therefore, for our prototype, we employ a simple training phase as described before. In Section 5.7.3 we discuss the real world usability of such an approach.

5.7.2 Implementation

The implementation of the page hash-based CF protection as security application in the hypervisor mainly consists of the code maintaining the jump table. For hashing pages, the application includes a minimalist SHA1 implementation. The jump table uses a hash table array indexed with the first 16 bits of the hash. The implementation furthermore employs a hash cache as an array with space for a hash value for each 4 KiB page in the 32 bits physical address space, i.e., 2^{20} hashes, and a corresponding array storing a valid-flag for each of the hashes. When a CF transfer happens, the application looks up the valid-flag of the corresponding hash cache slot and retrieves the hash or, if the hash flag is invalid, computes and stores the hash there. As soon as the page gets mapped writable, the according hash cache slot is invalidated by the application.

The prototype implementation provides a *training* mode which executes with only one executable page and initially builds up the jump table data structure. On an `HVC` call, the hypervisor can switch to an *enforcing* mode in which it checks new jumps for their validity as described before. Our implementation generates a single jump table data structure for the whole system, shared between both cores. The application adds about 800 LOC to the approximately 3000 LOC of the hypervisor with tracing framework.

5.7.3 Discussion

The page hash-based CF protection can potentially prevent control flow hijacking attacks by enforcing only pre-defined transfers between pages and at the same time ensuring their code integrity. In our attacker model in Section 5.1, we basically introduced three types of attacks: Code corruption/injection, CRAs in user space and CRAs in kernel space. First, by building and checking hashes of all code in the system and enforcing W^X, our concept is able to prevent code corruption and injection in cases where the kernel does not provide sufficient protection, e.g., with DEP. Second, with the enforcement of coarse, page-grained CFI, the concept poses strong constraints on the attacker implementing a CRA. He is forced to find all his gadgets in the set of executable pages and follow only permitted page transfers in between gadgets. As opposed to other approaches, the protection extends to kernel space and enforces the same page-grained CFI there. Hence, the attacker is constrained the same way when trying to use a CRA to elevate his privileges.

Identifying pages by their hashed contents has, besides the integrity protection, the additional advantage that the jump table can store jumps independently of the actual page address. Therefore, moving pages around in memory with unchanged content does not necessarily corrupt the information about them in the jump table, which is useful for PIC. As shown in Section 5.8, the application performs reasonably well on top of the tracing framework. Especially the hash cache is highly effective. The evaluation also shows that the complexity of the jump table resulting from running the training mode during a complete Android boot is manageable.

Nonetheless, the approach certainly has some issues. The main issue is the generation of a jump table representing the page-grained CF of the whole target platform. Such an effort only seems manageable for a typical embedded device with a static process list and configuration. For more dynamically managed systems with user-installable software it is certainly more promising to build jump tables for single processes and find a way to overlay them for enforcing the CF for the whole system. While this remains a topic for future work, the current version of the CF protection systems shows the feasibility of the concept in general. Additionally, there are some minor compatibility issues with the current implementation and its jump table generation. First, the implementation does conflict with ASLR since ASLR causes addresses to change for each run of an application which in turn changes branch targets and the corresponding page hashes. Second, the implementation is currently incompatible with JIT-compiled and self-modifying code as for both executable code is generated or modified at run-time making the generation of a jump table complicated.

Despite the issues, the page-grained CF application succeeds in its main goal of demonstrating the usefulness of the presented tracing framework for realizing novel security mechanisms.

5.8 Performance Evaluation

To get an impression of how much the prototype implementation of our framework and our example security application affect the system's performance, we conducted a series of experiments. As the main parameter in all experiments we varied the number of executable pages n. For all experiments, we chose the maximum n to be 100, equaling a window of 4 to 400 KiB of code that can be executed by the guest without generating tracing data. Despite the accuracy and usefulness of the CF data is already relatively poor for $n = 100$, taking higher values into consideration helps to show how the system generally performs for different n.

For all experiments, we ran a Linux 3.0.31 kernel and Android version 4.1.1 on the Arndale board. We disabled ASLR and the Android Dalvik JIT to prevent them from interfering with the page hash-based CF protection and to reduce variance. For all experiments, we ran the framework with the W^X protection as described in Section 5.3. As benchmarks we chose the Linux and Android boot time, CoreMark [Con] and the Antutu app [AnT].

The results of all benchmarks are depicted in Figure 5.7. The results are shown as percentage of the native performance of the benchmark, i.e., running without the framework. In the following we describe the different benchmarks in detail and discuss the results.

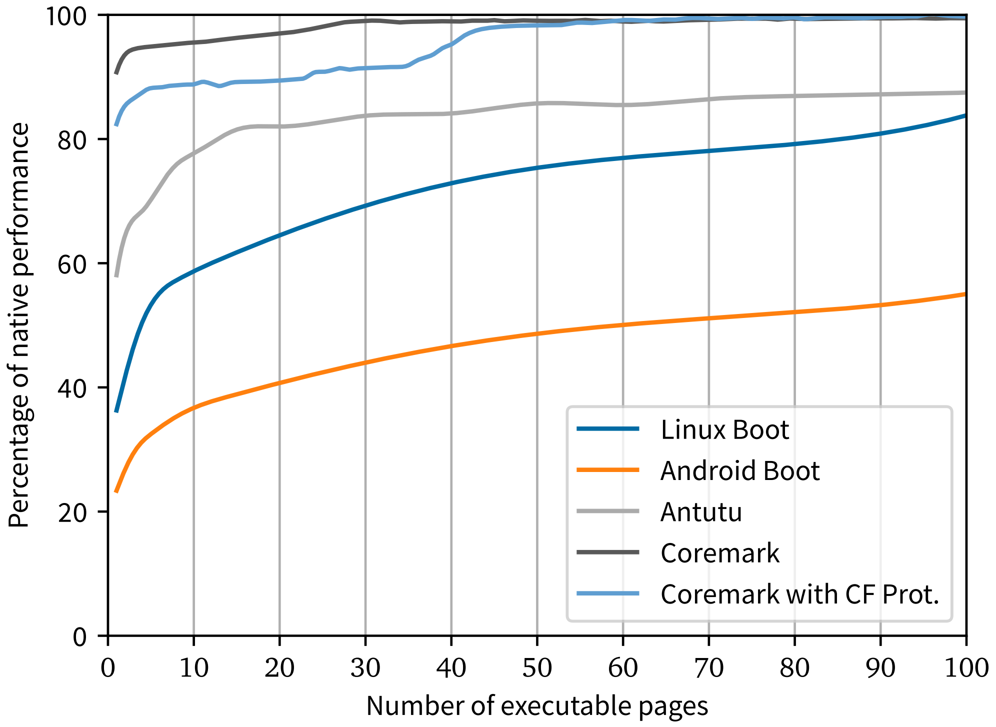


Figure 5.7: Tracing framework performance benchmark results.

5.8.1 Linux and Android Boot Time

For this benchmark, we measured the time the Arndale board running our bare framework (without security applications) requires to boot the Linux kernel and a full Android OS afterwards. For the Linux boot, we measured the time from the kernel boot start until the first userland process is started (`init`). For the Android boot time, we measured the time until the Android property `sys.boot_completed` is set to 1 by the Android framework. For both, we used the kernel message timestamps as timer by emitting a debug message to the kernel log at the described points.

In addition to the boot performance graphs in Figure 5.7, details of the boot time measurements are depicted in Figure 5.8. We primarily took measurements at points where n is a multiple of ten. For the region $n < 20$, we measured with a finer granularity as the graph changes more rapidly there. For each chosen n, we took multiple measurements to reduce the variance of the result. The graph shows the measurements together with a spline fitted graph through their averages. The native boot time, i.e., the optimum, is shown with a dashed line for both, kernel and Android boot. The results show that the boot time decreases rapidly at first when increasing n. For higher n the improvement gets less significant without vanishing completely. This is true for both, Linux and Android boot

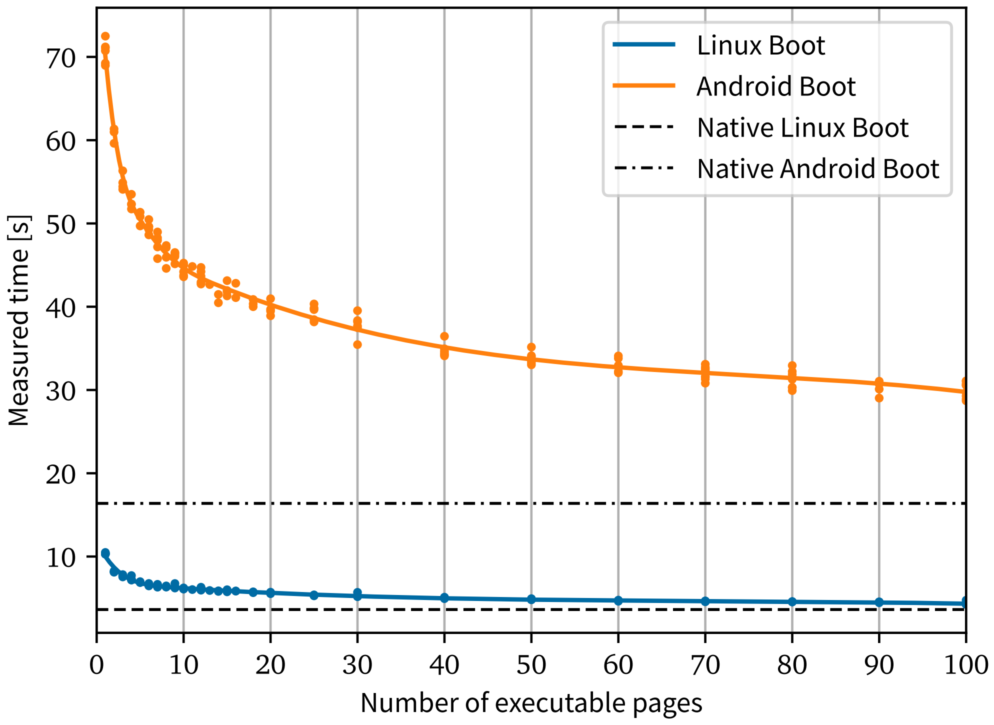


Figure 5.8: Linux and Android boot time measurements.

time. As expected, the performance impact of the framework on the Android boot time is more significant than on the Linux boot time because it has a bigger code base. Since the boot process can be seen as a worst case scenario for the framework's performance with a large code base, many different processes being started and, hence, very frequent context switches, the performance is surprisingly good for both Linux and Android boot. This confirms that a context switch into the hypervisor mode can be executed with minimal overhead on ARM platforms.

To get a better understanding how the performance relates to context switches and framework interactions, we furthermore counted the replacements of executable pages in our framework during a complete Android boot for different n accumulated for both cores. The results are shown in Figure 5.9. The maximum number of replacements is about 30 million for one executable page. The replacements decrease in a very regular manner down to less than half a million for $n = 100$. While the basic graph behavior is the same as for the boot performance, i.e., strong decline at the beginning, the graph shows a less pronounced curvature and gets less saturated for higher n. This shows that reducing the necessary replacements has a stronger effect on the performance for lower n than for higher n. This accommodates our concept as it allows using lower n and therefore yielding higher accuracy for the CF data with reasonable performance.

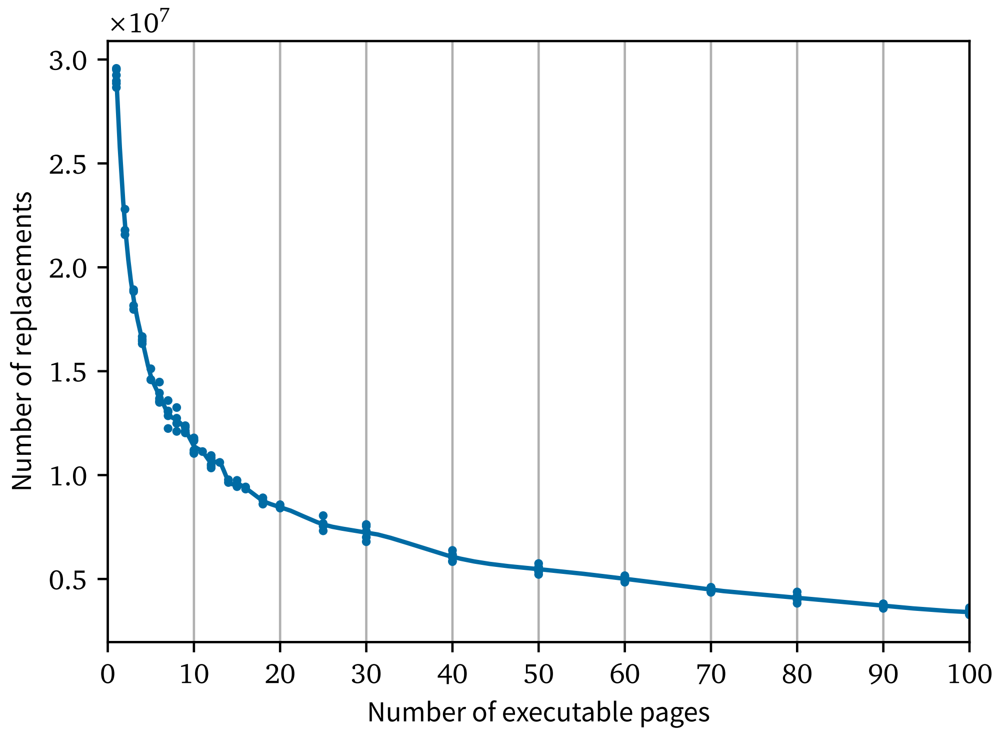


Figure 5.9: Executable page replacements during Android boot.

Figures 5.7 and 5.8 show the boot time results for the bare framework and do not include boot time measurements for the page hash-based CF protection (see Section 5.7) example application. To get an impression of the boot time with active protection, we took several measurements of the system's boot time with *one* executable page and activated learning mode, i.e., building up the jump table structure (see Section 5.7.1) in memory during boot. The results are summarized in Table 5.1. The Linux boot took 15.17 and the Android boot 185 seconds on average. This is a significant increase compared to the basic concept (see Figure 5.8), which is due to building up the jump table and more complicated locking mechanisms protecting the table and the hash cache between both cores. For other benchmarks, the overhead of the CF protection is much less (see CoreMark benchmark). The resulting jump table data structure contains about 3900 target hashes and about 77000 source hashes. This means that there are 3900 unique code pages used during the Android boot and the prototype identified 77000 possible control flow transfers between these 3900 pages. This means that, at this point, on the average only about 20 pages (equaling 80 KiB) can be reached from each single page, drastically reducing the possibilities to find the required gadgets for a successful ROP attack. As expected, code pages are not modified very often (especially with JIT disabled) and accordingly the hash cache performs very well with a miss rate below 0.005%.

Table 5.1: Key figures of the boot process with control flow protection.

Linux boot	Android boot	Target hashes	Source hashes	Hash cache miss rate
15.17s	185s	3904	77503	$< 0,005\%$

5.8.2 CoreMark Benchmark

The CoreMark benchmark [Con] is a modern Dhrystone replacement for measuring CPU integer performance with common operations like matrix manipulations and Cyclic Redundancy Check (CRC). We used version 1.01 and compiled it for multi-core measurements with the flags `-O2 -DMULTITHREAD=2 -DUSE_PTHREAD` using the Android NDK. Running the benchmark on our prototype on the Arndale board produced so much load that the kernel began throttling the CPU unpredictably from the maximum frequency of 1.7 GHz to 800 MHz resulting in higher variance for the results. Therefore, we reduced the maximum frequency to 1.6 GHz and installed a passive cooling unit on the SoC, which completely eliminated throttling. To further reduce variance, we conducted the benchmarks with a minimal Android OS, i.e., without booting the complete Android framework. We tested our bare tracing framework and the framework with CF protection application. For the latter we ran the benchmark once in learning mode with *one* executable page before switching to enforcing mode and running the benchmark for each n.

The results were already shown as percentage of the native performance in Figure 5.7. Additionally, Figure 5.10 provides an absolute view on the measurement results. As for the boot time, we ran the benchmark multiple times for each n and spline fitted a graph through the average values. The native performance is shown as dashed, horizontal line.

As expected, both the bare framework and the CF protection application perform much better in this benchmark with a much smaller code base than in the boot time benchmark. They already reach about 91% and 82% of the native performance with only *one* executable page. Both still show the characteristic steep performance increase for the first few n. The basic version reaches native performance at about $n = 30$ after a almost linear growth in the region $5 < n < 30$. A probable explanation for this is that the CoreMark code already fits into less than 5 pages. The CF protection application shows an almost constant course in the region $5 < n < 35$ before showing another increase and reaching native performance at about $n = 50$. The deceleration can be explained by the algorithm, which has to check for each replacement if a jump to the target page is allowed from any of the pages currently executable. So for one replacement with n executable pages, in the worst case, n checks in the jump table structure have to be performed.

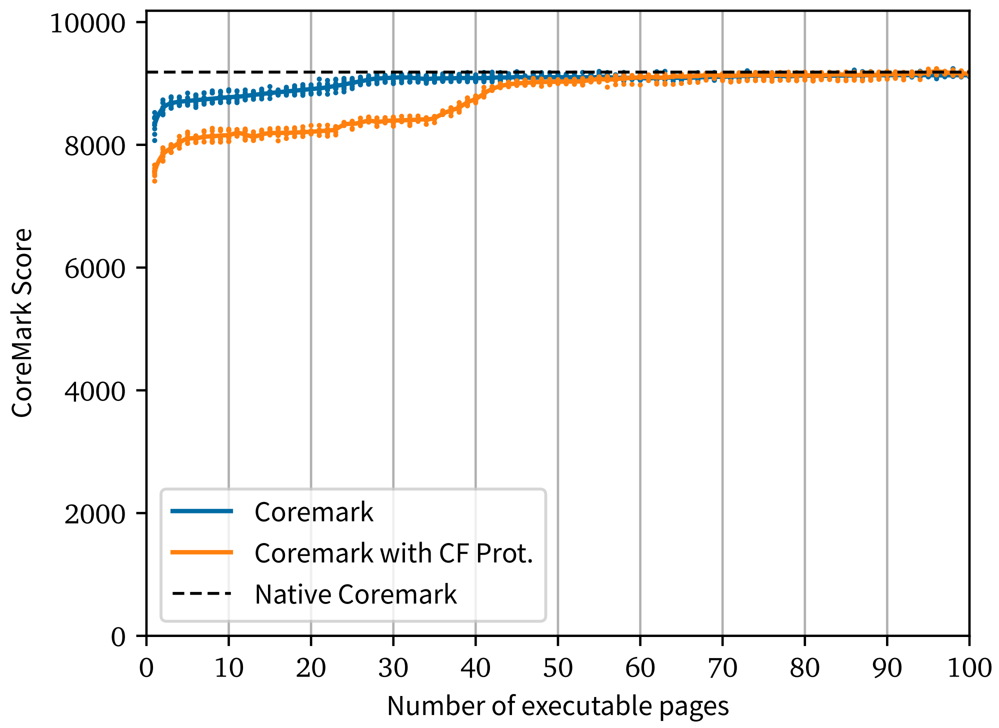


Figure 5.10: CoreMark benchmark results.

5.8.3 Antutu Benchmark

Antutu [AnT] is a widely used Android benchmark app, which tries to measure the overall performance of an Android device by testing CPU, RAM, GPU and I/O speed. In contrast to CoreMark, the Antutu benchmark runs as normal app and therefore requires the complete Android framework and a display attached to the Arndale board. Thus, it should well reflect the normal user experience for a device incorporating our framework. As for the boot time, we primarily ran tests for values of n that are multiples of ten, but chose a finer granularity for $n < 10$. For this benchmark, we only tested the performance of the bare framework without security application. As for the CoreMark benchmark, we had to limit the CPU to 1.6 GHz. The results are shown as percentage of the native performance in Figure 5.7. Figure 5.11 shows the results as absolute Antutu scores. Once again, we did several measurements per chosen n and spline fitted a graph through the average results. The native performance is indicated by a dashed, horizontal line.

The measurements show a steep performance increase for the region $1 < n < 15$. For higher n the performance gain is less distinct. Our maximum of a hundred executable pages is not sufficient to reach native performance but about 87% of it. This can be explained

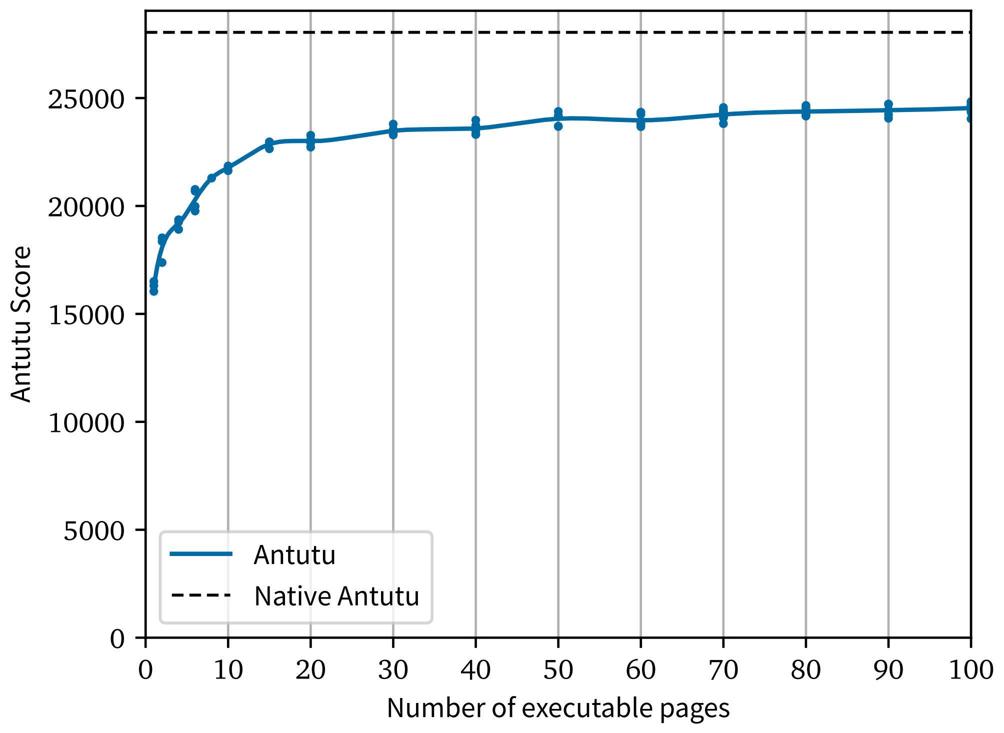


Figure 5.11: Antutu benchmark results.

with the larger code base of the benchmark, consisting of the benchmark app itself and also the corresponding Android framework parts.

5.8.4 Discussion

Our benchmarks confirm the expectation that the performance of the framework strongly depends on the code base of the tested application with the Android boot performing the worst and CoreMark the best. All benchmarks show the highest performance gain for the first few increases of n, which accommodates our use case, allowing us to reach good performance for reasonable small n values.

All in all, the performance of all benchmarks is surprisingly good. This is especially true for lower values of n where only minimal portions of code are executable at the same time and a vast number of context switches into the hypervisor and corresponding page replacements are necessary. A little more than 20% of native performance for $n = 1$ in the Android boot might sound slow but keeping in mind that this is a worst case scenario for the framework, the result is actually quite acceptable. Furthermore, our prototype implementation still can be optimized. The results show that the framework as an example for VMM-based security concepts relying on very frequent hypervisor context switches is realizable on ARM and fast enough to be used in real-world scenarios.

5.9 Summary

Our goal for this chapter was to find a way to monitor and protect code at run-time on our target platform after the initial boot-time integrity protections. Our solution should leverage logical separation to isolate the required code and data in a protected compartment, to reduce the TCB and to make the mechanism highly independent of and transparent to the specific software running on the target platform. Consequently, we decided to explore ways using hardware-assisted virtualization on modern ARM CPUs to realize our goal.

In a first step, we introduced a framework for gathering page-granular CF data transparently in a minimal ARM hypervisor. The concept utilizes the SLAT of the ARM virtualization extensions to restrict the number of executable pages in the system and to recognize control flow transfers between pages without interfering or modifying software in higher layers. We showed that the framework is able to provide a variety of tracing data to security applications running in the hypervisor. The framework can be useful to realize existing x86-based VMM security applications on ARM for a variety of use cases. Because of the extremely lightweight hypervisor context switch on ARM, the framework can also be used to realize new approaches that rely on frequent hypervisor interaction and would be too slow on x86.

As an example for such a security application, we presented the page hash-based CF protection. This application uses the framework-provided CF data to enforce a specific page-granular control flow based on hashed contents of the pages. It is therefore able to protect against different kinds of CF hijacking attacks.

We developed a complete prototype consisting of a minimal ARM hypervisor including the tracing framework and the described page hash-based CF protection application. Our detailed performance analysis for different application scenarios shows that the framework and our example application perform comparably well even in unfavorable circumstances.

Protecting Main Memory Confidentiality 6

In the previous chapters we proposed solutions to ensure our target platform's code integrity at boot-time and at run-time. Those approaches primarily protect the target platform from an attacker leveraging vulnerabilities in outward-facing processes running on the main CPU. Even with the protections in place, a physical attacker might still be able to use its immediate access to the target platform's hardware and interfaces to successfully attack the confidentiality of data in the system. A simple example is a physical attacker who removes the secondary storage of the target platform and extracts its data. While protection of such persistent data by means of Full Disk Encryption (FDE) is already very common, sensitive and private data in the volatile main memory, i.e., the RAM, remains unprotected. This includes, for example, classic secrets such as cryptographic keys and passwords but also private data like documents, pictures and others. There are different *memory attacks* which extract main memory data from a system. Examples are cold boot [Hal+09; Gut01; MS13] or DMA attacks [Boi06; BDK05; SB12; Mar+19]. Some of them might even be executed remotely, e.g., through a baseband processor with memory access [Wei12; Gol18]. Additionally, the RAM of the system typically also stores the key for the FDE so that a memory attack often also leads to full disclosure of data on the secondary storage. In this chapter, we explore ways to protect our target platform against memory attacks. Following our basic design principle of leveraging separations to reduce the TCB and increase transparency, we want to find a way that realizes memory encryption transparently from a privileged level in the target platform.

There is a variety of related work proposing different concepts to protect sensitive data in RAM against memory attacks. Several approaches try to specifically protect keys, e.g., for FDE, normally stored in RAM, for example, by moving them from the memory into special processor registers [MFD11; GM13; Sim11] or into higher privilege levels [MTF12]. While those approaches protect keys, they do not protect other sensitive data in RAM. There are several approaches for actual main memory encryption. Most of them rely on custom hardware [GLQ99; DK06; Lie+00; Gut99; Suh+03] and are therefore expensive and difficult to realize. Other software-based RAM encryption approaches have real-world usability issues [HT13] or encrypt only selected processes [CDC08; Col+15] or parts of selected processes [Göt+16b]. Existing hypervisor-based approaches [YS08; Che+08] focus on a different attacker model and do not prevent cold boot, DMA or similar physical attacks.

To overcome downsides of existing approaches, we propose a combination of two novel mechanisms, one for run-time encryption and one for suspend-time encryption. First, we introduce TransCrypt, a concept for transparent, run-time main memory encryption using a minimal hypervisor. As for our hypervisor-based monitoring framework, discussed in Chapter 5, we use hardware-supported virtualization mechanisms to transparently restrict the guest's memory access to a small and dynamically changing subset of physical RAM pages. In contrast to our monitoring framework, we not only restrict the *execution* of pages but *all* types of accesses, encrypting and decrypting pages entering or leaving the set on-the-fly. This ensures that only a small part of memory containing the most recently accessed pages is unencrypted, while the major part remains securely encrypted. There are special pages, such as the ones shared with peripheral devices via DMA, for which an encryption might lead to malfunction. Hence, we introduce a mechanism to transparently detect those pages to temporarily exclude them from encryption. Being located in the hypervisor, TransCrypt has several advantages. It does not require any changes to the guest, i.e., the kernel and user space software. It is almost completely agnostic to the guest OS and encrypts not only user space process memory but also kernel code and data. Using a custom and minimal hypervisor, TransCrypt provides a small, less error-prone code base. While the general concepts of TransCrypt are applicable to all CPU architectures with support for virtualization, our target platform typically is an ARM-based system and TransCrypt therefore focuses on the ARM architecture [ARM14; ARM17] and the ARM Virtualization Extensions [MN11]. Like our virtualization-based monitoring framework presented in the previous chapter, TransCrypt relies on frequent switches into the hypervisor, which we showed to be especially efficient on ARM platforms in Section 5.8.

Additionally, we propose the combination of TransCrypt with a mechanism for memory encryption during suspension, i.e., temporary sleep, of a system or process. The complementary suspend-time encryption scheme is realized in the Linux kernel suspend subsystem and forces each process to encrypt itself immediately before being frozen for suspension. This makes the mechanism fast and efficient and ensures that the encryption does not interfere with the normal function of the target device. Since the approach, in contrast to TransCrypt, does not encrypt and decrypt memory constantly during normal operation, it can utilize stronger means of protection for its keys. Therefore, the suspend-time encryption offers a second level of protection against stronger attackers for suspended processes as a supplement to TransCrypt. Being located in the Linux kernel, the mechanism can be combined with hypervisor-based TransCrypt in several ways. In a system with multiple guests running on top of the TransCrypt hypervisor, it can be used to encrypt an inactive guest. Furthermore, the concept supports partial encryption of a guest in form of suspended containers in a OS-level virtualization (see Section 2.2 and Chapter 3) scenario.

Finally, as comparison to our software-based approach, we analyze the security of a recent hardware-based memory encryption scheme, namely AMD SEV. In traditional systems and architectures, including our target platform's, less privileged layers have to fully trust more privileged layers. TEEs are one development aiming to remove some of the necessary trust in OSs and hypervisors by moving sensitive functions into a parallel, isolated execution environment. AMD SEV promises to provide confidentiality for full VMs towards the hypervisor. In our analysis, we use techniques similar to the ones we used for our virtualization-based tracing framework (see Chapter 5) and show that the hypervisor, even without directly accessing the VM's memory, is still able to gain access to the VM's data. Our analysis emphasizes that an architecture that removes trust from privileged layers, contrasting our generic architecture (see Chapter 3) in which higher layers are more trusted and handle more sensitive data, is difficult to realize.

The chapter is organized as follows. We first motivate our memory encryption scheme by discussing an attacker model in Section 6.1 and related work in Section 6.2. Then we introduce our run-time memory encryption TransCrypt in Sections 6.3, 6.4 and 6.5. In Section 6.6 we shortly introduce our suspend-time encryption concept and discuss how it can be combined with TransCrypt. Finally, we present our attack to extract plain text memory from AMD SEV in Section 6.7 and summarize the chapter in Section 6.8.

Parts of this chapter have been published in [HHW17a] and Sections 6.6 and 6.7 are in parts based on research conducted for joint work publications [HHW17b; Hub+17; Hub+18] and [Mor+18], respectively.

6.1 Attacker Model and Memory Attacks

In Section 3.3, we introduced our generic attacker model. Based on the attacker types defined there, in the following, we specify the exact capabilities of an attacker against which our memory encryption scheme is designed.

The primary, abstract characterization of our attacker is his ability to read parts or all of the main memory without involving software, especially privileged system software, on the target platform's CPU. In the following, we refer to such attacks as *memory attacks*. This attacker complements the remote attacker of the previous chapters, who is able to attack software processes running on the main CPU of our target platform. A memory attack can be conducted using different techniques.

First, a physical attacker with direct access to the target platform can execute a *cold boot* attack [Hal+09; Gut01; MS13; Tau+15]. In a typical cold boot attack, the attacker physically cools down the memory and moves it to another device in order to extract its data. Because of the remanence effect, most of the RAM contents are still intact and can be read by the attacker. In most mobile devices, the main memory is non-removable. In such a scenario, an attacker might be able to reboot the original device into a minimal privileged memory dump tool to extract data [MS13; Tau+15].

Second, an attacker can execute a DMA attack [Boi06; BDK05; SB12; Mar+19]. In such an attack, the attacker controls a peripheral device in the target to directly access the main memory via DMA without involving the CPU. Our attacker is capable of executing different DMA attacks including attacks via physical peripheral connection and via remote control of a DMA device in the target platform. Examples for possible target peripherals include common devices such as GPUs and Network Interface Controllers (NICs) but also mobile-specific targets such as the baseband processor in a smartphone [Wei12]. The latter often shares parts of the main memory with the application processor and provides an independent remote connection as possible attack vector. For a successful remote DMA attack targeting a device other than the one used for the attack communication, the attacker first has to achieve RCE on the target platform. Hence, for this chapter, we exclude such attacks and leave their defense to run-time integrity protections such as the one described in Chapter 5. We assume that our attacker is able to successfully execute DMA attacks despite the possible presence of IOMMUs or SMMUs controlling the access of peripheral devices to memory, for example, because of a misconfiguration or hardware limitations. Recent attacks [Mar+19] show that such an assumption is reasonable.

None of the attackers is able to break cryptographic primitives. All attackers access the system from outside the CPU and are therefore not able to directly influence workload on the CPU. In Section 6.5.1, we evaluate TransCrypt against our attacker model.

6.2 Related Work

A variety of research approaches exist that try to protect all or specific parts of main memory data. *CPU-bound encryption* concepts focus on removing encryption keys from RAM. Multiple approaches [MFD11; GM13; Sim11; MTF12] use special CPU registers to store a key so that it is never stored in RAM. While these concepts protect keys, e.g., for FDE, they leave the rest of the RAM contents unprotected.

Hardware-based RAM encryption concepts, such as *XOM* [Lie+00], *Aegis* [Suh+03], *CryptoPage* [DK06] and others [GLQ99; Lie+00; Gut99], focus on different aspects of designing processors with encrypted memory and encrypted memory buses. As they require changes to the hardware, they are much more expensive and harder to realize than our software-based approaches.

As introduced in detail in Section 2.3, ARM TrustZone [AF04; ARM14; ARM17] and Intel SGX [McK+13] are hardware security extensions of current CPUs. TrustZone and SGX are designed to allow certain small and specially designed applications to run in a secure CPU mode, isolated from normal execution. Only SGX provides hardware-based memory encryption for the memory regions of its *secure enclaves*. ARM TrustZone does not provide a solution for memory encryption, but typically protects its software's data by storing it in SRAM, highly integrated within the ARM SoC, but unencrypted. Hence, both TEEs are not able to provide a solution for encrypting larger parts of memory for normal applications and the kernel. Nonetheless, TrustZone can be seen complementary to our concept as it provides a way to secure the key for our memory encryption scheme.

AMD SME [KPW16] (see Section 2.2.3) and its future Intel counterpart Total Memory Encryption (TME) [Int17] are hardware-based solutions for full memory encryption. While those approaches are able to solve the problem of memory attacks efficiently and permanently, only AMD's solution is actually available at the time of writing. Intel TME is in an early specification phase and for ARM, the most important architecture for mobile devices, no hardware-based solution is in sight. Typically, the key management of such solutions is quite inflexible, preventing, for example, a physical separation of encryption keys for memory contents of suspended components, as done in Section 6.6. Furthermore, as our analysis of AMD SEV in Section 6.7 emphasizes, an extensive security analysis of those hardware features is necessary. AMD SEV is a hardware encryption feature that builds on SME and enables encryption of VMs, protecting their data in memory, supposedly even from a possibly malicious hypervisor. As our analysis shows, this goal cannot be achieved with current versions of SEV.

Henson et al. [HT13] realize RAM encryption in software on an ARM Cortex-A8 platform using a microkernel in the SRAM, sometimes also called On-Chip RAM (OCRAM) or Internal RAM (iRAM), which is often part of ARM TrustZone platforms. As discussed in Section 2.3.1, the OCRAM is separated from the normal RAM and, thus, normally invulnerable to classic memory attacks. Henson et al. dynamically swap and en-/decrypt processes between RAM and OCRAM using the hardware crypto accelerator in the SoC. As the OCRAM is very small and basic features such as virtual memory are not supported, the concept suffers from performance issues and is very hard to combine with real applications such as a normal Android OS environment.

The approaches by Chen et al. [CDC08] and *Sentry* by Colp et al. [Col+15] both encrypt memory of selected, "sensitive" processes. These processes are decrypted into on-SoC memory, either into locked cache (*Cache as RAM*) or into OCRAM. In case of *Sentry*, processes are encrypted when the device is suspended and decryption into cache or OCRAM is only done for processes that have to run in background. *Sentry* proposes an Advanced Encryption Standard (AES) implementation running completely out of OCRAM. Both approaches suffer from the fact that cache locking is an optional legacy feature in newer ARM architectures [ARM14; ARM17]. In contrast to our concepts, both approaches only encrypt selected processes and do not cover kernel space with their encryption. They require changes to the kernel or user space while TransCrypt is completely transparent. Furthermore, *Sentry* only encrypts memory when the system is suspended.

CleanOS [Tan+12] is a mechanism implemented in the Android framework. It defines special *sensitive data objects* which are encrypted if not actively used for a certain amount of time (*idle eviction*). The encryption keys are managed in the cloud, meaning that they are removed from the device after encryption and requested from a remote server as soon as the encrypted object is needed again. In *CleanOS*, not all RAM is encrypted and the concept suffers from the trust required towards the cloud infrastructure. Furthermore, *CleanOS* only works if the protected device is online.

CryptKeeper by Peterson [Pet10] extends the memory hierarchy by dividing the RAM into a smaller unencrypted and a bigger encrypted part. As in TransCrypt, only a small part of recently accessed data in RAM is left unencrypted. In contrast to TransCrypt, *CryptKeeper* is realized in the Linux kernel and can therefore not protect kernel memory itself, is not transparent or agnostic to the OS and does not provide a concept for securing the encryption key.

RamCrypt [Göt+16b] modifies Linux memory management to realize encryption of least-recently accessed pages. *RamCrypt* is a run-time, kernel-based mechanism which, in contrast to TransCrypt, is activated for specific processes only. *RamCrypt* is therefore not able to encrypt the kernel itself. Furthermore, it does not encrypt file-backed data in memory, including code and possibly confidential memory-mapped files.

Overshadow by Chen et al. [Che+08] and an approach by Yang et al. [YS08] both, like TransCrypt, use encryption from a privileged hypervisor to protect memory contents. But instead of protecting the system from physical attacks, such as cold boot, their goal is to protect user space data from an attack from a malicious guest OS. Hence, both systems rely on storing keys or even unencrypted versions of encrypted pages [YS08] in RAM. This makes them highly susceptible to memory attacks prevented by our memory encryption concepts. Furthermore, both approaches are x86-based, specific to Linux guests, do not encrypt guest kernel data and require intercepts from the hypervisor on *every* context switch in the guest, e.g., on interrupts and syscalls. *Overwshadow* and Yang et al. use completely software-based hypervisors for which especially syscall intercepts cause much less *relative* overhead than in a more recent virtualization scenario. In a modern, hardware-assisted virtualization, such as the one TransCrypt uses, such intercepts might not even be architecturally supported[1] and would cause a huge relative overhead.

HyperCrypt [Göt+16a] is a hypervisor-based approach for RAM encryption developed concurrently to TransCrypt. In contrast to TransCrypt, the approach targets x86 and selectively focuses on server applications avoiding challenges, such as GPU support, posed by targeting a full-fledged end user device like an Android smartphone. Furthermore, *HyperCrypt* does not provide a dynamic, guest-transparent way to detect DMA pages. Instead, it relies on paravirtualization for DMA page identification. Therefore, it requires changes to the guest as well as driver support in the hypervisor, increasing its complexity and code size. Additionally, *HyperCrypt* does not offer a solution for efficient multi-core design and dynamic adaption of the unencrypted memory window size.

Hypnoguard [ZM16] is a suspend-time memory encryption approach. Developed concurrently, it shows some similarities to our suspend-time encryption scheme (see Section 6.6).

[1]In the ARMv7-A architecture, routing of general exceptions, such as syscalls, to the hypervisor is "unpredictable" if guest-controlled first level address translation is active [ARM14].

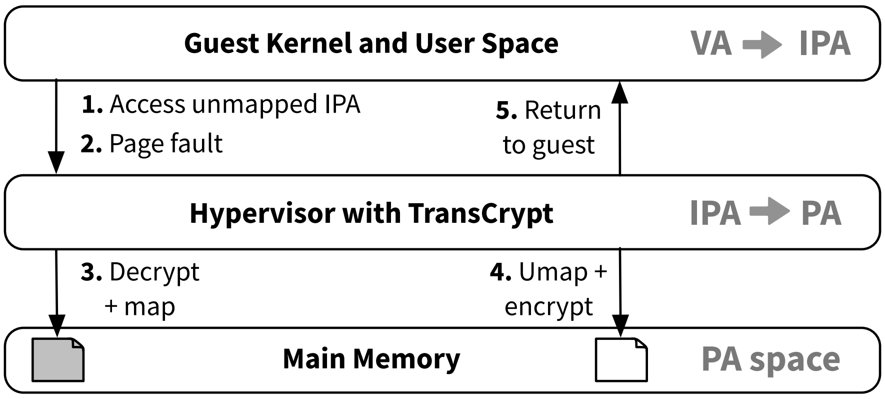


Figure 6.1: TransCrypt memory encryption concept.

In contrast to our approach, *Hypnoguard* is not realized in the Linux kernel but as low-level system software. While this reduces the TCB and gives better independence from a specific OS, it increases *Hypnoguard*'s hardware dependency and makes the approach less portable than ours. Furthermore, being located in the Linux kernel, our suspend-time concept is able to encrypt groups of processes independently. Finally, a suspend-time encryption on its own without a complementing run-time solution such as TransCrypt can protect suspended devices and processes but not secrets in RAM owned by running processes.

Summarizing, our run-time and suspend-time memory encryption concepts provide novel approaches for their respective scenarios. Furthermore, none of the previous works combines run-time and a suspend-time encryption.

6.3 TransCrypt: Run-time Main Memory Encryption

In the following, we introduce TransCrypt, our mechanism for run-time main memory encryption. TransCrypt uses the SLAT of hardware-supported virtualization to transparently restrict the guest to a small set of unencrypted physical memory pages, the *Unencrypted Page Set (UPS)*, while keeping the rest encrypted (with some exceptions as discussed in Section 6.3.8). TransCrypt dynamically encrypts and decrypts pages based on guest accesses and the resulting page faults. This basic idea is illustrated in Figure 6.1. On a SLAT page fault the corresponding physical page is decrypted and mapped while unmapping and encrypting another page.

In the following, we shortly discuss specific hardware requirements of TransCrypt and introduce details of the TransCrypt architecture and its page encryption mechanism. Furthermore, we discuss the size of the UPS and multi-core operation.

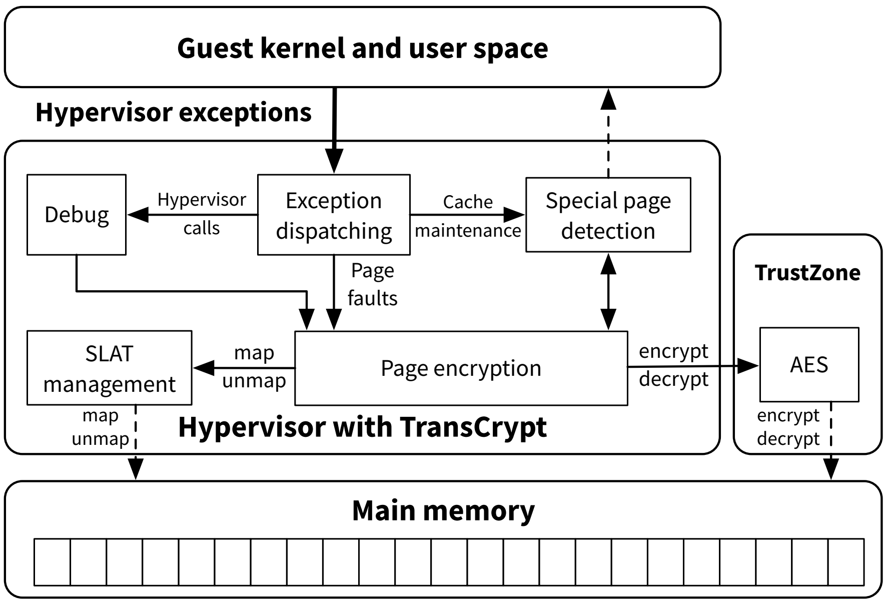


Figure 6.2: Architecture of the TransCrypt hypervisor.

6.3.1 Hardware Requirements

As for our monitoring framework (Section 5.3.1), the most important hardware feature for TransCrypt is the SLAT, which introduces an additional level of translation for all addresses accessed by the guest (see also Section 2.2.2). The additional translation allows TransCrypt to transparently control access to all physical memory building the basis for the encryption concept.

Another important virtualization feature for TransCrypt is the ability to trap certain privileged instructions, i.e., to automatically jump into the hypervisor before executing them. As described in Section 6.3.8, TransCrypt utilizes this feature to realize a technique for recognizing memory pages used for DMA.

6.3.2 Architecture

A system using TransCrypt basically consists of the guest running in kernel and user space and the encryption components running in hypervisor mode and partly in a secure enclave mode such as the ARM TrustZone to secure the encryption key as described in Section 6.3.5. Like our monitoring framework (see Chapter 5), TransCrypt is designed as part of a minimal, custom hypervisor only implementing SLAT management for a single guest. The detailed architecture of TransCrypt is depicted in Figure 6.2. Exceptions from the guest are initially handled in the *exception dispatching module*. HVCs are only included for debugging purposes and are forwarded to a *debug module*, which allows analyzing page statistics and changing the size of the UPS. Traps of certain cache maintenance operations are forwarded to the *special page detection module* where they are analyzed to determine if a page is used for DMA and must therefore be left unencrypted as detailed in Section 6.3.8.2. Page faults are the most frequent and important exceptions handled by TransCrypt. They are forwarded to the *page encryption module*, which uses the special page detection module to determine if a page should be encrypted and uses the *SLAT management* and *AES* modules to map and encrypt pages transparently for the guest. The AES module is located in the TrustZone TEE or secured with a comparable key storage mechanism as discussed in Section 6.3.5.

In the context of our general architecture introduced in Chapter 3, TransCrypt is part of the hypervisor layer on the target platform. It runs alongside the other hypervisor-based mechanisms, i.e., especially the monitoring framework (see Chapter 5). As part of the hypervisor, its boot-time integrity is ensured together with the other components by the mechanisms described in Chapter 4.

6.3.3 Definitions and Initialization

As explained in Section 2.2, in a hardware-virtualized system, the guest controls its own translation from VAs to IPAs for accessing memory while the hypervisor controls the SLAT from IPAs to PAs overlaying memory attributes given by the guest. Despite the architectures often providing different configurable page sizes for guest translation and SLAT, TransCrypt uses the same size to have the same access granularity as the guest. Before the guest runs, TransCrypt reserves memory for its own operation and restricts the guest from accessing it. For the N memory pages allocated to the guest, we define two basic sets:

Mapped Page Set (MPS). This set contains all pages that are currently mapped in the SLAT and therefore unencrypted and available to the guest without further intervention by TransCrypt.

Special Page Set (SPS). This set contains all pages that have been identified as *special* and, hence, should not be encrypted even when they are unmapped, which is the case, for example, for DMA pages. Details regarding special pages are discussed in Section 6.3.8.

Together, the sets contain all unencrypted guest pages. Hence, their union constitutes the Unencrypted Page Set (UPS):

$$\text{MPS} \cup \text{SPS} = \text{UPS}$$

All pages that are not part of the UPS are encrypted. We define a maximum size M for the MPS. M is the maximum number of pages accessible to the guest at any time.

TransCrypt starts with all pages unmapped ($|\text{MPS}| = 0$) and unencrypted ($|\text{UPS}| = N$). Therefore, by definition, all pages are special ($|\text{SPS}| = N$) before being evaluated and mapped for the first time.

6.3.4 Basic Mechanism

After initialization, TransCrypt hands execution to the guest. Since the MPS is initialized to contain no pages, this immediately leads to an instruction fetch page fault. The following steps are repeatedly executed to realize the memory encryption (see also Figure 6.1):

1. The guest accesses a page $p_{in} \notin$ MPS either by executing it or by reading or writing data from or to it.
2. The access triggers a page fault into TransCrypt. If $p_{in} \notin$ SPS, TransCrypt decrypts it.
3. TransCrypt determines if p_{in} is special (see Section 6.3.8) in which case it adds it to the SPS. Then it maps p_{in} to the guest, i.e., adds it to the MPS.
4. If the MPS does not exceed its maximum size, execution is returned to the guest. Otherwise, i.e., if $|\text{MPS}| > M$, a page $p_{out} \in$ MPS is selected for eviction using a specific eviction mechanism discussed in the following.
5. If $p_{out} \notin$ SPS TransCrypt encrypts it. Then p_{out} is unmapped, i.e., removed from the MPS, and execution is returned to the guest.

There is no differentiation between executable pages and data pages and both are equal candidates for encryption. All steps are completely transparent to the guest and do not require any explicit commands or support by the guest. The concept is therefore completely agnostic to the guest OS. It is furthermore not specific to the ARM architecture. Depending on the target platform, detecting special pages requires the use of architecture or OS specific mechanisms as described in Section 6.3.8.

As mentioned before, if the MPS exceeds its maximum size M, a page must be evicted. Since the guest is able to access pages in the MPS without TransCrypt being able to intervene, we do *not* have actual information which page was accessed least recently and should therefore be the first candidate for eviction. The next best eviction candidate is the page that we least recently mapped to the guest. We therefore introduce an eviction algorithm we call Least Recently Mapped (LRM), similar to a typical Least Recently Used (LRU) but with mapping time as basis for deciding which page to evict next. Section 6.4 discusses how we implement this algorithm with constant complexity. Figure 6.3 illustrates the basic encryption mechanism and its relation to the LRM page eviction. The top part of the figure shows the guest-controlled translation from VAs to IPAs and the active mappings in form of arrows to pages in IPA space. On the bottom, the figure shows the TransCrypt-controlled translation from IPAs to PAs. The figure depicts the system just at the beginning of handling a SLAT page fault in which the most recently accessed page is decrypted and mapped while the page that least recently entered the MPS is encrypted and unmapped, i.e., removed from the MPS. In the example, recent accesses by the guest have added three pages to the MPS, which are therefore also unencrypted ($|\text{MPS}| = M = 3$). Furthermore, three pages

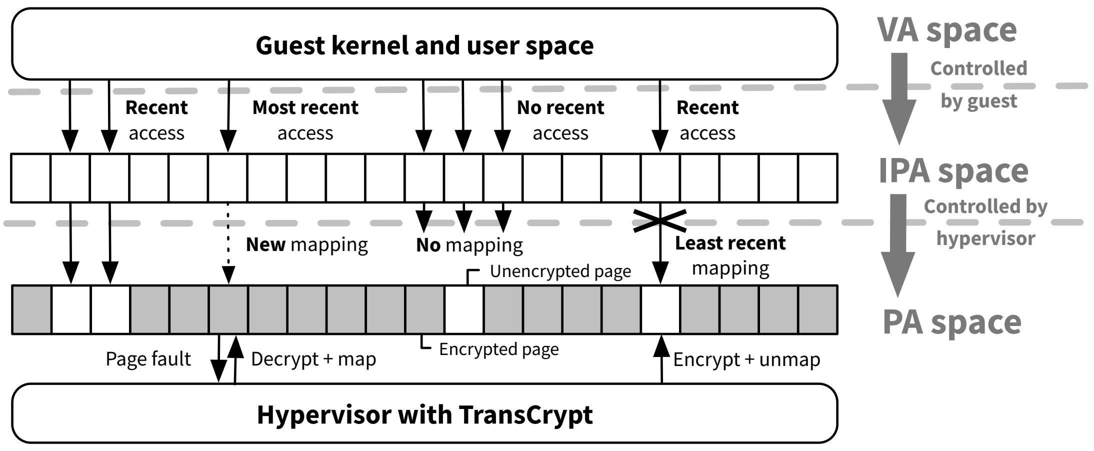


Figure 6.3: TransCrypt encryption and translation mechanism.

are mapped in the guest but not part of the MPS, the normal state for most of the pages in the system. One of these pages is part of the SPS (|SPS| = 1; |UPS| = 4) and hence unencrypted despite the fact that it is not part of the MPS and that it can currently not even be accessed by the guest. Such a state is important, for example, for DMA pages as discussed in Section 6.3.8. As soon as the page is accessed again by the guest, its special status is reevaluated. This is important since the page could be re-purposed by the guest kernel as normal, non-DMA memory.

6.3.5 Page Encryption

For the actual encryption of a page, several aspects have to be considered, including key management, encryption algorithm, mode choice, and Initialization Vector (IV) generation.

Key management. The key used for the actual page encryption *must not* be located in memory itself because we assume an attacker to be able to access all physical memory (see Section 6.1). To solve this problem, our concept relies on a secure execution environment such as the ARM TrustZone [AF04; ARM14; ARM17] where we can securely execute cryptographic operations and store the key at run-time, either in secure OCRAM or in a cryptographic coprocessor. This is not a real conceptual constraint since TrustZone is almost always present in current ARM application processors and OCRAM is a crucial component to all TrustZone platforms. For systems where no crypto coprocessor is available, Colp et al. describe how to securely implement AES [Col+15] in software only using OCRAM. Since RAM data is not persistent, the key can be randomly generated at each system reset. Hence, a concept for persistent key storage is not necessary. If, although unlikely, a TrustZone-based or similar solution is not possible on our target platform, we can store the TransCrypt key using *CPU-bound encryption* schemes [MFD11; GM13; Sim11; MTF12; Göt+16b] already introduced as related work in Section 6.2.

Encryption algorithm. The pages are encrypted symmetrically with AES. To prevent leaking information about the encrypted data, same content blocks must not result in the same ciphertext blocks. Hence, an encryption mode with IV such as Cipher Block Chaining (CBC) must be used. Page encryption runs exclusively and uninterruptibly on the core that accessed the page (see Section 6.3.7), so that, normally, no performance can be gained by choosing a highly parallelizable mode. The choice might also be made dependent on hardware crypto accelerators available on the target platform. While Authenticated Encryption with Associated Data (AEAD) modes can conceptually be used, their authentication is not necessary in terms of our attacker model, only assuming reading memory attacks, and adds performance overhead, since it requires the hypervisor to handle authentication data.

IV generation. Each page must be encrypted using a different IV to ensure that same content pages result in different encrypted pages. An obvious choice for the IV is the physical page base address. In FDE theory there is an attack on such predictable IVs in combination with CBC known as *Watermarking Attack*. In this, an attacker with write access to the disk but without access to the underlying encryption secrets creates patterns, the watermarks, on the disk, which he can then detect on the encrypted disk without knowledge of the key. If a malicious process or guest is able to write to consecutive memory pages, despite being very unlikely, such an attack might also be possible in RAM and could, for example, be used to detect the existence of watermarked data in an extracted, encrypted RAM image. We thus propose the usage of encrypted base addresses as IVs for vulnerable modes, comparably to Encrypted Salt-Sector Initialization Vector (ESSIV) [Fru05] for FDE.

6.3.6 Unencrypted Page Set Size

The maximum size M of the MPS is the main configurable value in the TransCrypt concept. Increasing M increases the number of pages the guest may access at the same time and therefore also increases the size of the UPS. This, in turn, decreases security of the system as more physical pages are unencrypted at any time. Thus, M acts as a configurable trade-off between performance and security and allows fine-tuning the system to the needs of the specific application. The actual impact of varying values of M is evaluated in Section 6.5 on the basis of our prototype implementation. Note that the size of the SPS is *not* configurable since the number of special pages is a system property and depends on the target platform's specific hardware and OS, e.g., on how many pages are used for DMA with the GPU. We propose two ways for configuring M:

Static MPS size. For this approach, M is predetermined and fixed to a certain value that fits the system's needs, especially regarding the acceptable performance overhead. This policy can be extended by allowing the guest to configure M depending on the current workload via a hypervisor call to TransCrypt. Restricting configurable values of M to certain bounds prevents abuse of the feature, e.g., for disabling the encryption.

Dynamic MPS size. As discussed in Section 6.1, TransCrypt is a defense against attackers who are not or only indirectly able to influence the workload of the system. Furthermore, typical mobile systems spend most time in idle states. Thus, we propose a mechanism for dynamically adapting M at run-time, balancing the performance loss for higher workloads by reducing the encrypted part of the memory. For that, TransCrypt can observe the time offset between SLAT page faults and adapt M to reach a specific page fault frequency. Additionally, the system should still provide a configurable hard upper bound for M to make sure that a certain amount of memory is always encrypted, independent of the workload. For page fault i occurring at time t_i, the new MPS size M_{i+1} can be dynamically calculated from the current size M_i using the following equation:

$$M_{i+1} := M_i + C\left(\frac{1}{f} - \frac{t_i - t_{i-m}}{m}\right)$$

In the equation, f [1/s] denotes the desired page fault frequency and C [1/s] configures the number of pages by which M is increased or decreased for each second the difference between two consecutive page faults deviates from the desired frequency. The configurable constant value m determines how sensitive the system reacts to quick changes in the workload.

6.3.7 Multi-core Design

On a multi-core system, all cores must have the same view on physical memory. It must be ensured that cores only have access to unencrypted pages to avoid corruption of the guest. This means that all cores must share one SLAT table. Hence, the MPS, SPS and UPS are all shared between the cores and when, for example, a core adds a page to the MPS it can be accessed by all other cores. Consequentially, synchronization mechanisms between cores are necessary. As these can significantly impact performance, they must be designed carefully.

On the one hand, synchronization is required for changing state or content of pages, for example, when encrypting a page. On the other hand, synchronization is required for keeping track of the MPS in order to realize page eviction. To almost eliminate all lock contesting between cores, we propose the following scheme. We introduce fine-granular locks, one for each physical page, which a core must acquire to change the page's content or state. Furthermore, we split the responsibility for the MPS between the cores, which

Listing 6.1 TransCrypt page fault handler.

```
State:
    $\mathrm{MPS}_c$: Set of pages globally mapped by core $c$
    SPS: Set of special pages
Input:
    $p_{in}$: The page the fault was triggered on
    $c$: The ID of the core the fault was triggered on
procedure HANDLE PAGE FAULT($p_{in}$, $c$)
    Acquire lock for $p_{in}$
    if $p_{in} \notin$ SPS then
        Decrypt $p_{in}$
    else
        Remove $p_{in}$ from SPS
    end if
    if $p_{in}$ is special (see Section 6.3.8) then
        Add $p_{in}$ to SPS
    end if
    Add $p_{in}$ to $\mathrm{MPS}_c$
    Release lock for $p_{in}$
    if $|\mathrm{MPS}_c| \leq M/\text{Total cores}$ then return end if
    Get LRM page $p_{out}$ from $\mathrm{MPS}_c$
    Acquire lock for $p_{out}$
    if $p_{out} \notin$ SPS then Encrypt $p_{out}$ end if
    Release lock for $p_{out}$
end procedure
```

allows us to avoid *one global* and therefore contested lock. Then, each core manages $M/\text{Number of cores}$ page slots in the MPS. For core i, we call the resulting subset MPS_i. A core can still access *all* pages in the MPS, also the ones mapped by other cores, but is only allowed to evict and map pages in MPS_i. A page in any MPS_i will *not* trigger another page fault on *any* core until evicted. As a result, the different MPS_i are *always* disjoint. As soon as a core has filled its MPS_i slots, it starts evicting pages even if other cores still have capacities in their MPS part. For the eviction, the core only chooses from its own MPS_i for which it does not need a lock. Listing 6.1 summarizes the encryption concept including multi-core handling as a pseudocode SLAT page fault handler implementation.

6.3.8 Special Pages

We define *special pages* as pages in the physical address map of a system for which an encryption might lead to malfunction of the system. We basically identified two physical memory types that fall into this category: Memory-mapped devices, i.e., *device memory*, and memory shared with devices, i.e., DMA memory. Recognizing pages of these categories and adding them to the SPS (see Section 6.3.3) to temporarily exclude them from encryption is crucial to ensure system functionality.

6.3.8.1 Device Memory

Not all addresses in the physical address space of a typical system refer to main memory. Some parts of the address space are not used at all and others are used as device registers, i.e., for memory mapped I/O with peripheral devices. For these memory regions, an attempted encryption would lead to malfunctions in the system. We identified two methods for handling device memory. First, we can analyze the guest-controlled address translation and recognize a page as special if the guest maps it as *device memory* type, i.e., for example, as *non-cacheable*. An advantage of this dynamic approach is that it does not require any prior knowledge about the location of device memory in the physical address map of the system.

The other approach is to statically analyze the physical address map of the system before run-time and simply generate fixed and always active mappings for all memory regions that do not refer to main memory. This solution has performance advantages compared to the dynamic approach. It also has the advantage that non-memory pages are completely removed from all operations and from our page sets described in the previous section and do not clutter the SPS. We combine both approaches in our prototype as described in Section 6.4.3.

6.3.8.2 Memory Shared with Devices

Normal memory might be shared with peripheral devices that have access to the RAM via DMA. A prime example is the system's GPU, which receives large amounts of data for rendering content on the screen from the CPU via shared memory regions and DMA. When TransCrypt unmaps and encrypts those shared pages, they might still be accessed by devices using DMA. As these devices are not aware of the encryption, accessing the pages will usually lead to malfunctions. Hence, it is crucial for correct system functionality that TransCrypt is able to recognize DMA pages to add them to the SPS and, hence, temporarily exclude them from encryption, as discussed in Section 6.3.4.

While devices might be able to access main memory, they are usually unable to access the cache hierarchy of the system's CPU. Therefore, cache coherency must be ensured when communicating with devices via DMA. This behavior is leveraged by the TransCrypt hypervisor to recognize DMA pages. In the following we discuss three basic types of DMA transfers and how TransCrypt detects them from hypervisor mode:

Always Coherent. For this type of DMA, CPU and device are always guaranteed to have the same view on their shared memory. Coherent DMA uses buffers on pages that have special attributes making them *cache coherent automatically*. They might be

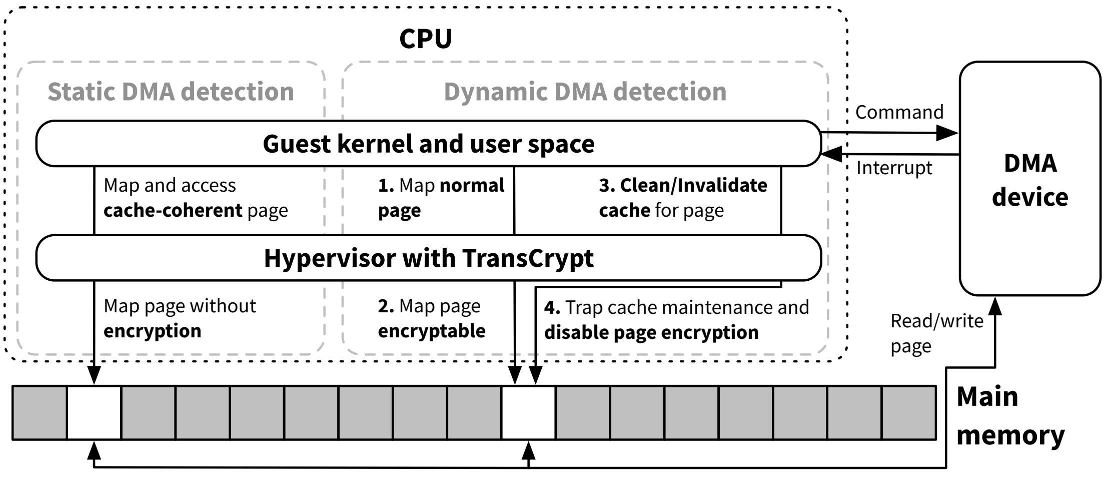

CPU
Static DMA detection
Dynamic DMA detection
Guest kernel and user space
Command
Interrupt
DMA device
Map and access cache-coherent page
1. Map normal page
3. Clean/Invalidate cache for page
Hypervisor with TransCrypt
Map page without encryption
2. Map page encryptable
4. Trap cache maintenance and disable page encryption
Read/write page
Main memory

Figure 6.4: Detecting DMA pages based on caching attributes and maintenance.

configured non-cacheable or, if the ARM platform supports it and the DMA device is configured to take part, use a special *outer shareability* attribute, which allows some DMA devices on ARM platforms to, in fact, access CPU caches. TransCrypt is able to detect when a guest maps a page this way by analyzing the guest-controlled translation during the respective page fault. We call this technique *static DMA detection* and it is illustrated in the left part of Figure 6.4.

CPU to Device. For this type, the CPU must make sure that the data it sends to the device is flushed out of the caches into main memory before triggering the device to read. This can either be done by using a buffer on a page with write-through cacheability or by using explicit architectural cache maintenance operations. TransCrypt detects the first with static DMA detection and the second by trapping and emulating all cache flush operations to the main memory. TransCrypt then determines the target of the cache flush and is able to exclude the page from encryption before it is read by the device. We call this trapping-based technique *dynamic DMA detection* and it is depicted in the right part of Figure 6.4.

Device to CPU. For this type, the CPU must make sure to invalidate the corresponding parts of the caches before reading from a buffer written by a device to not accidentally read old cached data. For cases where the invalidation happens before triggering the device, which seems to be the normal case, it can be detected with the dynamic DMA detection described before. In other cases, one has to analyze the devices and drivers in question and statically exclude the DMA pages.

Our approach has the main advantage of being agnostic and transparent to the guest software and the specific DMA devices in use. All trapped instructions are privileged.

Hence, an unprivileged guest process *cannot* “disable” encryption for certain pages. Cache maintenance operations can be trapped very selectively on ARM. This allows us to react only to operations really affecting the main memory, i.e., the *point of coherency*. Additionally, modern ARM platforms, such as the one used for our prototype, allow configuring automatic cache coherency between cores. With this, normally no cache maintenance operations necessary for inter-core communication. This allows the DMA detection to specifically only target pages that are really used for communication with devices.

6.4 TransCrypt: Implementation

We implemented TransCrypt as part of a minimal, self-developed, standalone hypervisor on an Arndale board, a dual-core ARM Cortex-A15 developer board supporting the ARMv7-A Virtualization Extensions [MN11; ARM14]. With the implementation and evaluation of our tracing framework described in Sections 5.6 and 5.8, we showed that the Cortex-A15 and ARM platforms in general provide a very small overhead for switching into the hypervisor, which is essential for the performance of TransCrypt. To keep the hypervisor as small as possible and reduce error-proneness, our implementation only includes functions necessary for TransCrypt. Therefore, as for our tracing framework, the hypervisor only implements memory management for a single guest and contains no further functionality required for a full virtualization solution, such as support for multiple VMs. Our prototype is fully functional, supports multi-core operation and is able to run an unmodified Linux kernel and Android userland including display output and touchscreen input with enabled encryption. The full implementation consists of about 4000 LOC written in C and ARM assembler.

In the following, we cover some details of our TransCrypt prototype implementation. While some implementation details, especially in the context of the DMA detection mechanisms, are introduced using terminology specific to ARMv7-A, the TransCrypt concepts are also applicable to ARMv8-A. As we will discuss later, such an implementation might even provide certain performance advantages.

6.4.1 Initialization

After setting up exception handlers, especially for SLAT page faults, the hypervisor initializes its own VA to PA translation as a flat mapping over the full RAM of the Arndale board. This makes it easy to access PAs from the hypervisor. The hypervisor furthermore initializes a flat mapping for the guest SLAT from IPAs to PAs but only for the part of memory allocated to the guest. The hypervisor uses 4 KiB pages, the smallest possible page size on ARM. It also enables trapping of cache maintenance operations to the main memory (point of coherency) via the `TPC` bit in the Hyp Configuration Register (HCR) for special page detection.

6.4.2 Basic Mechanism

The core of the TransCrypt prototype is the implementation of the page fault handler as specified in Section 6.3.7. The handler manages an array of page metadata structures, one for each physical page allocated to the guest. Our prototype supports multi-core operation. Each core keeps track of its MPS$_i$ (see Section 6.3.7) and the LRM eviction with a list pointing to elements of the page metadata array. We implemented the fine-granular locking as described in Section 6.3.7 using a spinlock in the metadata of each page. When mapping a page, the core appends it to its MPS list. For eviction, the core just pops the first item of its list. Both operations have constant complexity, making the implementation very efficient.

When mapping a page, it must be ensured that a possible decryption is completely finished and visible before letting the guest access the page. The ARM Cortex-A15 CPU provides a Harvard-style first level cache, i.e., instructions and data are cached separately. Therefore, our hypervisor must ensure that decrypted code is flushed from the first level data cache and the respective instruction cache parts are invalidated to prevent the guest from executing encrypted instructions.

When unmapping a page, it must be ensured that none of the cores still accesses the page because of a stale entry in the TLB. The hypervisor must hence invalidate all entries associated with the unmapped IPA on all cores. Our hypervisor ensures that all cores are in the *inner shareable domain* and uses TLB maintenance operations to broadcast an invalidation executed on one core to all cores. Unfortunately, the ARMv7-A virtualization extensions do not provide invalidation operations based on IPAs. We therefore invalidate *all* guest translations on every page unmapping, which negatively impacts performance. Fortunately, the ARMv8-A architecture [ARM17] provides IPA-based invalidation operations, so that this is no issue for future implementations.

The actual encryption is done with an ARM-optimized software AES implementation. As our Proof of Concept (PoC) focuses on the feasibility of the hypervisor-based concept, it currently does not use encryption from the TrustZone. Running on the same CPU, the only overhead imposed by such an implementation is the TrustZone context switch. As the switch itself is very lightweight [Zha14], especially in relation to the encryption itself, such an implementation should only add negligible overhead to our evaluated prototype. As most of the overhead comes from the actual encryption (see Section 6.5.2), the crypto extensions in ARMv8-A [ARM17] promise to provide significant performance gains for the future. The current prototype uses the page base addresses as IVs for simplicity (see Section 6.3.5).

The implementation provides an additional, optional hashing component which calculates a SHA1 hash of evicted pages to be compared with their content when mapping them again. This allows recognizing changes to pages not originating from the CPU.

The prototype does not implement the dynamic MPS size adaption described in Section 6.3.6 but implements functionality for changing M at run-time for evaluation purposes.

6.4.3 Special Page Detection

Our TransCrypt PoC implementation on the Arndale board uses several techniques to be able to exclude all special pages from encryption as described in Section 6.3.8. First, the hypervisor generates a fixed mapping for all physical address space regions that do not refer to main memory, as discussed in Section 6.3.8.1, making them always available to the guest and excluding them from encryption. On the Arndale board, 2 GiB from `0x40000000` to `0xbfffffff` referring to main memory remain, minus the space allocated to the hypervisor for the main encryption operation.

Second, to determine if a page qualifies as special during a page fault, the PoC queries the guest translation using the architectural translation registers `ATS1CP*` [ARM14]. The hypervisor analyzes the obtained guest's memory attributes and marks a page as special if it is not normal memory with inner and outer write-back cacheability or if it is outer shareable. This realizes both the dynamic approach for device memory discussed in Section 6.3.8.1 and the static DMA detection (see Section 6.3.8.2) in a single step.

Third, the prototype realizes dynamic DMA detection (see also Section 6.3.8.2) by trapping and emulating all cache maintenance operations that use VAs and target the main memory. For all data cache operations in this group, namely `DCIMVAC`, `DCCMVAC` and `DCCIMVAC`, the hypervisor finds the affected page, decrypts it if necessary and marks it as special.

While the described techniques catch most of the special pages, there is one platform-specific exception originating in the Mali GPU driver of the guest kernel. An analysis of this driver revealed that it employs a "physical allocator" which allocates highmem pages, i.e., pages from memory normally used for user space data, and maps them into the kernel space using `kmap()`. These pages end up in the `pkmap` (persistent kernel map) VA region in the kernel VA address space. Hence, we exclude this virtual address range (2 MiB on the Arndale) from encryption. Note that the guest still runs unmodified. On other platforms or kernel and driver versions this might not be necessary.

With the described mechanisms, our memory encryption prototype is able to host a full, unmodified Android OS, without impairing the system's functionality.

6.5 TransCrypt: Evaluation

To evaluate the security and performance of TransCrypt, we ran several experiments and benchmarks on our prototype implementation. For all experiments, we used a multi-core Linux 3.0.31 kernel and Android 4.1.1 on the Arndale board. In most experiments, we tested with different fixed values for the maximum MPS size M, the maximum number of pages mapped and accessible to the guest at the same time. As discussed in Section 6.3.6, M can be used to adapt the security margin as a trade-off versus the performance of the system. The smaller M, the less data is unencrypted in memory and the more recently the data must have been accessed to be still unencrypted.

Table 6.1: TransCrypt page statistics for Android boot with $M = 6000$.

	SPS		**MPS**		**Total**	
	Dyn.	Stat.	$\cap$ SPS	$\setminus$ SPS	UPS	Enc.
All	15072	1890	85	5915	22877	**69352**
Kernel	791	1810	22	1572	4173	**10706**

6.5.1 Security

To get an impression how the memory encryption and the special page detection behave, we collected page statistics from our TransCrypt prototype immediately after the guest finished booting up Android. We chose a fixed M of 6000 for the experiment. Since the Arndale board has two cores, this results in each core managing a maximum of 3000 mapped pages, as described in Section 6.3.7. The results are summarized in Table 6.1. The statistics include the SPS size and the number of special pages detected dynamically and statically (see Section 6.3.8). Furthermore, the statistics show how many of the MPS pages are also part of the SPS and a total of encrypted and unencrypted pages, i.e., the UPS. Additional to the statistics over all pages, we generated data for pages with VAs in the Linux kernel `lowmem` region to determine how actively the kernel space is encrypted. The `lowmem` is the virtual memory region that maps physical memory linearly into the kernel at all times. Typically, kernel code and most of the kernel data structures are accessed through this area. Hence, we can use it to approximate kernel memory page accesses.

There are several interesting results to be read from the statistics. The SPS dynamic detection is primarily active for user space pages, while the static detection is almost exclusively active for kernel pages. The high amount of special pages can be explained by the fact that the system just finished booting, a process that involves a higher than normal kernel activity. The MPS is almost completely filled with non-special pages, which shows that the special page detection is quite accurate, so that most of the current working set is considered for encryption. The summary of encrypted and unencrypted pages shows that about 75% of all pages and 72% of the kernel `lowmem` pages used during the Android boot are currently encrypted. The amount of encrypted memory can be increased further by choosing a smaller M.

To confirm that the kernel space encryption is very active, we attached a JTAG debugger to the system to simulate a memory attack. Without memory encryption, the debugger is able to read and interpret kernel data structures, such as page tables and task lists, for example, to show the processes running on the system. With memory encryption, most of this functionality stops working. While this does not yet necessarily mean that secrets are protected in memory, memory dumps are immediately much harder to analyze forensically without reliable access to kernel data structures.

Table 6.2: Percentage of time the E-mail app's memory contains the plaintext account password for different M in the sample use case (1 min E-mail app $\longrightarrow$ 1 min home screen $\longrightarrow$ 1 min browser) and for the *typical user* calculated based on the average samples.

$M =$	4000	8000	**10000**	14000	18000	36000
Worst	0.66	4.35	**12.49**	33.04	38.74	73.03
Best	0.11	0.28	**0.35**	8.77	30.16	68.70
Avg.	0.34	1.14	**3.37**	16.98	34.07	70.43
Typ.	0.1	0.34	**1.01**	5.09	10.22	23.5

To find out how well the system protects user space secrets, we analyzed the process memory of the Android Mail app. After initializing the app with a test E-Mail account, during normal operation the app's memory showed four occurrences of the address/password combination on three different pages. We marked the physical pages in our hypervisor, tracking their encryption and decryption. While one of the pages remains encrypted all the time, the two others are decrypted when the app is brought to foreground (and checks for new mail). To quantify the time the password is unencrypted in memory, we devised the following sample use case:

1. Open the E-Mail app and wait for one minute.
2. Close the E-Mail app and wait for one minute.
3. Open the Browser app and wait for one minute.

Furthermore, we define a *typical user* who uses the E-Mail app for 3 minutes every 30 minutes (including automatic fetches) for 24 hours of the day. In 85% of the mail app uses, he opens another app such as the browser afterwards. Table 6.2 shows the percentage of time the E-Mail password is unencrypted in the app's memory for our sample test case and the *typical user* for which we calculated the percentage based on the average sample case. The password is counted as unencrypted as soon as *one* of the three pages containing it is unencrypted.

The results for the sample use case can be interpreted as follows. If the percentage is below or around 33% it means that the password is encrypted at the latest when the E-Mail app is closed. This is the case for all M except for $M = 36000$. For this case, the password is encrypted as soon as the browser is opened. Based on these observations, we can calculate the percentage for our *typical user* based on the average case test results. For example, for our typical user and $M = 10000$, the E-Mail password is unencrypted in memory for less than 15 minutes over the whole day. Considering that we get very good benchmark results already for $M \leq 10000$ as shown in the next section, the results confirm that TransCrypt is able to efficiently protect real secrets in memory.

Table 6.3: Performance of TransCrypt memory encryption prototype as percentage of native performance for different benchmarks.

$M =$	4000	8000	**10000**	18000	36000
CoreMark	99.0	99.8	**99.8**	99.9	99.9
Antutu	60.9	77.9	**81.7**	82.6	87.4

6.5.2 Performance

We evaluated the performance of the prototype using two benchmarks. The CoreMark [Con] benchmark, measuring integer performance without GUI, and the Antutu [AnT] Android app, measuring overall performance including CPU, RAM, GPU and I/O speed. The averaged results are summed up in Table 6.3, showing the TransCrypt prototype's performance as percentage of native performance without TransCrypt. Using the same target hardware, the results can be directly compared to the results of the performance evaluation of our hypervisor-based tracing framework described in Section 5.8.

For CoreMark, the TransCrypt performance is almost indistinguishable from the native performance. The reason for this is that the benchmark fits into a small amount of pages. The present impact of less than a percent is probably caused by Android running on the test system and causing pages to be encrypted occasionally. The Antutu benchmark has a much larger working data set including assets for 3D graphics and user interface and should give an impression on how TransCrypt performs running a resource-intensive app such as a game. Despite being a challenging real-world test, the Antutu benchmark already reaches more than 80% of native performance for $M = 10000$.

Most of the performance impact is caused by the encryption itself. Our prototype uses a software AES implementation, so switching to a hardware implementation such as the AES extensions in ARMv8-A, promises a significant performance improvement. As described in Section 6.4, ARMv8-A also offers a much more efficient way for SLAT TLB invalidation, which can further improve performance.

6.5.3 Discussion

In the following, we discuss TransCrypt and the results of our evaluation with respect to the attacker model and attacks defined in Section 6.1.

Our attacker is able to execute a cold boot attack. On a normal system, this leads to a leak of all RAM data including keys, e.g., E-mails, passwords and documents. If the system has a key protection mechanism, such as CPU-bound encryption [MFD11; GM13; Sim11; MTF12], the protected keys are secure but other sensitive memory contents are compromised. With TransCrypt, as shown for the E-mail account password, sensitive data is encrypted with high probability using a key stored in a location not vulnerable to cold boot.

Our attacker is able to execute different reading DMA attacks, as discussed in our attacker model. In all those attacks, the attacking DMA device “acts on its own”, meaning that the guest kernel on the CPU does not initiate the DMA access, at least not for the addresses the device maliciously accesses. Therefore, our DMA detection mechanism described in the Section 6.3.8, does *not* exclude the attacked pages from encryption, providing security for these cases. On a normal system, a DMA memory attack leads to leakage of all memory data. With our memory encryption, a DMA attack reads encrypted memory instead of sensitive data with a high probability as shown for the E-Mail password.

As shown in Section 6.5.1, the probability that an attack reads encrypted memory can be influenced by the choice of the MPS size M. We based our evaluation on experiments with different fixed values of M to analyze its influence on the encryption probability and the performance. As shown, choosing $M = 10000$ already provides a very good trade-off between security and performance. By using a dynamic M adaption, as sketched in Section 6.3.6, this trade-off can be improved further.

While TransCrypt succeeds in reaching its goal to protect the majority of main memory, some heavily used keys, such as symmetric keys used for FDE, could, depending on M, stay unencrypted because of the least-recent access principle. We therefore propose to combine TransCrypt with other protection mechanisms:

- TransCrypt can be combined with a suspend-time memory encryption as described in Section 6.6.3. Additional to the TransCrypt security, such a combination is able to protect all memory including keys of currently inactive process groups or whole systems.

- TransCrypt can be combined with a secure key mechanism, storing heavily used keys in a memory region inaccessible to the discussed memory attacks, for example, in TrustZone OCRAM, as described in Section 6.3.5 for the TransCrypt encryption key itself.

- Keys can be protected by an external device, such as the token in our generic architecture (see Chapter 3). Optimally, the keys are stored exclusively on the token, encrypting and decrypting on behalf of the target platform. Fully externalizing symmetric cryptography is often not an option performance-wise, especially for heavily used symmetric keys, such as the FDE key. For those cases we describe an approach that combines keys from target platform and token, realizing fast external symmetric cryptography in Section 7.2. Furthermore, in Section 7.3, we describe an approach for protecting user IDs with asymmetric keys stored on the token.

Those options are not mutually exclusive and can be combined with each other depending on the hardware features of the target platform used.

6.6 Suspend-time Main Memory Encryption

TransCrypt encrypts data objects in main memory not accessed recently. But because of its heuristic run-time approach, it cannot guarantee the encryption of specific objects, even if it is logically ensured that they are currently not required. We therefore combine TransCrypt with a complementary suspend-time RAM encryption scheme which securely encrypts data of processes currently not needed.

Our suspend-time encryption mechanism builds upon the CGroup freezer functionality in the Linux kernel [Lin]. This subsystem is able to freeze, i.e., *suspend*, groups of processes. While a process is *suspended*, it is ensured that it does not access its data in RAM. This data can therefore safely be encrypted as long as the process is suspended. Our mechanism extends the Linux freezer subsystem to encrypt processes being suspended, i.e., frozen, and decrypt processes being thawed.

Normally, the scope of the encryption is *group-based*. This means that groups of processes, such as containers, are frozen and encrypted together. But, since the Linux kernel also realizes its normal, system-wide suspend-to-RAM functionality with the freezer subsystem, the encryption can also be *system-based* and encrypt all processes during a suspension of the whole system. Hence, in a platform-virtualized environment, a Linux VM is able to encrypt itself when being suspended with our mechanism. Both encryption scopes can be used together to encrypt suspended containers and to encrypt the entire user space of a guest VM when it is suspended.

In the following, we shortly summarize the suspend-time encryption concept including important aspects of the architecture and the encryption process. Afterwards, we discuss details regarding the combination of both our run-time and suspend-time RAM encryption concepts. Further details of our suspend-time RAM encryption approach can be found in the respective publications. The group-based variant in [Hub+17; Hub+18], the system-based variant in [HHW17b].

6.6.1 Architecture and Encryption Concept

Figure 6.5 shows the architecture of a system using our suspend-time memory encryption concept in its group-based form. Resembling our generic architecture (see Chapter 3), there are two hardware components, the target platform and the token. For our suspend-time encryption, the token can be any kind of SE that allows wrapping a key for the host, i.e., is able to encrypt a key with a non-extractable key stored in the token. The user space processes are normally grouped into containers but can also exist on their own, as shown in Figure 6.5 on the left. As mentioned for our generic architecture (see Section 3.1), a user space container on Linux is established using some basic kernel technologies, i.e., primarily namespaces and CGroups. As discussed before, the smallest unit of suspension and, thus, encryption with our suspend-time encryption is a cgroup. For the following discussion, we

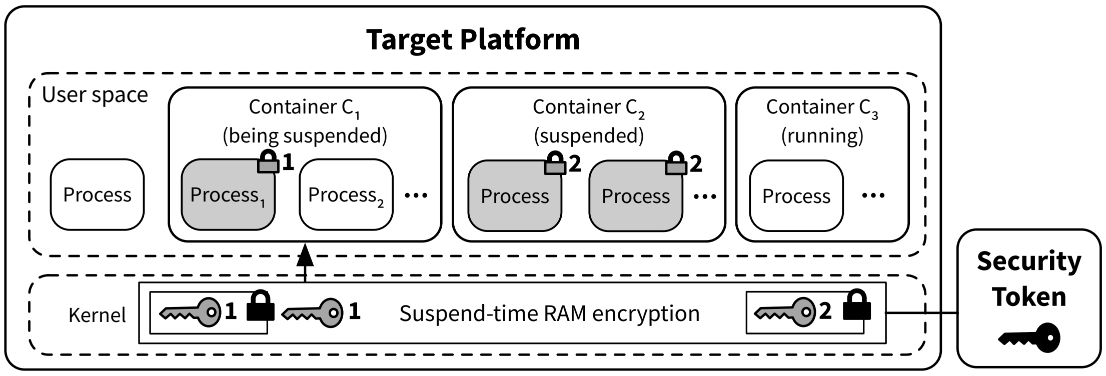


Figure 6.5: Architecture of the suspend-time memory encryption system.

assume that each container equals one cgroup, meaning that, in the presented architecture, the smallest unit of encryption is a container.

Figure 6.5 shows a system with three containers. Container C_2 is currently suspended. The memory contents of its processes are encrypted with Key 2. As it is not required until the container is woken up from its suspended state, the key is only present in the target platform's memory in wrapped form, i.e., encrypted using the token. Container C_3 is currently running. Its processes run completely unchanged from a normal system. They are unencrypted until the container is suspended. As described in Section 6.6.3, the combination with TransCrypt fills that gap. Container C_1 is currently being suspended. Details of the encryption process and the decryption of a suspended container are described in the following.

Encryption. The suspension of a container is triggered through freezing the corresponding cgroup. The freeze command is generated by a user space management component and received in the CGroup freezer kernel subsystem, which signals all processes in the targeted cgroup to enter an infinite loop in the kernel, the *refrigerator* loop. As long as the processes are in the refrigerator loop, it is ensured that they do not access their user space memory contents. Our encryption mechanism extends the CGroup freezer subsystem resulting in the following steps during suspension of a container:

1. The freezer subsystem receives the command to freeze a specific cgroup (container C_1 in Figure 6.5).

2. A management component (typically in user space) generates a new symmetric key for memory encryption and uses the token to generate a wrapped version of the key. The example in Figure 6.5 shows both wrapped and unwrapped Key 1 used for encrypting container C_1. Both keys are stored in the target platform while freezing C_1.

3. The freezer signals all processes in the suspending cgroup to freeze and enter the refrigerator loop in the kernel.
4. Before entering the refrigerator loop, each process encrypts its own memory using the key generated earlier. The encryption is done page-wise using the physical page base address as IV. In the example, $Process_1$ has already finished encrypting itself with Key 1 while $Process_2$ is still unencrypted.
5. As soon as all processes of the cgroup have entered the refrigerator loop, the cgroup is frozen and encrypted and the unwrapped encryption key is purged (see container C_2 in the example).

Having the processes encrypt themselves has two major advantages. First, the memory contents of the process can be easily addressed without having to switch translation tables or generate specific mappings. Second, the encryption of the cgroup is automatically executed in parallel and is therefore very fast. Nonetheless, since processes often share memory in form of select pages or even their entire address space, the approach requires synchronization to avoid malfunctions, such as the double encryption of pages. To this end, our mechanism ensures that each encrypted page is marked and only the last process entering the refrigerator encrypts a shared page, as described in detail in [HHW17b; Hub+17; Hub+18].

Decryption. Waking or *thawing* a cgroup basically inverts the steps executed during freezing. After receiving the command to thaw a container or cgroup, the management component uses the token to unwrap the wrapped key generated and stored during suspension of the container. For this step, the token can optionally be protected with user input, such as a PIN (see also Section 3.2). The extended freezer subsystem signals all processes of the container to thaw, i.e., leave the refrigerator loop. Immediately after leaving the refrigerator loop, the processes use the previously unwrapped key to decrypt themselves. As the encryption, the decryption is automatically executed in parallel and must be synchronized. In contrast to the encryption, the *first* process leaving the refrigerator decrypts a shared encrypted page. As soon as all processes have left the freezer kernel component, the container is fully thawed and both associated keys are purged.

Asymmetric key wrapping. As extension to the presented concept from [HHW17b; Hub+17; Hub+18], we propose an asymmetric key wrapping variant for the suspend-time encryption. For this, the token must contain an asymmetric key pair and provide functionality to the target platform to use the private key without exposing it. The corresponding public key is available to the target platform at all times. When encrypting a container, as before, the target platform generates a new symmetric key but now wraps it using the public key of the token. The wrapped key can only be unwrapped using the token. This approach provides the same security guarantees as the former approach but has the

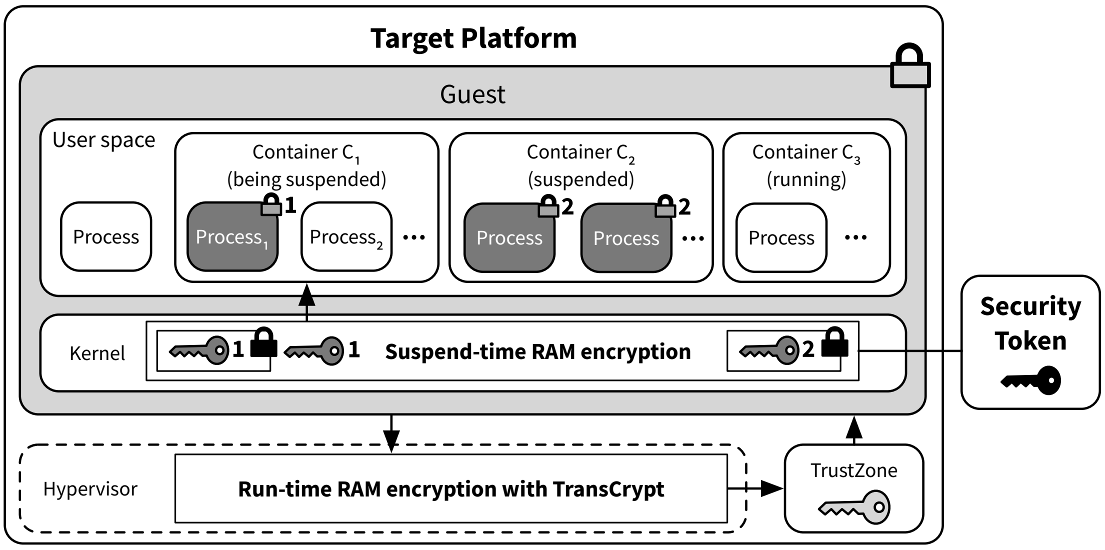


Figure 6.6: Combination of suspend- and run-time memory encryption.

distinct advantage that the suspension and encryption of containers can happen at any time without interaction with the token. Therefore, encryption can even happen without the token being connected to the target platform.

6.6.2 Implementation and Performance

We implemented fully working prototypes of the suspend-time encryption mechanism in both the group-based and the system-based variant. For the system-based variant we chose an x86 platform and for the group-based variant an Android ARM platform. Our system-based prototype on x86 is able to suspend a complete Debian user land running a variety of programs under different loads in about 0.8 seconds on the average, encrypting about 900,000 pages. Our group-based prototype is able to suspend a container running a full-fledged Android in about 2.5 seconds on the average, encrypting about 142,500 pages. Further details regarding the implementation and evaluation can be found in our respective publications for the group-based variant [Hub+17; Hub+18] and the system-based variant [HHW17b].

6.6.3 Combination with TransCrypt

The suspend-time encryption concept is able to protect inactive processes' memory but cannot protect processes of running containers. Hence, in the following, we describe a system that combines the suspend-time encryption with our run-time RAM encryption concept TransCrypt presented earlier in this chapter. Figure 6.6 shows a possible architecture of such a system. Both approaches are orthogonal and can be combined without requiring any changes. This is mainly due to the transparent nature of TransCrypt, which is simply inserted into the hypervisor beneath the guest whose kernel implements the suspend-time encryption. In the resulting system, suspended cgroups are encrypted by the guest kernel itself and TransCrypt transparently encrypts a certain portion of the guest memory following the least-recently mapped principle. For TransCrypt, all guest-memory pages except special pages (see Section 6.3.8) are equally considered for encryption. TransCrypt, therefore, bridges the gap of the suspend-time encryption encrypting running containers and the guest kernel but without giving any guarantees about which parts are encrypted. This also means that in this simple combination of the approaches, a majority of the pages of suspended containers will be double-encrypted, by the suspend-time encryption and TransCrypt. This could be avoided by implementing a communication interface between both mechanisms to allow the guest to notify TransCrypt about pages of suspended containers so that TransCrypt can exclude them from its encryption until told otherwise.

The key for TransCrypt is used regularly during run-time and is therefore stored in a reasonably secure location on the target platform, such as the TrustZone, as discussed in Section 6.3.5. Keys for the suspend-time encryption are only required during suspend and resume and can therefore be protected by the token as discussed before.

The performance impact of both approaches on the combined system is orthogonal as well. At normal run-time, the performance is only impacted by TransCrypt as discussed in Section 6.5.2. As soon as a container is suspended or woken up, the performance is impacted by our suspend-time encryption as described in Section 6.6.2.

6.6.4 Security Discussion

In the following, we discuss the security of the system combining run-time and suspend-time RAM encryption. Our TransCrypt attacker model (see Section 6.1) considers a *basic* attacker who is able to execute a memory attack extracting the complete memory of the target platform without going through the main CPU of the system. For the following discussion, we extend the attacker model as follows:

- The attacker is in possession of the target platform but cannot gain access to the keys stored in the token either because it is removed from the target platform or because it is locked.
- Additionally to the memory attack, an *advanced* attacker is able to launch an attack on the target platform and extract all its secrets including those stored in higher privilege layers, such as TEEs (TrustZone).

All pages encrypted by TransCrypt *or* by the suspend-time encryption are secure against the basic attacker. With its ability to extract a full memory image from the target platform, the basic attacker cannot gain access to the TransCrypt key, since it is not stored in normal RAM. Keys of suspend-encrypted containers are only stored in memory in wrapped form and cannot be unwrapped without the token. For containers that are in the process of being encrypted when the attack happens, the suspend-encryption can be circumvented since the unwrapped key is temporarily in main memory. Nonetheless, parts of the container that are currently encrypted by TransCrypt are still secure.

With its complete control over the target platform, the advanced attacker is able to gain access to the TransCrypt key. Therefore, pages that are only encrypted by TransCrypt can be decrypted by the attacker. Nonetheless, suspend-encrypted processes and containers *remain* secure against the advanced attacker since the keys are only available in combination with the unlocked token. The TransCrypt key is stored on the target since it is constantly in use.

In a real world example use case, such as the one we described in [Hub+15] where the target platform is a smartphone running multiple Android instances, the protection gained by the combined RAM encryption almost always protects major parts of memory against both attackers. In such a scenario, only a single container can be active at any time. This means that all other containers are in any case protected against both attackers. Furthermore, as long as the smartphone is not actively used, *all* containers are suspend-encrypted. Parts of the running container are furthermore protected against the basic attacker by TransCrypt. Note that the suspend-encryption also encrypts keys used by the suspended container, including especially the FDE key. Hence, the protection extends to the secondary storage.

Summarizing, the combined system provides the advantages of both approaches implementing two security layers protecting the confidentiality of main memory. Suspended containers are encrypted with a key protected by the security token and, hence, secure against the advanced attacker. Running containers are secured against normal memory attacks using the best-effort encryption of TransCrypt.

6.7 SEVered: An Attack on Hardware-based VM Encryption

Both our presented memory encryption schemes are software-based. As a recent development, hardware manufacturers try to incorporate protections directly into their CPUs. As discussed before in Section 6.2, Intel plans to include TME, a technology for encryption of the entire main memory, in future CPUs. With SME and SEV (see also Section 2.2.3), AMD already supports memory encryption in their most recent high-end CPUs.

SME encrypts the entire RAM with a single key in order to prevent physical memory attacks. The attacker model of SME is therefore similar to the one we used for TransCrypt. SEV, on the other hand, builds upon AMD's virtualization extensions and encrypts VMs, each with a different key. The encryption itself is handled by a dedicated co-processor, the AMD-SP, and the keys are never exposed to the CPU. In comparison to SME, the memory of an encrypted VM should not only be protected against physical attacks but also against a possibly malicious hypervisor managing the VM. The goal of SEV is to relieve VM owners from having to fully trust their server providers. Hence, SEV implements a trust model that

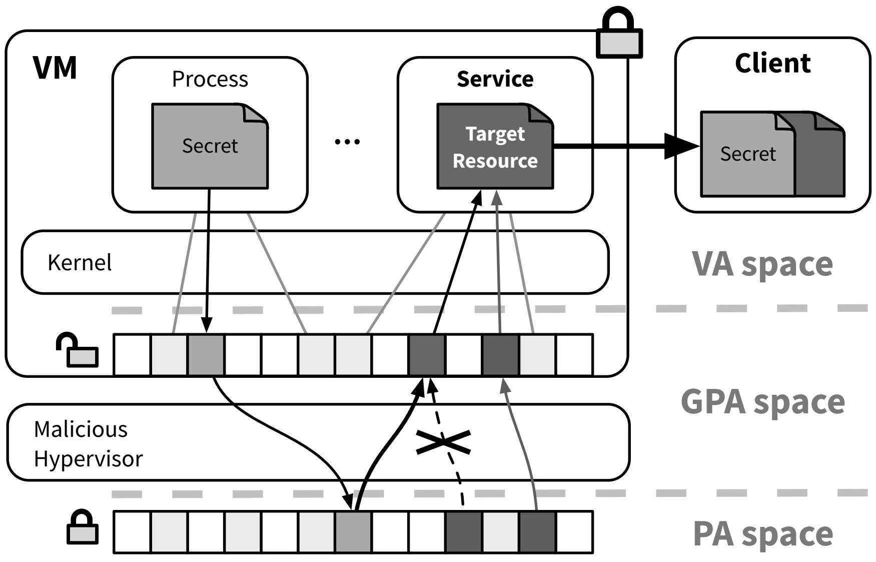


Figure 6.7: Architecture and mechanism of the SEVered attack.

is very different from our generic architecture, in which we try to move security-critical functionality in more privileged and, thus, *more trusted* layers.

In the following we introduce SEVered, an attack on SEV allowing a malicious hypervisor to extract the entire memory of an encrypted VM. The attack exploits the fact that SEV does not provide integrity or replay protection for encrypted pages of a VM and tricks a service in the VM, such as a web server, into sending arbitrary VM pages to the attacker. Even if the basic problem that enables our attack is solvable with a hardware revision, the attack shows that the associated trust model is hard to realize securely.

In the following, we first discuss conceptional details of the attack. Then we summarize the evaluation of our PoC on an SEV-enabled system powered by an AMD Epyc CPU before shortly discussing impact and possible countermeasures.

Further details of the SEVered attack have been published in [Mor+18].

6.7.1 Attack Concept

Figure 6.7 shows the basic concept and the architectural components of the SEVered attack. SEVered is executed from a malicious hypervisor and targets an SEV-encrypted VM. As it can be seen from the figure, the VM controls its own translations from VAs to GPAs[1] and the relationship between those address types is completely hidden from the hypervisor. The hypervisor, as in normal virtualization solutions, controls the translation from GPAs to

[1]Instead of the ARM architecture-specific term Intermediate Physical Address (IPA), we use the more general term Guest-Physical Address (GPA), which has the same meaning, in this AMD-centric section.

PA. The attack requires the presence of two specific **components** in the targeted encrypted VM:

Service. A process running in the targeted VM that provides a remotely accessible public interface via which it offers a resource to clients. The service could, for example, be a web server or an SSH server.

Resource. One or more pages in the memory of the VM offered to clients via the target service. An example for a typical resource of a web server is a web page.

The suitability of a resource for use in SEVered depends on its size and stickiness. We define a resource to be *sticky* if it is probable that it remains in the VM's memory without changing position or being evicted during the attack. A resource should be at least page-sized and optimally be aligned to a page. In [Mor+18], we describe how to find such resources.

The basic idea of the attack is to identify pages belonging to the chosen target resource and then switch those pages with other (secret) pages of the VM by modifying the SLAT mapping in the hypervisor. In a following request to the resource by a client, the service transparently and unintentionally reads the switched page's data from VM memory and sends it to the client. Since the request is executed by a process inside the VM, the SEV encryption engine allows the decryption of the data. Based on this, we structure the attack in two steps:

1. **Resource identification.** In this attack step, the hypervisor identifies which pages of the VM, i.e., GPAs, belong to the target resource by tracking page accesses while interacting as client with the target service.
2. **Data extraction.** In this attack step, the identified resource pages (GPAs) are re-mapped to other pages of the VM (PAs) and the resource is requested from the service. Repeating this allows the attacker to successively extract all memory regions of interest from the targeted VM.

In the following, we discuss the conceptual details of both phases.

6.7.1.1 Resource Identification

In order to be able to switch the pages of the target resource with other pages of the VM's memory, the resource pages must be identified first. The goal of this phase of the attack is therefore to identify the following set of pages:

$$R = \{p : \text{Page } p \text{ contains (part of) the target resource}\}$$

Since the VM is SEV-encrypted, normal introspection is not an option. The VM's translations are completely hidden from the hypervisor but page faults in the SLAT can still be observed. Based on this, we establish a basic *page access tracking technique* similar to the one we use for our hypervisor tracing framework (see Section 5.3.4): We invalidate all PTEs of the VM and record page accesses based on SLAT page faults afterwards. After an initial invalidation, the tracking triggers only *once* per page accessed and results in a sequence of pages, we call a *recording* in the following.

By interacting with the target service in the VM while tracking page accesses, we infer knowledge about the VM's memory layout. We propose an iterative approach to be able to dynamically cope with different levels of *noise*, i.e., concurrent activity in our target VM during recordings. For this, we repeat different recordings and combine the results after each iteration. Increasing the number of iterations should remove higher levels of noise and lead to a more accurate approximation of R.

We conduct $n \in \mathbb{N}$ iterations, with i denoting current iteration $1 \leq i \leq n$, repeating the following steps. First, we request the target resource from our target service and record the pages the VM accesses until we receive the reply with the resource:

$$R_i = \{p : \text{Page } p \text{ accessed during request } i \text{ for target resource}\}$$

Since we request the resource ourselves, it is guaranteed that the VM accesses it and that it is therefore part of R_i:

$$R \subseteq R_i$$

Additionally to R, the set R_i contains all other pages required to fulfill our request and also pages accessed due to concurrent activity. As it directly obscures our result and makes it less accurate, we consider *all* concurrent activity during the recording of R_i as noise and call it *R-noise* in the following. Each R_i contains an unknown amount of R-noise but is also guaranteed to contain R. Based on this observation, we define a set R^i as the intersection of all recorded R_i, removing variable components of R-noise:

$$R^i = R^{i-1} \cap R_i \qquad R^0 = R_1$$

After an appropriate number of iterations depending on the amount of R-noise, the resulting set R^i only contains pages that are immediately required for fulfilling our request to the target resource. Since the resource is part of all recordings ($R \subseteq R_i$), it must also be part of their intersection ($R \subseteq R^i$).

R_i still contains *all* pages of the VM required to deliver the resource, including data and code pages of the service and the kernel and is therefore typically still quite large ($|R| \ll |R_i|$). In order to filter all pages from the set that are not the actual target resource, we record page accesses while requesting a similar but different resource from our target resource:

$$X_i = \{p : \text{Page } p \text{ accessed during request } i \text{ for similar resource}\}$$

As another client might access the target resource while we record X_i, we cannot be sure that R is not part of X_i. But since we do not access it ourselves, we assume that X_i is *unlikely* to contain R. We then combine X_i with R^i, which is guaranteed to contain R, to obtain a set of *likely candidates* C_i for each iteration:

$$C_i = R^i \setminus X_i$$

Since we cannot be sure that $R \notin X_i$, we also cannot be sure that $R \in C_i$. If R is accessed while we record X_i, the candidate set is incorrect for this iteration. Therefore, concurrent accesses to our target resource must be considered as noise during recording X_i. We call this type of noise *X-noise*. All concurrent accesses that do not access R do not adversely affect our attack. To reduce X-noise in all iterations, a rarely accessed resource should be chosen as target R, if possible.

In the next step, we want to gather all candidates of all iterations without losing information about *how often* each page was a candidate. For this, we define the multiset C^i, which is updated in each iteration. A multiset is a set that may contain more than one instance of an element. We denote the multiplicity of an element p in the multiset A as $A(p)$. Furthermore, we specify the following operations on multisets: The union of a multiset with a set or a multiset we specify as the sum of multiplicities, i.e., $(A \uplus B)(p) = A(p) + B(p)$, and the intersection as multiplication of multiplicities, i.e., $(A \cap B)(p) = A(p) \cdot B(p)$. With these definitions in place, we define C^i as:

$$C^i = (C^{i-1} \uplus C_i) \cap R^i \qquad C^0 = \emptyset$$

The first term of the equation $(C^{i-1} \uplus C_i)$ gathers all candidates from all iterations ensuring that candidates are represented in the set according to the number of their occurrences in the samples. For example, if a page is present in the candidate set of three iterations, its multiplicity in C^i is three. As explained before, each iteration refines and possibly downsizes the set R^i while guaranteeing that $R \subseteq R^i$. From $R \subseteq R^i$ follows that pages $p \notin R^i$ cannot be part of R. For the current iteration, the candidates C_i are cleaned, i.e., intersected, with the most recent version of R^i. But C^i might still contain pages from previous iterations that can be safely excluded from the candidates with the knowledge and the updated set R^i acquired in the current iteration. Therefore, the intersection, as second part of the equation, ensures that those pages are completely removed from C^i. After iteration i, based on C^i, we can calculate for each candidate page how probable it is part of R:

$$\mathrm{P}_i[p \in R] = \frac{C^i(p)}{|C^i|}$$

If the resource spans over multiple pages, i.e., $|R| > 1$, the probability is distributed over all pages $p \in R$. The calculation mainly serves to assess the pages' probability in relation to each other and not as an absolute value.

After each iteration, we can generate an ordered list of the pages that most probably contain R. By increasing the number of total iterations n executed before proceeding into the next phase of the attack, the model is able to filter higher levels of R- and X-noise, as we discuss in our evaluation in Section 6.7.2. With the generated list of probable resource pages, we start the actual data extraction from the encrypted VM in the next phase.

[Mor+18] contains a detailed example with actual values, further clarifying our resource identification algorithm.

6.7.1.2 Data Extraction

Based on the knowledge about the location of the resource in the VM's memory gained in the previous phase, in this attack phase we exploit the control of the SLAT in the hypervisor to extract arbitrary data from the VM. While the exact number of pages containing parts of the resource, i.e., $|R|$, is unknown, resource size S_r and page size S_p are known and can be used to calculate an approximation $r = S_r/S_p \leq |R|$. The data extraction phase is structured into two steps:

1. **Resource Re-mapping.** In this step, we take the first r pages that, according to the list generated in the previous step, most probably contain parts of the target resource and switch their SLAT mapping to r VM memory pages of interest, as illustrated in Figure 6.7.

2. **Resource Request.** In this step, we request the resource as a client from the target service in the VM. Since the pages in which the resource is stored are re-mapped, the service unintentionally sends the pages of interest instead of the actual resource.

These two steps are repeated until all memory regions of interest are extracted from the VM. Since the pages are accessed by the service from within the VM itself, the AMD-SP deliberately decrypts them using the corresponding VM key. As VMs use different keys, the attack is limited to memory belonging to the targeted VM. If the resource received in the second step contains (parts of) the actual resource data, the re-mapping is reversed and carried out for the next pages in the candidates list. Depending on the position in the list and the stickiness of the resource, the identification phase might have to be repeated.

Concurrent accesses to the re-mapped resource, e.g., by another client, also access the *currently* mapped pages. Depending on its nature, concurrent activity might therefore lead to unexpected behavior. Hence, SEVered should be used with a page-aligned, read-only, rarely used resource to exclude malfunctions during the data extraction.

6.7.2 Evaluation

We implemented a fully functional PoC on one of the first available SEV-enabled AMD Epyc processors. The implementation is based on Linux Kernel-based Virtual Machine (KVM) and QEMU. As target services we evaluated the Apache and nginx web servers and

Table 6.4: Number of iterations and time required until top set converges ($|T| \leq 5$) for different noise levels.

Noise Level	Apache	Nginx	OpenSSH
20	10 / 7.4 s	8 / 5.56 s	21 / 38.85 s
30	10 / 7.5 s	9 / 6.62 s	42 / 85.47 s
40	12 / 9.7 s	13 / 13.2 s	46 / 111.09 s
50	22 / 23.0 s	16 / 17.84 s	>100 / >5 min

OpenSSH, all serving a 4 KiB sized file filling exactly one page as target resource. In the following we shortly summarize the results from evaluating the attack PoC structured in its two phases. Further details of the implementation and the full evaluation can be found in [Mor+18].

Resource Identification. For the evaluation of the resource identification phase, the target resource page t is known. This allows us to define a *top set* as basis for measuring the performance of our identification mechanism:

$$T_i = \{p \in R^i : \mathrm{P}_i[p \in R] \geq \mathrm{P}_i[t \in R]\}$$

After i iterations, the top set T_i contains all pages that our algorithm considers *at least as likely* to contain our target resource as the actual target page t. The fewer pages T_i contains, i.e., the smaller $|T|$, the better the identification. Optimally, $|T|$ reaches 1, i.e., only contains t, in which case the page-sized target resource's location is precisely determined.

Furthermore, we use a noise model for our evaluation in which we generate randomly distributed requests to our three target services with a certain fixed frequency, the *noise level*. For our tests, we varied the noise level from 20 to 50 requests per second. With its randomly distributed requests, including concurrent requests to the target resource, the model generates both X- and R-noise.

Table 6.4 summarizes the results of the evaluation of the resource identification phase. It shows the iterations and the time required to reduce the top set size to five pages or fewer, providing a good basis for proceeding to the next phase. It furthermore differentiates between several noise levels.

Both web servers converge reasonably quickly for all noise levels. The OpenSSH server has a higher latency answering requests. Since the duration for recording X_i and R_i equals the time required for answering a request, this causes larger samples and therefore impedes identification. Hence, for OpenSSH, the identification requires significantly more iterations but still converges for most noise levels. The results confirm that our identification mechanism works as intended and is able to quickly identify a target resource even in VMs with high amounts of both X- and R-noise.

Table 6.5: Extraction speed for different target services using a page-sized resource.

Apache	Nginx	OpenSSH
79.4 KB/sec	79.4 KB/sec	41.6 KB/sec

Data Extraction. With knowledge about the target resource's location, we were able to reliably extract *all* memory of the target VM. Table 6.5 shows the extraction speed for our three test services. As expected, enabling noise based on the noise model introduced before did not adversely affect performance or functionality of the extraction. Both web servers show the same extraction speed while OpenSSH performs worse, again because of its higher response latency. The measured speeds are fast enough to extract large amounts of memory in reasonable time. Nonetheless, it should be possible to significantly improve the extraction speed by using resources allocating multiple pages, since such a resource allows the extraction of more data per request.

6.7.3 Discussion and Countermeasures

Our PoC and evaluation clearly show that SEVered is able to successfully attack AMD SEV-encrypted VMs. Even in noisy VMs, i.e., VMs with a lot of concurrent activity, the attack succeeds reliably and within reasonable time. In a follow-up publication [MHH19] not further discussed in this thesis, we were able to additionally improve our attack by observing the behavior of the VM and analyzing page faults to specifically identify pages that contain secret keys in the VM's memory and, then, prioritize them for extraction.

In the following, we discuss some basic ideas for countermeasures against SEVered. As a software-based countermeasure, one might try to implement mechanisms to prevent a VM from leaking information through page faults. For example, the VM could try to access pages in deterministic patterns, independent of user input. Such an approach might be able inhibit the SEVered target resource identification, but obviously is very hard to implement and comes with a significant performance overhead to the whole VM. Another software-based countermeasure could be to prohibit actual data extraction by protecting page integrity *from within* the VM. This could prevent the hypervisor from switching mappings without the VM noticing. Apart from the massive performance implications, such an approach would require a secure location for storing metadata, e.g., hashes of pages. Furthermore, the approach has the major problem that the VM cannot know *when* a switch might happen and an integrity check is therefore required.

Considering the outlined problems, a software-based defense against SEVered seems hard or even impossible to realize. Therefore, an adaption of the SEV hardware (or firmware) seems inevitable. Such a hardware-based countermeasure could be to include the guest-assigned GPA of a VM's page into the SEV protection. As a base solution, the GPA could be used as some kind of input (e.g., as IV) for the encryption. Then the decryption of a switched page in the AMD-SP would fail implicitly or even better explicitly depending on the actual integration of the GPA. As a more costly solution, the GPA could be combined with an explicit integrity protection, i.e., a hash digest of the page's contents. Furthermore, a nonce could be integrated to prevent replay attacks of older versions of pages. All stages of the solution have the advantage that a benign hypervisor would still be able to move pages around in physical memory and change the SLAT mappings, for example, to dynamically re-organize memory running multiple VMs as long as the newly allocated physical page has the exact same contents.

6.8 Summary

Our goal for this chapter was to explore ways to protect the confidentiality of our target platform's main memory against memory attacks. Those attacks are typically executed by a physical attacker using different attack vectors, such as cold boot, to extract a full memory dump from the target platform. In order to fully protect our target platform, we combined two complementary software-based mechanisms, one for run-time and one for suspend-time memory encryption.

First, we presented TransCrypt, a concept and implementation for guest-transparent, run-time encryption of kernel and user memory from a custom, minimal hypervisor. TransCrypt utilizes the SLAT control in the hypervisor to restrict the guest access to a small working set of physical pages. By keeping the other pages encrypted, only recently accessed memory pages remain unencrypted. Since run-time encryption of certain memory regions used for peripheral device control and communication, e.g., for DMA, can lead to malfunctions, we introduced a concept for detecting those special pages to exclude them from the encryption in a transparent and guest-agnostic way on ARM architectures. We developed a fully functional prototype of TransCrypt on the Arndale ARM Cortex-A15 development board. Our evaluation shows that the system can effectively protect secrets in memory while keeping the performance impact relatively small. For example, TransCrypt is able to keep the E-mail account password of a typical user in the Android mail app's memory encrypted 98.99% of the time, while still reaching 81.7% and 99.8% of native performance in our two benchmarks.

TransCrypt is able to protect kernel and user space memory at run-time but always leaves a small portion of recently accessed memory unencrypted. Because of the continuous run-time encryption, TransCrypt furthermore requires constant access to its encryption key whose protection is therefore constrained by performance considerations. Typically, not all functional parts of a system are required at all times. Based on those observations, we proposed the combination with a suspend-time encryption scheme, able to protect the memory of a suspended system or suspended groups of processes. The mechanism is built into the Linux kernel and utilizes its functionality to force suspending and resuming processes to encrypt and decrypt their own memory. The keys used for encryption are only available on the target platform during the actual encryption and decryption operations and encrypted using the token otherwise. With this, the mechanism is able to protect suspended processes on the target platform even against an attacker with complete software control.

Finally, we presented an attack on AMD SEV, a hardware mechanism in recent AMD CPUs able to encrypt main memory of VMs. In contrast to TransCrypt, SEV's goal is to remove the hypervisor from the TCB required for ensuring memory confidentiality of a VM. With our attack we show that a trust model that removes trust from privileged layers is hard to realize securely.

Protecting Keys and Identities 7

In the previous chapters we introduced different mechanisms protecting the integrity and confidentiality of our target platform. Against a remote attacker, we introduced techniques to verify and protect code integrity and execution at boot- and run-time. Against a physical attacker, we proposed schemes for memory encryption of running and suspended processes. The leveraging of logical separations provided by the target platform's hardware is a fundamental characteristic and design goal of our architecture. Hence, most of the introduced protection techniques heavily rely on the available separations inside the target platform. For example, TransCrypt uses a TEE for its memory encryption.

While some of the CPU's hardware resources, such as modes of execution and specific registers, are effectively separated, other resources, such as caches, are still shared between compartments in modern, high performance CPUs. As recent research results show, this resource sharing can be exploited by advanced attackers to extract sensitive data from other, architecturally separated compartments inside the same CPU. For example, cryptographic implementations in separated compartments can be targeted by cache timing side channel attacks [Ber05; YF14] to extract keys. But even more advanced techniques are possible. *Spectre* [Koc+18; Hor18] and *Meltdown* [Lip+18b; Hor18] are attacks that exploit speculative execution in conjunction with side channels, typically cache timing, on modern CPUs to read data from other, separated contexts. *Meltdown* triggers speculative execution of instructions accessing privileged data from the context of a user process. On some CPUs, primarily Intel processors, the permissions of accessed memory pages are checked *after* the speculative access is performed. Additionally, on some ARM processors, permissions for accessing privileged processor registers are checked *after* the speculative execution of the corresponding instructions [ARM18]. While the access instruction is reverted later and has no architectural effect, its effect on the cache can still be observed by the attacker leaking information and ultimately contents of privileged memory. *Spectre* uses the same principle but additionally leverages the fact that also the branch prediction resources are shared between privilege levels. Based on this observation, *Spectre* manipulates the branch prediction from a less privileged context to trigger speculative memory accesses in code running in a more privileged context. Afterwards, the attacker can analyze side effects on the cache [Koc+18; Hor18] or other microarchitectural components, such as Single Instruction Multiple Data (SIMD) units [Sch+18], to extract privileged memory contents. Since the privileged context is able to access the targeted memory without violating access permissions, *Spectre* is not limited to Intel CPUs that defer permission checks after speculative accesses, but leaves almost all popular high performance CPU, including ARM [ARM18], Intel and AMD vulnerable. The discussed side channel techniques illustrate the fact that isolation between privilege levels and compartments inside a CPU is not as secure as physical separation. A remote attacker who manages to achieve unprivileged code execution on our target platform might, depending on the hardware, be able to extract sensitive data, such as keys, possibly even from a TEE [Che+18]. *NetSpectre* [Sch+18] is even able to extract memory remotely without code execution, only by measuring network packet response times.

Spectre, *Meltdown* and other attacks that exploit microarchitectural effects are typically summarized as *microarchitectural attacks*. Another recent class of microarchitectural attacks are *Rowhammer* attacks. Like *Spectre* and *Meltdown*, *Rowhammer* attacks exploit hardware characteristics and microarchitectural effects, but of DRAM instead of CPUs. Architecturally, an access to a specific address in DRAM should not influence other contents of memory to allow for sharing of DRAM between isolated contexts managed by system software through translation tables. Unfortunately, as it turns out [Kim+14], accessing a specific DRAM address frequently in rapid succession can trigger bit flips in nearby DRAM locations. As hardware effect, the flips can be triggered across logically separated CPU contexts. One of the first *Rowhammer* attacks [Sea15] exploits the effect on an x86 Linux platform to flip bits in page tables, escalating the privileges of a user space process. *Drammer* [Vee+16] shows that *Rowhammer* is also possible on ARM platforms. In contrast to x86, the ARM architecture does not expose cache maintenance operations to user space, requiring Drammer to use an uncached DMA memory region mapped into user space for the attack. Another recent publication [Fri+18] shows that *Rowhammer* can be triggered by the GPU in a smartphone SoC. Based on this observation, the authors construct a *remote* exploit triggering a *Rowhammer* attack with WebGL and achieving RCE. Two other recent publications [Tat+18; Lip+18a] trigger *Rowhammer* exploits remotely via NICs, not requiring any prior code execution on the target. Last but not least, *Rowhammer* attacks might not only be used to gain kernel privileges, but also to break isolation of stronger logical separations, such as the TrustZone TEE [Car17].

Besides microarchitectural attacks, also software bugs can be exploited to gain control over or extract secrets from higher privilege levels as already discussed in Section 5.1. This is not only true for large and complex software layers, such as the kernel on our target platform, but also for typical TrustZone implementations [Ros14; Ben17].

Naturally, the attacks discussed affect all kinds of defense strategies that rely on privilege separation for key protection and, for example, store keys in special processor registers

[MFD11; GM13; Sim11], in higher privilege levels [MTF12; HWF15] and/or special memory regions [Col+15].

Apart from microarchitectural attacks and software bug exploits, the security of keys in RAM is still also threatened by memory attacks (see Section 6.1), if not specifically protected. Local, physical attacks, e.g., via Cold Boot [Hal+09; MS13] and DMA [Boi06; SB12; Mar+19], are able to extract keys from normal RAM disregarding privilege levels and compartments in the targeted CPU. While the combined memory encryption scheme we introduced in Section 6.6.3 protects most of the RAM on our target platform, as discussed before, frequently accessed memory regions of running processes might remain unencrypted. This also affects frequently used keys, such as the FDE key.

The attacks described illustrate the usefulness of physical separation, at least for small, high-value data objects, such as keys. In this chapter, we want to explore ways to protect keys in our system assuming a partly or fully compromised target platform. To achieve better key protection, we want to leverage the physical separation offered by the token in our architecture. More specifically, we want to externalize major aspects of both asymmetric and symmetric cryptography and corresponding keys to the token in a usable way. The token, as dedicated security device, can be hardened much more effectively and protects itself by checking the integrity of the target platform before offering its security functionality as described in Chapter 4. The token and its security can be further bound to the user by being removable and equipped with a locking mechanism, as detailed in Section 3.2.

Protecting symmetric and asymmetric cryptography poses different challenges. Symmetric cryptography typically happens very frequently and is therefore often very performance-critical. Hence, the operations are normally executed directly on the host making the keys used susceptible to attacks. A common solution to this problem is to move the cryptographic operations completely to an external, specially secured crypto device, such as the token in our architecture. Either because of the slow cryptography on the device or its slow connection to the host, such approaches are usually not suitable for data-intensive use cases such as FDE on behalf of the target platform. Other approaches involving external cryptographic devices either share keys with the host at least for a short time [JH10; CN05; Kao+12; Sea06; Hit08], are inflexible [Hig], suffer from slow performance [Yuba] or do not protect data encrypted for different applications from each other [Hig; Sea06; Hit08]. To overcome these problems when externalizing symmetric cryptography, in this chapter, we propose CoKey, a concept which partially moves symmetric cryptography out of our target platform into the token in a way fast enough for real-world, data-intensive use cases and multiplying the speed of the underlying physical connection. For CoKey, the target platform takes the role of the *host* on whose behalf the token executes cryptographic operations.

Asymmetric cryptography typically happens less frequently. It is primarily used to authenticate communication and for deriving symmetric keys for bulk data encryption. Therefore, its operations and keys, representing a user's *IDs*, can be fully externalized to our token without raising performance problems. But externalizing IDs to a token connected to a possibly malicious target platform poses its own set of challenges. Especially when considering highly portable and versatile target platforms, such as smartphones, the stored IDs might be at risk and their management becomes more important. Therefore, in the second part of this chapter, we extend our architecture with a strong, long-term issued generic root identity device, called *RootID*, and a trusted third party, the Trusted Identity Provider (TIP), and introduce TrustID, a concept allowing a user to securely derive IDs from his RootID into the token even when connected to a malicious target platform. The derived IDs can be used via all interfaces the target platform provides without requiring the physical presence of the RootID, which can therefore be stored in a secure location.

The chapter is organized as follows. In Section 7.1, we specify our extended attacker model for this chapter, assuming a partly or fully compromised target platform. In Section 7.2, we introduce and discuss CoKey, our concept and implementation for securing symmetric cryptography with our token. In Section 7.3, we then introduce TrustID, our concept for improving the security of identities using our token.

Parts of this chapter have been published in [HWE16; Hor+14a].

7.1 Attacker Model

As defined in our generic attacker model in Section 3.3, we differentiate between two types of attackers, namely a remote and a physical attacker. For both, we assume that they are not able to attack the token and especially not able to extract any key material from it. This assumption is reasonable considering that the token, as a special purpose device, can be protected well using techniques such as a secure boot, moving keys into a TEE, such as the TrustZone, and prohibiting leaking of special keys by hardware means. Furthermore, the token only offers a very minimalist interface (see Section 7.2.3), lowering the risk of software exploits through the target platform, i.e., the host. The RootID and TIP for TrustID are assumed to be trusted as well.

In the beginning of this chapter, we discussed attacks breaking logical separations on the target platform. Remote attackers gaining unprivileged code execution might be able to leak keys from privileged contexts using cache timing side channels [Ber05; YF14] and *Spectre* [Koc+18; Hor18] or *Meltdown* [Lip+18b; Hor18] might even allow an attacker to read arbitrary data from other, more privileged contexts. Furthermore, *Rowhammer* attacks might be used for privilege escalation on x86 [Sea15] and ARM [Vee+16], for remote RCE via GPUs [Fri+18] or NICs [Tat+18; Lip+18a], and for attacks on the ARM TrustZone [Car17]. Additionally, an attacker might be able to escalate his privileges using software bug exploits [Ros14; Ben17]. A physical attacker, on the other hand, might be able to use memory attacks to access RAM and break through logical separations on our target platform, e.g., via Cold Boot [Hal+09; Gut01; MS13] or DMA [Boi06; BDK05; SB12; Mar+19]. In order to be able to handle also future attacks on logical separation in the target platform, we therefore conservatively assume that both attackers are able to

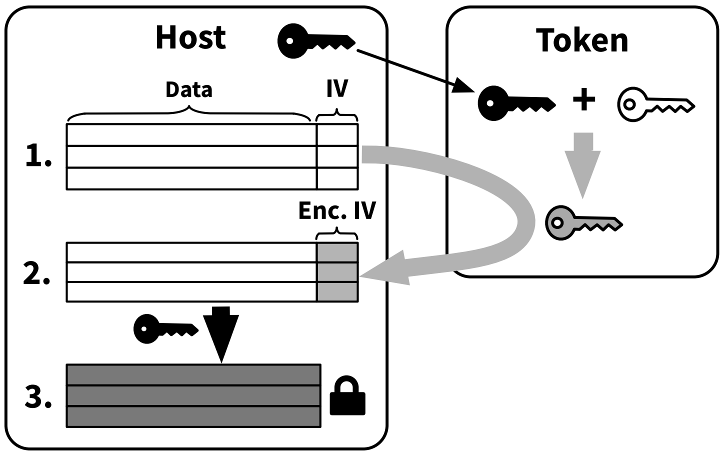


Figure 7.1: CoKey conceptual overview.

completely compromise the host. Both attackers are able to modify software, run arbitrary programs and read arbitrary memory on the target platform. Since the token typically only communicates via the target platform, this implicitly means that the attacker can intercept and manipulate all communication of the token with other components in the architecture.

Besides the main ability to compromise the target platform, there are some secondary properties that are specific to the attacker types. The *remote attacker* has the advantage of being stealthy, i.e., having access to the target platform while the legitimate user is using it. But he is not able to influence *when* the token is connected to the target platform and when it is unlocked. The *physical attacker* might have physical possession of the target platform, the token or both and can therefore possibly decide about when the token is connected. But he is not stealthy and cannot unlock the token without the legitimate user.

The main goal of the attacker is to access the security functionality offered by the token on the target platform *without* the token being connected and unlocked.

7.2 CoKey: Fast Token-based Symmetric Cryptography

With CoKey, we present a concept to improve the security of symmetric cryptography on our target platform by leveraging the physical separation provided by the token in our architecture. The basic CoKey concept is illustrated in Figure 7.1. The token executes cryptographic functions on behalf of the target platform, which, therefore, takes the role of the *host* in terms of CoKey. CoKey combines keys stored on the host and on the token and uses Initialization Vectors (IVs) encrypted on the token in the host's cryptographic operation. This forces both the host and the token to cooperate *during the whole* encryption and decryption operation. It further effectively binds encrypted data on the host to the specific token. Afterwards, possession of the specific unclonable hardware token is required for accessing the encrypted data. We present a prototype implementation, including solutions for the host and token component. Our prototype uses the USB armory [Inv], an affordable, off-the-shelf, open source ARM computer in USB flash drive form factor, making it very convenient and applicable in real-world scenarios. CoKey effectively protects symmetric cryptography implementations from attacks targeting host-accessible keys. Our main CoKey application is the protection of data-at-rest on the target platform, e.g., for disk encryption, but the system can also be used in other scenarios involving symmetric encryption such as for Virtual Private Network (VPN) access.

The remainder of the section is organized as follows. We first discuss related work on protecting keys for symmetric cryptography in Section 7.2.1. In Section 7.2.2, we describe details of the CoKey architecture and design. In Section 7.2.5, we discuss the security of CoKey. Then, we describe our prototype implementation in Section 7.2.3 and discuss the results of our performance evaluation in Section 7.2.4.

7.2.1 Related Work

As discussed to some extent in the context of main memory encryption in Chapter 6, there are many approaches that try to protect keys from memory attacks by storing them not in RAM but in locations that are supposed to be invulnerable to cold boot [Hal+09; MS13] and DMA [Boi06; Wei12] attacks. Most of these so-called *CPU-bound encryption* schemes use special CPU registers to store keys [GM13; MFD11; Sim11]. For example, *Tresor* [MFD11] uses x86 debug registers for key storage. *TreVisor* [MTF12] additionally protects Tresor by moving it into a hypervisor. *Sentry* [Col+15] uses special SRAM located directly on the SoC of many ARM systems or cache locking to store keys and temporary data during cryptographic operations. TEEs of modern CPUs, such as ARM TrustZone and Intel SGX can be used in the same scenarios and offer a stronger in-CPU separation and key protection. For example, our memory encryption scheme TransCrypt uses a TEE for securing its encryption and another approach uses the ARM TrustZone to provide secure block devices [HWF15]. As described before, all those concepts rely on logical, in-CPU separation and are therefore susceptible to microarchitectural attacks [Koc+18; Lip+18b; Hor18; Sea15] and privilege escalations via software bug exploits. They also do not provide any means to physically detach the security relevant parts of the system to immediately prevent decryption of sensitive data. As further downsides, the described mechanisms typically use highly architecture-specific features and might have other specific weaknesses [BR12]. Nonetheless, those concepts can be combined with CoKey to protect the host-based CoKey keys.

As long as cryptographic operations are executed directly on the host, it is generally hard to protect against side channel attacks [Lam73; Tir07]. Consequentially, there are many concepts and commercial products moving bulk symmetric cryptographic operations and keys to external devices. Generic cryptographic coprocessors [Gut00], cryptographic accelerators like the *Sahara* in the i.MX53 ARM SoC [Fre12] and Hardware Security

Modules (HSMs), which are also available for USB [Yuba], help to move keys out of RAM and to protect against side channel attacks. Nonetheless, these devices often initially get their keys from the host which makes them vulnerable in case of a compromised host. Furthermore, they do not provide the CoKey key combination mechanism.

A concept to specifically protect FDE is disk self-encryption [Hit08; Sea06]. In this approach, the cryptographic operations are executed inside the disk. In a basic form, the host is responsible for initially setting the key for the encryption in a write-only register in the disk controller. Besides the fact that there might be problems with the actual implementation of the feature [AKm15; MG18], the concept is not very flexible as it is specific to FDE and the disk used and cannot protect data from different applications on the same host from each other by using different keys. Furthermore, as the key is initially set by the host, a completely compromised host can gain access to the key and hence the disk's data. An approach somewhat similar to self-encrypting disks is the *hiddn coCrypt* [Hig], an external USB thumb drive form factor device that plugs between host and an external storage medium. It provides a keypad and a display for authentication and transparently encrypts and decrypts data written and read from the attached storage medium. The concept provides key protection in case of a compromised host but shares all other problems of self-encrypting disks.

Because of performance issues, most other concepts involving external cryptography do not support bulk symmetric encryption but only provide services for key storage and authentication. Utilizing a TPM [Tru11] to secure FDE [JH10] only secures symmetric encryption keys as long as they are not used. For the actual cryptographic operation, they are exposed to the host making these approaches susceptible to all described memory and run-time attacks. The same holds for transient authentication approaches, which store encryption keys in mobile phones [Kao+12] or other wireless tokens [CN05]. Furthermore, there are a lot of products and concepts using USB tokens as smart cards [Ali11; TCS05], for secure authentication [LYQ07], as password store [Sta11] and/or for generating OTPs [Yubb]. None of these provides a solution for secure and fast symmetric cryptography.

In contrast to CoKey, none of the approaches solves the problem of securing bulk symmetric cryptography for a possibly compromised host in a way fast enough for data-intensive use cases like FDE. We further discuss and compare the security of CoKey and the relevant approaches in detail in Section 7.2.5.

7.2.2 Architecture and Design

In order to leverage physical separation, CoKey uses both physical entities of our generic architecture defined in Chapter 3: The external security token and the target platform the token is attached to. In terms of CoKey, the target platform takes the role of the *host* for which the token executes cryptographic functions. For CoKey, the host typically is a high performance device, such as a laptop or a smartphone, and the crypto token is a highly portable, small form factor embedded device like the USB armory [Inv] in our prototype

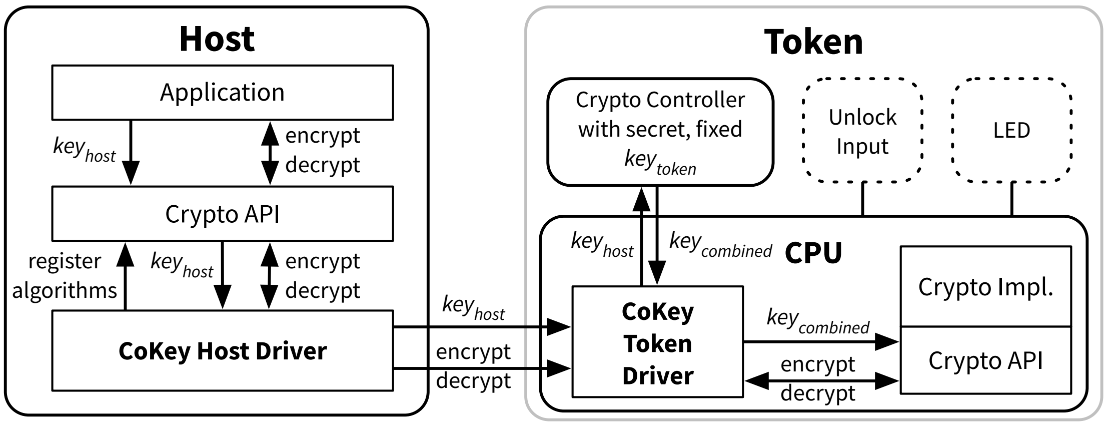


Figure 7.2: CoKey architecture.

implementation. The CoKey token and drivers offer symmetric block cipher cryptography to applications on the host. Figure 7.2 shows both entities and their main components.

The host contains three main components. The application using CoKey to encrypt and decrypt its data providing its own key key_{host}. The application can be, for example, the disk encryption layer `dm-crypt` in the Linux kernel. The second component is the Crypto API[1], a generic placeholder module for managing and using cryptographic functions in the host OS. The Crypto API allows for registering crypto algorithms to be used by applications in a generic way afterwards. The main component in the host is the CoKey driver. The driver registers the CoKey-provided algorithms at the Crypto API so that applications can use them. Afterwards, it receives key_{host} followed by encryption and decryption requests from the Crypto API and uses the Crypto API as client and the crypto token to process them. The driver is also responsible for managing concurrent accesses from different applications with different key_{host}.

On our token, there are multiple software components running on the main CPU. Additionally, there are some hardware peripherals involved in the CoKey operation. The main software component is the CoKey token driver, which handles the communication with the host and utilizes the other modules to process the cryptographic requests. This basically comprises two tasks. First, there is the key combination, in which the driver uses the dedicated crypto controller provided by the platform to derive $key_{combined}$ from key_{host}. The crypto controller contains a fixed, unique, secret key_{token} which *cannot* be read but only be *used* in cryptographic operations on the controller. Such a hardware module is very common in state-of-the-art ARM SoCs and is also present in our prototype hardware in the Freescale i.MX53 in form of the Security Controller (SCC) [Fre12]. While the fused key in such a controller might not provide the same security level as a smart card and might not

[1]We adopt the term Crypto API from the corresponding Linux module, but the underlying concept can be applied to any OS.

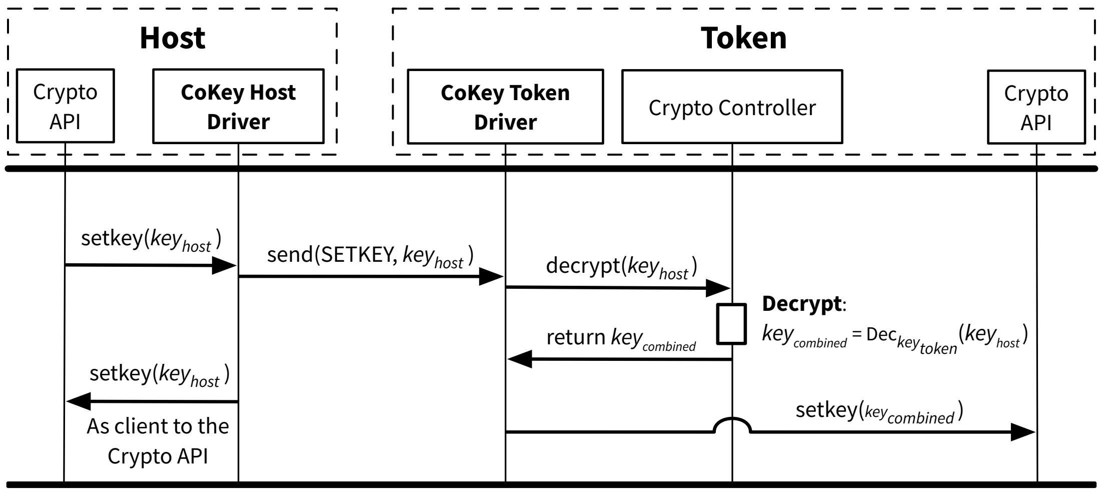


Figure 7.3: Interaction between components during CoKey key combination.

withstand a lab attack, it should be secure against software attacks targeting the crypto token. The second task of the token driver is to act as a client to the Crypto API using $key_{combined}$ to encrypt or decrypt IVs as requested by the host. The Crypto API may use software implementations or an optional crypto accelerator hardware module to speed up the actual cryptographic operations. Two other optional crypto token components are an LED, used to indicate CoKey driver activity, and some kind of secure input to authenticate the legitimate user and unlock the crypto token, as discussed in Section 3.2. The latter could be, for example, a PIN pad or a fingerprint reader. Both these components help to prevent certain attacks as discussed in Section 7.2.5. Because of its special purpose, reduced complexity and the minimalist API exposed, the crypto token is easier to secure than the host. For example, code integrity can be ensured by a secure boot [AFS97]. If the secure boot fails, access to key_{token} is prevented by the hardware.

7.2.2.1 Key Combination

Before any cryptographic operations are executed, a key_{host} must be set by the host, which is combined with key_{token} to $key_{combined}$ on the token. We call this process *key combination* and it is illustrated in Figure 7.3. In a first step, the key_{host} is set by the application on the host and forwarded via the Crypto API to the CoKey host driver. The host driver, in turn, relays the key_{host} to the token. The CoKey driver on the token does *not* use key_{host} directly for encryption but decrypts it first via the crypto controller using its secret, unique and fixed key_{token}. The resulting $key_{combined}$ is used for following encryption operations and is *never* exposed to the host. Both $key_{combined}$ and key_{host} are *not* stored permanently on the token so that after each time the token is disconnected, the key combination has to be repeated, thus, again, requiring knowledge of key_{host}.

The key combination can also be described in terms of the *Key-And*-Function described by Gifford [Gif82]. Gifford defines the *Key-And*-Function to create a derived key *dk* from two input keys *ka* and *kb* in a way that an object encrypted with *dk* can only be decrypted with *dk* or both *ka* and *kb*. CoKey basically moves one of the input keys to the external token and allows the host to provide the other input key but not to access the resulting *dk*, i.e., $key_{combined}$. Without access to both input keys or to *dk*, the host is not able to decrypt data encrypted with *dk*. The CoKey key combination effectively achieves two goals:

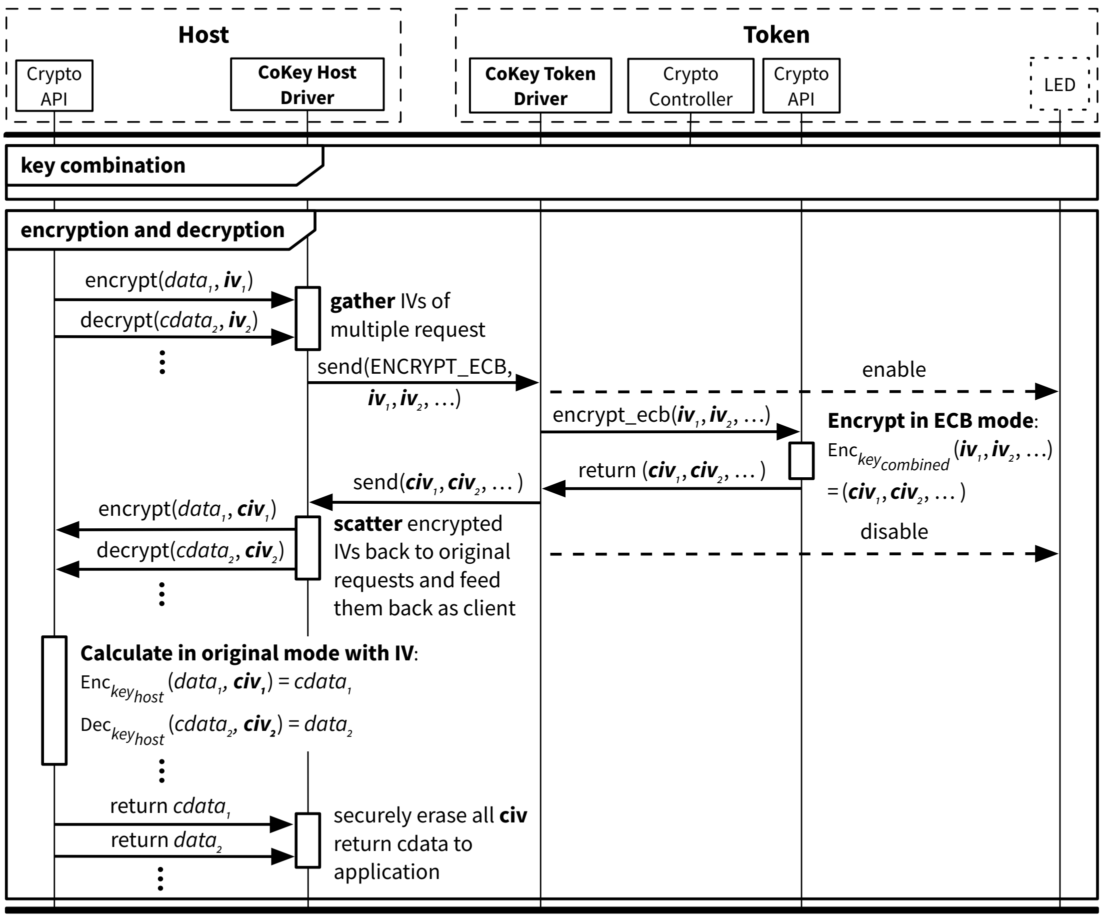
Binding. Data encrypted with a CoKey token can only be decrypted using the exact same token.

Parameterization. Data encrypted using CoKey can only be decrypted with knowledge of the key_{host} sent to the token before encryption. Different applications on a host and different hosts can have different keys key_{host} preventing applications to decrypt data from other applications without knowledge of their key_{host}. In other words, the availability of a token alone is not sufficient to decrypt data encrypted with it.

7.2.2.2 Encryption and Decryption

For the actual encryption and decryption on behalf of the host, CoKey transfers the IVs of the operation to the token where they are securely encrypted. The encrypted IVs are then used in the encryption and decryption of the actual data on the host with key_{host}. The process is illustrated in Figure 7.1 and detailed in Figure 7.4. The key-setting and combination are executed as described in the previous section and shown in Figure 7.3.

For each en-/decryption request coming from applications using CoKey through the Crypto API, the CoKey host driver extracts the IV. For improved performance, the driver gathers and concatenates a variable number of IVs before issuing a command to the crypto token to encrypt them in Electronic Code Book (ECB) mode. Using ECB mode on the token ensures that an IV always results in the same encrypted IV, independently of the order or group in which it is sent to the token. The IVs are encrypted using $key_{combined}$ and therefore show the *binding* and *parameterization* properties as described in Section 7.2.2.1. They can only be re-calculated by a host with knowledge of key_{host} and with the specific crypto token attached. The host driver receives the encrypted IVs and scatters them back into the original crypto requests replacing the IVs originally created by the host application. The actual payload data contained in the requests is then encrypted or decrypted by the host itself using key_{host}. After initializing the cryptographic operation, the encrypted IVs are removed. This makes decryption of the data encrypted with CoKey impossible without physical presence of the specific crypto token and knowledge of key_{host} and the original IVs.



Host
Token
Crypto API
CoKey Host Driver
CoKey Token Driver
Crypto Controller
Crypto API
LED
key combination
encryption and decryption
encrypt(data₁, iv₁)
decrypt(cdata₂, iv₂)
gather IVs of multiple request
send(ENCRYPT_ECB, iv₁, iv₂, ...)
enable
encrypt_ecb(iv₁, iv₂, ...)
Encrypt in ECB mode: Enc_keycombined(iv₁, iv₂, ...) = (civ₁, civ₂, ...)
return (civ₁, civ₂, ...)
send(civ₁, civ₂, ...)
encrypt(data₁, civ₁)
decrypt(cdata₂, civ₂)
scatter encrypted IVs back to original requests and feed them back as client
disable
Calculate in original mode with IV: Enc_keyhost(data₁, civ₁) = cdata₁
Dec_keyhost(cdata₂, civ₂) = data₂
return cdata₁
return data₂
securely erase all civ return cdata to application

Figure 7.4: Interaction between components during CoKey external cryptography.

The size of the data chunks encrypted with the same IV acts as a trade-off between security and performance. Big chunks can be encrypted faster as only one IV has to be transmitted and encrypted by the token. Small chunks provide flexibility and increase attack resistance as discussed in Section 7.2.5. For example, Linux disk encryption `dm-crypt` uses fixed, small chunk sizes of 512 bytes to be able to randomly access specific blocks on the disk without much encryption overhead. All host-based IV generation security measures, e.g., ESSIV, still can be applied on top of CoKey.

When using CoKey, the host should employ algorithms with modes that require IVs because CoKey will encrypt these. However, modes in which successive blocks of ciphertext can be decrypted without an IV, such as CBC mode, should be avoided because an adversary with access to any pair of blocks will not require an IV to decipher the second. Therefore, we propose Counter (CTR) mode as host-based encryption mode. In CTR mode, the IV is necessary for the decryption of *all* blocks. This ensures that the IV encrypted by the token is necessary for decryption of any block in the chunk. The encryption of the IVs on the crypto token is always done in ECB mode but the actual algorithm and key size used for it remain configurable. To protect $key_{combined}$ from a fully compromised host (see also Section 7.2.5), the crypto algorithm for IV encryption must withstand a chosen plaintext attack, which should be the case for all modern ciphers.

Ciphertexts computed with the same algorithm, mode of operation and key_{host} are different for each token and obviously different from using key_{host} in normal cryptography directly on the host. Hence, distinctive names for the algorithms provided by CoKey must be assigned on the host, including a unique identifier of the specific CoKey token used, such as its serial number.

7.2.2.3 Token Key Wrapping

The presented CoKey design intentionally directly includes the fixed, secret, unique key_{token} in the key combination process to bind the encrypted data inseparable to the specific crypto token used for their encryption. In some scenarios it might be necessary to handle the binding to key_{token} more dynamically, for example, to provide a way to have multiple crypto tokens with the ability to en-/decrypt the same data. Another use case could be having multiple virtual crypto token instances on the same physical token, e.g., for virtualized environments.

For such scenarios, we propose a dynamic key wrapping mechanism in which key_{token} is not directly used for the key combination. Instead, key_{token} is used to wrap, i.e., encrypt and protect the integrity of, one or more virtual key_{token} keys to be able to store them securely on secondary storage, e.g., an SD card. These key_{vtoken} keys can be chosen freely and are bound to the specific crypto token via the encryption with key_{token}. In a system with this extension, these key_{vtoken} keys are then used instead of key_{token} in the CoKey key combination process. In terms of Gifford [Gif82], the token key wrapping simply adds another indirection via the *Key-And*-Function to the key.

There are multiple options for initializing the key_{vtoken} keys. For virtual tokens, the keys can be randomly generated at the first startup of the crypto token, one for each virtual token. In the other use case where multiple crypto tokens should be able to en-/decrypt the same data, the same key_{vtoken} must be installed into tokens with different key_{token} keys. This means that there must be functionality to install specific keys, e.g., via some kind of provisioning process preceding the normal operation. A simple realization could enforce that keys installed into the system must be signed with a certificate that can be verified using the secure boot public key(s) on the token.

While gaining flexibility, the obvious downside of the key wrapping extension is the fact that the key_{vtoken} keys, in contrast to key_{token}, are stored in the main memory of the crypto token at some point. This makes them more susceptible to attacks on the crypto token than the key_{token} which cannot be read by software on the device. A result could be a possible cloning of crypto tokens using the key wrapping. Assuming a secure crypto

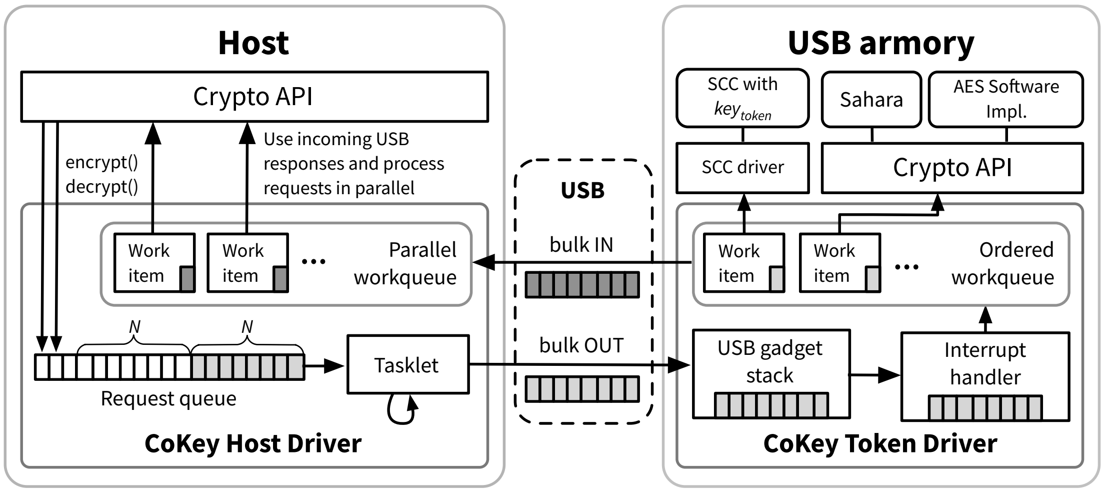


Figure 7.5: Architecture of the CoKey prototype implementation.

token, the extension adds flexibility while offering the same guarantees to the host as the basic system.

7.2.3 Implementation

We implemented a USB-based prototype realizing the described CoKey concepts. Our implementation basically consists of three parts. The platform independent USB protocol, the host driver in form of a Linux USB device driver, and the token driver in form of a Linux USB gadget driver for the USB armory as our crypto token. The host driver is completely hardware-agnostic and supports every Linux-based host with USB. The token driver is also completely hardware-agnostic except for the key combination part, which requires communication with the SCC specific to the Freescale SoC. An overview of the implementation architecture is depicted in Figure 7.5. Each driver consists of about 900 LOC. The source code of both CoKey token[1] and CoKey host driver[2] is available online.

7.2.3.1 USB Protocol

A USB device always provides one or more *configurations*, each composed of one or more USB *interfaces*. There is always only one configuration active at any time, but interfaces of the active configuration can be used in parallel. CoKey is designed on the USB *interface* level to be combinable and flexible. The interface is intentionally kept very simple and is heavily inspired by the USB Mass Storage Class Bulk-Only protocol. It uses two bulk USB endpoints, one for transmission from host to token (`OUT`) and from token to host (`IN`). The

[1]`https://github.com/jumaho/cokey-token-linux`
[2]`https://github.com/jumaho/cokey-host-driver`

protocol defines fixed length commands for the token to be sent from the host via the `OUT` endpoint. A command consists of three fields:

Command code. The command code determines which action the token should execute. There is one command for setting key_{host} and one command for encrypting IVs, i.e., `AES_ECB_ENCRYPT`. The length of the key given to the key-setting command determines the version of the algorithm used, e.g., setting a 256-bit key results in the crypto token using AES-256 for IV encryption.

Data length. Determines the length of the payload of the command to be read by the token immediately after receiving the command. In combination with the command code this implicitly also determines the length of the response.

Tag. This is either a sequence number or a random value to be able to match status words with commands.

The protocol defines fixed length status words in token-to-host direction consisting of a tag and a status code. These are sent by the token after answering a command and indicate if the command was successful. The tag must match the command's tag and can therefore reveal de-synchronization. The protocol requires strictly ordered answering of the commands so that both host and token always know how much data to expect at their inbound endpoints avoiding zero length packets to end transfers. The key_{host} is set once and is used for all following crypto commands until a new key-setting command is sent.

7.2.3.2 Crypto Token Implementation

As crypto token, we use the USB armory, an ultra-portable, USB flash drive-sized ARM Cortex-A8 computer [Inv]. The device allows developers to access the TrustZone and to fuse own keys to control the secure boot process. It runs Linux and the CoKey driver integrates into the kernel in form of a USB Gadget composite function. The Linux USB gadget subsystem [Bro] is the driver stack for implementing device-side USB functionality for USB devices running Linux. The subsystem allows drivers to be written as *composite functions* implementing exactly one USB interface. These functions can then be combined into different USB configurations creating a so called *composite device*, even at run-time via `configfs`. As composite function, the CoKey driver can therefore be used in combination with other USB interfaces in the same configuration, e.g., with a USB Ethernet interface. It is also possible to configure multiple parallel instances of the CoKey interface in the same configuration, which, combined with the key wrapping described in Section 7.2.2.3, can be useful for virtualized environments.

The token driver supports commands for key-setting and en-/decryption with AES with different key sizes in ECB mode. The driver receives commands from the host in interrupt callbacks through the Linux Gadget API and uses an *ordered workqueue* to defer handling these into process context while guaranteeing strictly ordered responses as required by the protocol. In case the connected host runs different applications using CoKey with different key_{host}, the CoKey host driver keeps track which key is currently active in the token and, if necessary, sends a command to set the correct key before an encryption command.

The key-setting command produces $key_{combined}$ by decrypting key_{host} with key_{token}. In the USB armory's i.MX53 SoC, the key_{token} is utilized via the SCC. The SCC only allows usage of the actual key_{token} after a successful secure boot and provides features to prevent further operations with key_{token} in case of an attack. For using the SCC, we had to port a driver from an old 2.6 Freescale Linux kernel[1] to the 3.19 Linux kernel we built the CoKey driver on. The corresponding SCC API calls are the only platform-specific part of the implementation, making the driver very easy to port to other Linux-based devices.

For cryptographic commands, the CoKey token driver simply calls the Linux Crypto API as client. The Crypto API automatically uses the fastest implementation of the chosen combination of algorithm and mode. With the *Sahara*, the i.MX53 SoC provides a crypto accelerator. Sahara only supports AES-128, so that the driver has to fall back to ARM-optimized software implementations for other AES forms. Unexpectedly, the evaluation in Section 7.2.4 shows that this has only very little negative impact on the performance of CoKey when using AES-256. The crypto token prototype currently does not implement the optional LED and unlocking feature.

7.2.3.3 Host Driver Implementation

The CoKey host driver acts as an algorithm provider and client to the host's Linux Crypto API and as a USB device driver for the crypto token. As soon as a token is attached to the host, the driver is probed and registers new algorithms to the host's Crypto API. It, furthermore, allocates original versions of the provided algorithms as client. As discussed in Section 7.2.2.2, CoKey algorithm names must be different from their host-only versions and should contain a token identifier, since different tokens produce different ciphertexts. The driver currently registers `ctr(aesusb)`, allowing the host to use AES in CTR mode with IVs encrypted by CoKey. In a productive version, the algorithm name should additionally include a unique identifier of the token.

The driver operation is driven by requests coming from applications via the Crypto API. These requests result in calls to corresponding callbacks in the driver in the process context of the requesting application. The driver enqueues the incoming requests in parallel and triggers execution of a tasklet. The tasklet iteratively takes requests from the queue and processes them until the queue is empty, after which it sleeps until rescheduled. The configurable maximal USB packet size N determines how many IVs can be sent to the token in one ECB encryption command. For each request, the tasklet extracts the IV and adds it to the current ECB encryption command payload. If the queue is empty and the current USB packet is smaller than $N/2$, the tasklet waits by rescheduling itself a variable number of times M before sending the underfull packet to avoid a lot of small packets. To enable

[1]`http://git.freescale.com/git/cgit.cgi/imx/linux-2.6-imx.git/`

concurrent usage by multiple applications, the driver keeps track of the key_{host} currently active on the token and dynamically switches the key, if necessary. Like all tasklets, the CoKey host driver tasklet runs exclusively, i.e., there is no need for synchronization except for accessing the request queue. Together with the token always answering commands in the order received, this ensures correct accounting of responses to commands in the host driver. This allows our current implementation to omit status words for the sake of simplicity.

Responses are received as callbacks from the Linux USB stack. A response may contain up to N encrypted IVs and for each of those a work item is generated to be handled asynchronously in a workqueue, as depicted in Figure 7.5. The work done for each item includes replacing the original IV of the corresponding crypto request with the encrypted IV and calling the Crypto API with the modified request on behalf of the original requester. The workqueue in which the items are handled, as opposed to the one used on the token, is allowed to process items in parallel. This is important to speed up the actual cryptographic operation.

The driver does not use any special hardware and should therefore be compatible with all Linux machines with USB support. The connection between host and crypto token currently uses USB 2.0 but both drivers only require minimal modification to be used with USB 3.0 as soon as there is token hardware available that supports it.

7.2.4 Performance Evaluation

For the performance evaluation, we used a Lenovo T450s Intel Core i7-5600U notebook with 12 GB RAM and a 512 GB Samsung SSD running Debian with a Linux 4.2 kernel and the CoKey kernel module. As token, we used the USB armory [Inv] with Freescale i.MX53 [Fre12] 800 MHz Cortex-A8 SoC and 512 MB RAM running a 3.19 Linux kernel with the ported SCC driver and the CoKey driver.

Our primary use case for CoKey is secure storage encryption. To test the CoKey real-world performance for this scenario, we measured the speed of `dd` writes from `/dev/zero` on file-backed virtual block devices encrypted using `dm-crypt`, the Linux FDE solution, with CoKey. We configured `dm-crypt` to use `ctr(aesusb)` and plain IVs to avoid side effects. We ran `dd` with the parameter `oflags=direct,sync`. The flag `direct` makes the `write`-syscall use the buffer directly from user space without copying it to kernel space and therefore avoids kernel caching effects. The flag `sync` makes sure that each block is completely encrypted and written to the device before continuing with the next block.

As a baseline for comparison, we consider a hypothetical *conventional crypto token* which encrypts *all* data on behalf of the host, in contrast to the fast IV-based approach of CoKey. We calculate the speed of this token based on the theoretical maximal effective throughput of USB 2.0, which is 35 MB/s [USB00]. As data must be sent *and* received, we divide the theoretical maximum by two resulting in 17.5 MB/s as maximal speed for encryption with

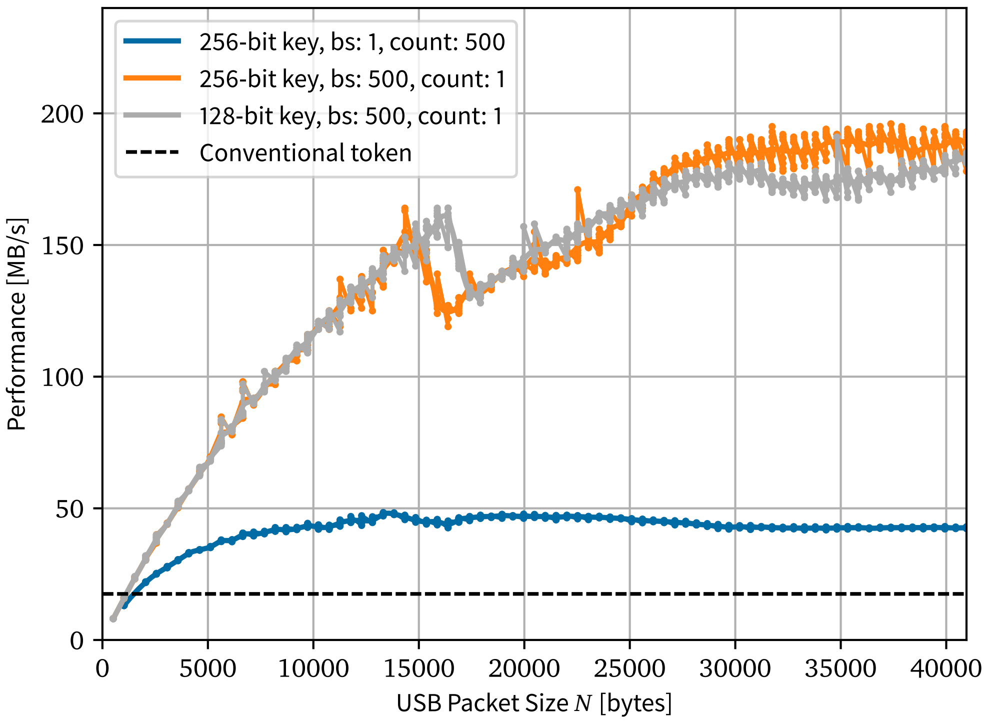


Figure 7.6: Performance of dd writes on a CoKey-encrypted disk.

the hypothetical token. This is a very conservative estimate as the theoretical maximum is never reached in real-world applications.

We tested dd with different block sizes, block counts, and key sizes for the underlying CoKey CTR-AES algorithm. As main parameter, we varied the maximal USB packet size *N* (see Section 7.2.3). The results of our tests are depicted in Figure 7.6 showing a dot for each measurement and a spline-fitted graph through the average values per packet size for each tested configuration. Additionally, the plot shows the theoretical speed of the conventional token as a dashed line. For dd with 500 blocks of 1 MB each, CoKey's performance reaches nearly 50 MB/s for *N* between 12 and 25 KB, which is already almost three times as fast as the conventional token. At first, the USB bandwidth is the bottleneck and then the dd block size, limiting concurrent requests to the crypto token. For even larger packet sizes, latency becomes an issue as cryptography on the host is blocked waiting for the encrypted IVs. A test with one 500 MB block with different key sizes shows even better performance. For both key sizes, the performance of CoKey increases almost linearly up to about 170 MB/s for an *N* of about 16 KB. In this region, the bottleneck is the USB bandwidth. Then, both key sizes show a relatively sharp drop, before regaining performance and reaching their maximum of about 200 MB/s between an *N* of 30 and 40 KB. To find out why the performance drops, we ran the same test without actually encrypting the IVs on the crypto token but only sending them back unencrypted. This test did not show the drop, so a probable explanation is that at this point the crypto token is not fast enough anymore to encrypt a whole packet of IVs before receiving the next. This means that the current encryption is interrupted by the next packet impairing the caching behavior on the device. The USB armory provides a 32 KB L1 data cache so it can contain roughly two packets if they are not bigger than 16 KB, which could explain the location of the drop. The performance increases again as the USB bandwidth is used more optimally with larger packets.

CoKey reaches up to ten times the performance of the conventional token. The results confirm that CoKey is very well usable for data-intensive use cases like FDE. Nonetheless, our test focuses on burst performance. Smaller, more random data accesses could show worse performance because of the USB latency. On the other hand, in normal operation kernel caching would hide most latency and, as a trade-off, these cases could be optimized by decreasing or dynamically adjusting the packet size N and the wait period (by adjusting the number of tasklet reschedulings M) for underfull packets. Furthermore, increasing the data per IV ratio by increasing the size of the encryption chunks can further improve performance as `dm-crypt` uses small fixed chunks of 512 bytes. Additionally, using USB 3.0 instead of USB 2.0 would improve performance without sacrificing security and require only minimal driver changes.

7.2.5 Security Discussion

In the following, we discuss the security of CoKey regarding our generic and specific attacker model defined in Section 3.3 and Section 7.1, respectively. Furthermore, we compare CoKey to normal in-host cryptography, special in-host key protection approaches like Tresor [MFD11; GM13; Sim11; Col+15; HWF15] and systems that store keys externally but provide them to the host for encryption, like in [JH10; Kao+12; CN05] or with smart cards, as discussed in Section 7.2.1.

The main goal of an attacker regarding CoKey is to gain access to keys and other cryptographic secrets used for the symmetric encryption to be able to decrypt arbitrary data on the host *without* the corresponding token being connected and unlocked. Both, the remote and physical attacker, are able to fully compromise the host. Already decrypted data residing in the host's memory may therefore be accessed by both. For normal in-host cryptography and Tresor-like systems, a fully compromised host furthermore immediately leads to disclosure of all cryptographic secrets and therefore enables the attacker to decrypt any encrypted data. For external key storage systems and for CoKey, the attacker types have to be differentiated to discuss possible consequences.

A remote attacker gains full access to encrypted data for an external key storage system as soon as the unaware user connects his key device to the host at least *once*, disclosing the actual key used for encryption. With CoKey, the remote attacker can compromise key_{host} and can use the crypto token for computing IVs *only as long as* it is connected to

Table 7.1: Comparison of attack potential (high ⊖, limited ○, low ⊕) for CoKey and related approaches with different attackers.

	Remote Att.		Physical Att.		
	Full	Partial	Host	Token	Both
Normal	⊖	⊖	⊖	-	-
Tresor-like	⊖	○	⊖	-	-
Key Store	⊖	○	○	⊕	⊖
CoKey	○	⊕	⊕	⊕	○

the host and unlocked by the legitimate user. If an attacker is able to predict unencrypted IVs, he can pre-compute encrypted IVs valid independently of the actual data but only for the specific combination of IV and key_{host}. The impact of the attack is limited in several ways. The actual key used for IV encryption is *never* exposed to the attacker, requiring him to do IV pre-computation “online”. The pre-computation requires prediction of the IV and knowledge of the matching key_{host} (which might not even be known to the host at that point). Furthermore, the pre-computation can be hardened by using host-based IV generation security measures on top of CoKey and by configuring the encryption chunks to be small. Additionally, it depends on the use case if IV prediction is even possible. In case of FDE, knowledge of key_{host} might be necessary for an IV prediction and in encrypted communication cases, a prediction might not be possible at all. Last but not least, large scale IV pre-computation takes time, which can reduce the impact of an attack substantially. Leveraging this aspect, the encryption chunk size or the speed of the token can be reduced providing a trade-off between attack potential and performance. The remote attacker cannot influence when the token is connected and unlocked. Furthermore, an observant user might notice an attack as unexpected activity of the crypto token indicated by the token’s (optional) LED. In summary, CoKey provides a substantial security improvement against a very capable remote attacker.

For the physical attacker, we differentiate whether he has physical access to the host, the token or both. With access to the host but not to the token, the physical attacker can fully compromise the host and therefore break normal in-host and Tresor-like encryption as described before. For external key storage systems, it depends on whether the host was compromised while holding the key in memory. In such a case these systems are also insecure. With CoKey, decryption is *only* possible *as long as* the crypto token is connected and unlocked. CoKey is therefore always secure against a physical attacker without access to the token.

A physical attacker in possession of the crypto token but without access to the host cannot decrypt any data because the crypto token does not contain any actual payload data. Even if the attacker is able to unlock the token, the hardware-based key_{token} is secure, preventing cloning of the token as described in Section 7.2.2. Because of the key combination, it is not even possible to store a $key_{combined}$ for later usage or to pre-compute *any* IVs without access to the corresponding key_{host}.

The most powerful physical attacker is in possession of host *and* token. For the external key store system, encrypted data is only secure against such an attacker if the attack happens when the key store is locked and the encryption key is currently not in the host's memory (no recent encryption or decryption). Furthermore, the system might be at risk if the key store uses a locking mechanism that is unlocked via the host, which is, for example, often the case for smart cards. With CoKey, the physical attacker with access to host and token can only compute encrypted IVs and therefore access arbitrary encrypted data on the fully compromised host *as long as* the token is *attached and unlocked*. Furthermore, as described for the remote attacker, such an attack is limited in several ways. In comparison to the remote attacker, the physical attacker has the advantage that the LED cannot be used for attack detection but the disadvantage that he is highly unlikely to gain possession of the token in unlocked state.

In case of a partially compromised host, for example, running a process or VM controlled by an attacker, the parameterization property of the CoKey key combination ensures that access to a key_{host} of a part of the system only allows decryption of this part's data and only as long as the token is connected and unlocked. In a partially compromised host, microarchitectural attacks, such as cache-timing attacks [Ber05; YF14; Koc+18; Lip+18b; Hor18], might expose keys used in uncompromised domains to the attacker. For in-host approaches, i.e., normal or Tresor-like cryptography, this might reveal all cryptographic secrets. For CoKey, only key_{host} is at risk with all restrictions that apply to such an attack, as discussed before. First and foremost, such an attacker cannot gain access to $key_{combined}$ and therefore has to decrypt data "online".

The results are summarized in Table 7.1. Our remote and physical attackers always succeed to attack data encrypted with Tresor-like approaches as well as with normal in-host cryptography. External key storage systems are secure until they expose their keys to the host *at least once*. CoKey is secure in all these cases and allows only partial attacks for cases in which the specific crypto token is connected and unlocked.

7.3 TrustID: Token-based Identity Derivation and Storage

As discussed in the beginning of the chapter, protecting asymmetric keys poses different challenges than protecting symmetric keys. In contrast to symmetric cryptography where full externalization of the operation is not feasible performance-wise and requires advanced trade-offs such as CoKey, asymmetric cryptography rarely happens and can therefore be completely moved from our target platform to an external device, such as our token. Together with certificates and a Public Key Infrastructure (PKI), asymmetric keys can serve as strong user IDs, substantially improving security compared to traditional identification methods, such as passwords. But the protection of those IDs poses new challenges besides the security of the actual cryptographic operation. Especially the question of how to securely *generate* IDs requires a lot of thought. In the following, we introduce TrustID, a concept for secure, ID generation and usage on an external token on behalf of a physically connected but separated *host*, i.e., our target platform. In our concept, an ID basically consists of an asymmetric key pair and an associated certificate containing additional information about the context and usage of the identity. We define the following constraints and design goals for TrustID:

- Asymmetric keys must be generated directly on the token and never leave the token.
- The generated IDs must be verifiable by third parties.
- The token can only communicate via the host it is connected to and provides *no* independent user input or output. This also excludes any kind of secure and independent unlocking mechanism.
- The strong attacker model defined in Section 7.1 applies, meaning that we assume that the host might be fully compromised.

For TrustID, the user possesses *one* strong, long-term issued, generic root identity, called *RootID*. This RootID could, for instance, be a government-issued Electronic Identity Card (EIC). The user then uses this RootID and his host device to derive context-specific IDs, which are stored securely in the connected token. The TrustID ID derivation protocol relies on a trusted third party, the TIP, to decide about ID requests based on the information stored in the RootID. Derived IDs can be used via all interfaces of the host, e.g., via Near Field Communication (NFC) but also via the Internet, while the RootID remains in the user's home or some other secure location, avoiding the risk of loss. Assuming a possibly malicious host and a token without independent user I/O makes traditional user authorization methods, such as a PIN entry, insecure for authorizing an ID derivation. We therefore propose a *secure combined PIN entry* mechanism which allows establishing a secure channel between token and RootID using their corresponding secret PINs without the user entering them in clear on the untrusted host. Hence, the security of the ID derivation mechanism and the resulting ID does *not* rely on the security of the host. If the user loses the token, he can use his RootID to revoke the IDs stored on it.

Not requiring a token with independent user I/O and the assumption of a possibly malicious host allows for some powerful use cases. The host could, for instance, be an off-the-shelf smartphone and the token an additional SE. Then, a typical use case could be the following: The user wants to store a context-specific identity in his smartphone, for example, a virtual bank card, to get access to this specific service with his smartphone and without carrying his RootID and/or physical bank card. The user starts the TrustID app on his smartphone, which he previously equipped with an SE token, and derives a bank card ID into the SE. Afterwards, he can use the ID via the smartphone's interfaces, e.g., via NFC, instead of the physical bank card. This application scenario is illustrated in

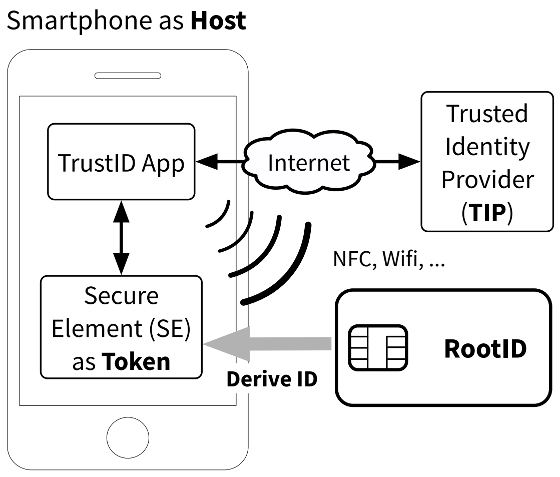


Figure 7.7: TrustID application scenario.

Figure 7.7. Most users already own a smartphone and carry it on them, making it the most convenient place to store IDs the user needs on a daily basis. The SE token can be flexibly added to the smartphone in different form factors, including embedded and wireless SEs, microSD cards and Universal Integrated Circuit Cards (UICCs). The concept also takes into consideration that there might be several SE manufacturers who are allowed to produce SEs prepared for the TrustID system. Last but not least, stronger protections of user IDs in smartphones is increasingly important since the past years show the growing relevance of smartphones as targets for diverse attacks [Dav+10a; Avi+10].

The remainder of the section is organized as follows. We first discuss some related research approaches in Section 7.3.1. Afterwards, we introduce the TrustID architecture in Section 7.3.2 and its ID derivation and usage protocols in Sections 7.3.3, 7.3.4 and 7.3.5. In Section 7.3.6, we shortly introduce our prototype implementation and in Section 7.3.7, we systematically discuss TrustID's security.

7.3.1 Related Work

In the following, we discuss related work in the context of ID protection, mainly focusing on the smartphone application scenario.

Leicher et al. [LSS12] provide an OpenID-based approach for mobile Single Sign On (SSO) across different devices. In contrast to our approach, their concept relies on the Mobile Network Operator (MNO) as root for the derivation of mobile identities and on a fully trusted mobile device.

Urien et al. [UMK11] provide a mobile ID in form of an Extensible Authentication Protocol (EAP)-Transport Layer Security (TLS) smart card. Similar to our approach, they store a key pair as identity credential in the smart card. For securing the communication, they use the TLS stack of those special smart cards to form secure channels between the different entities of their protocol. However, they do not cover a compromised smartphone in any case.

Hyppönen [Hyp08] describes a protocol for deriving an ID from an identity issuer sending the credentials to the SE over an identity proxy (a mobile phone). Nevertheless, this approach does not deal with identity theft by the means of a relay attack as described in our security evaluation (see Section 7.3.7) or with a compromised mobile device in general. Chen et al. [Che+11] propose a mobile payment ID derived from the Citizen Digital Card (CDC), a governmental PKI-based ID card, onto an NFC smartphone. This is similar to our ID derivation application scenario. However, their protocol relies on the MNO as trusted ID provider, owning the SE in their scenario (the Subscriber Identification Module (SIM) card). Furthermore, the PIN for the CDC is entered in clear, allowing an attacker compromising the smartphone to eavesdrop it.

Dmitrienko et al. [Dmi+12] provide an approach for delegable access control utilizing NFC-enabled smartphones. Their system is not based on a RootID, but relies on users directly delegating their credentials to each other. In contrast to our approach where the smartphone can be completely untrusted, they rely on a large TCB.

Summarizing, none of the previous approaches covers the full process of deriving multiple identities from a strong root identity employing a trusted identity provider, which might be integrated in a governmental PKI, while taking a possibly compromised host, such as the user's smartphone, into account.

7.3.2 Architecture

The TrustID architecture is depicted in Figure 7.8. It builds upon our generic system architecture described in Chapter 3 and extends it with two components, the TIP and the RootID. The components of the TrustID architecture are described in the following:

Host. The target platform as defined by our generic system architecture. A smartphone in the main TrustID application scenario.

Token. The token as defined by our generic system architecture. An SE (e.g., a microSD card or UICC) in the main TrustID application scenario. For TrustID, the token must contain key material verifiable by the TIP, as discussed in detail in Section 7.3.3. The key material is *not* user-specific, but is used to provide a guarantee that the newly derived ID is stored in a valid token. Therefore, no personalization or pre-provisioning is necessary.

RootID. The RootID is a long-term issued smart card, securely storing the personal data of the user, such as name and date of birth. The RootID must contain key material to authenticate itself towards an entity requesting the user's data and to ensure that only authorized entities may actually read the data. A typical example for

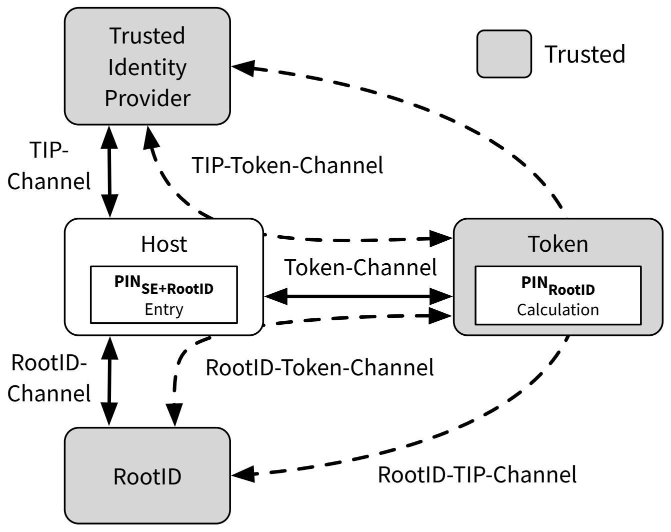


Figure 7.8: TrustID system architecture.

RootIDs fulfilling these requirements are government-issued EICs like the German identity card Neuer Personalausweis (nPA) used in our prototype implementation (see Section 7.3.6).

Trusted Identity Provider. The TIP is the back-end of the TrustID architecture and is remotely connected to the host, e.g., via the Internet. In the ID derivation protocol, the TIP must decide if an ID request by a user is granted based on the user's RootID and the token connected to the user's host. The TIP is also responsible for generating the certificate for the new ID.

During TrustID ID derivation, several logical channels are established. Figure 7.8 shows all channels between the entities. We differentiate between channels that conceptually directly connect two entities (solid lines) and channels that are established between two entities through a third TrustID entity (dashed lines):

RootID-Channel. This channel connects the host to the RootID. It is *unprotected* and is realized differently depending on the host's hardware and RootID's form factor. For example, the channel could be established using an NFC connection to a contactless smart card as RootID.

Token-Channel. This channel connects the host to the token. It is *unprotected* and established differently depending on the token's form factor. For example, a smartphone host could be connected to a token in form of a microSD card SE via SDIO.

TIP-Channel. This channel connects the host to the TIP. The channel is established remotely and does *not* have to be protected, but can optionally use TLS.

TIP-Token-Channel. This channel connects the TIP and the token. It is mutually authenticated and encrypted and established based on static asymmetric keys present in the involved entities.

RootID-Token-Channel. This channel connects the RootID and the token. Similar to the TIP-Token-Channel, this channel is mutually authenticated and encrypted. It is established using the PIN mechanism described in Section 7.3.4.

RootID-TIP-Channel. This channel connects the RootID to the TIP. The channel is mutually authenticated and encrypted. It is established through the TIP-Token-Channel and the RootID-Token-Channel using the static key material in RootID and TIP. After its establishment, any further communication between TIP and RootID is routed through this channel, which therefore replaces the two previous logical channels.

Attacks on these channels are discussed in the security evaluation in Section 7.3.7 after introducing the ID derivation protocol in detail in the next sections.

7.3.3 Protocol Prerequisites

The TrustID concept requires specific static key material in the RootID and the token for its ID derivation protocol. First, the RootID must contain an asymmetric, e.g., Elliptic Curve Cryptography (ECC) or RSA, key pair consisting of the private key SK_{RootID} and public key PK_{RootID}. For the public key PK_{RootID}, the RootID additionally contains a certificate issued by the TIP or some other Certificate Authority (CA) that can be verified by the TIP according to the PKI used. Furthermore, the RootID contains the public key of the root of the PKI. In our protocol description, the TIP is the issuer of the certificate $Cert_{TIP}(PK_{RootID})$ and is also the root of the PKI, which simplifies the protocol discussion. Access to the RootID is protected by a PIN named PIN_{RootID}.

Second, also the token must contain an asymmetric key pair (SK_{Token}, PK_{Token}), a certificate for its public key and the public key of the CA used. In a typical scenario, the certificate is issued by the manufacturer of the token who guarantees certain physical security properties of the token and the generated private key SK_{Token}. The token contains a PIN named PIN_{Token} to be used in the ID derivation protocol. Depending on the chosen usage protocol (see Section 7.3.5), it furthermore contains a PIN_{Use} protecting usage of the derived IDs. Again, for simplicity in our description, we assume the TIP to be the CA. In summary, the requirements regarding the overall PKI are:

- The TIP must be able to verify the certificates of both, the RootID and the token.
- The TIP must be able to provide a certificate or a certificate chain verifiable with the root CA keys in the token and RootID (in our protocol description PK_{TIP} for both).

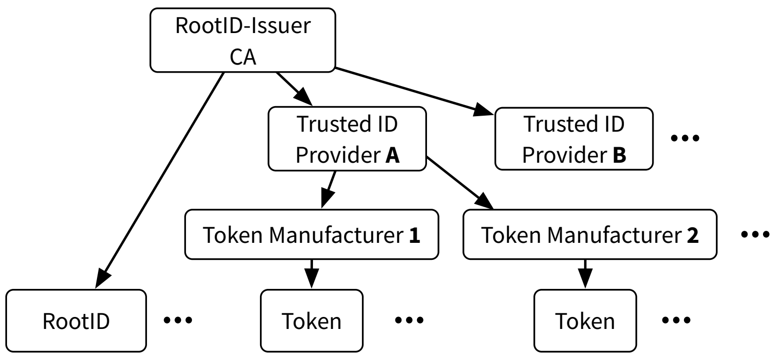


Figure 7.9: TrustID PKI example.

A simple example for a PKI fulfilling these requirements is depicted in Figure 7.9. The shown PKI allows for a system in which there are different TIPs, certified by the RootID issuer. The TIPs, in turn, certify different token manufacturers.

7.3.4 Protocol

With a RootID and token meeting the introduced requirements, a user can derive IDs as described in the following. The ID derivation protocol is depicted in Figure 7.10. The figure also summarizes the results from the previous section, showing the necessary static key material below each entity. At the end of the protocol, a new identity is stored securely in the token connected to the host. In the following, we discuss the steps of the protocol.

TIP-Token-Channel Establishment. In a first step, the token and TIP establish the mutually authenticated and encrypted TIP-Token-Channel. The channel is established using an unspecified protocol based on the static keys of both entities. Different protocols are possible, for example, a mutually authenticated certificate-based form of TLS. Since the token can only communicate via the host, all communication must be relayed by the host and possibly be converted between different formats, for example, from Application Protocol Data Units (APDUs) to Hypertext Transfer Protocol (HTTP) requests and vice versa.

Secure Combined PIN Entry. In the second step, the user must authorize the ID derivation. For this authorization, we introduce a special PIN entry mechanism which builds the basis for establishing the RootID-Token-Channel in the next step without the host being able to eavesdrop it. As described in Section 7.3.3, both the RootID and the token each provide a secret PIN (PIN_{RootID} and PIN_{Token}) only known to the user. According to our attacker model (see Section 7.1), the host might be compromised. Instead of entering one of the PINs directly via the host and therefore possibly exposing it to an attacker, the user does a simple calculation on both PINs to derive a *combined* PIN named $PIN_{Token+RootID}$. We

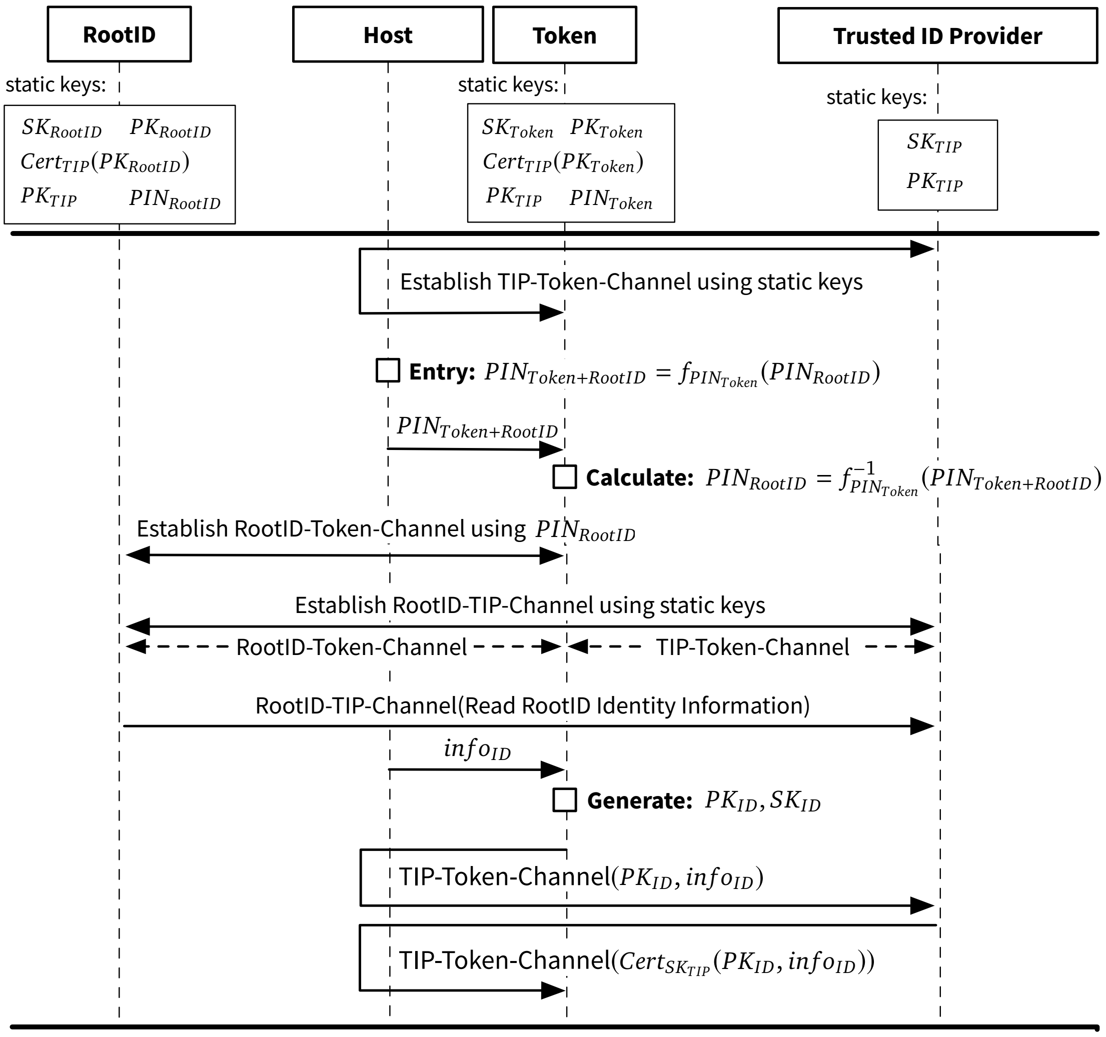

RootID
Host
Token
Trusted ID Provider
static keys:
SK_{RootID} PK_{RootID}
$Cert_{TIP}(PK_{RootID})$
PK_{TIP} PIN_{RootID}
static keys:
SK_{Token} PK_{Token}
$Cert_{TIP}(PK_{Token})$
PK_{TIP} PIN_{Token}
static keys:
SK_{TIP}
PK_{TIP}
Establish TIP-Token-Channel using static keys
Entry: $PIN_{Token+RootID} = f_{PIN_{Token}}(PIN_{RootID})$
$PIN_{Token+RootID}$
Calculate: $PIN_{RootID} = f^{-1}_{PIN_{Token}}(PIN_{Token+RootID})$
Establish RootID-Token-Channel using PIN_{RootID}
Establish RootID-TIP-Channel using static keys
RootID-Token-Channel
TIP-Token-Channel
RootID-TIP-Channel(Read RootID Identity Information)
$info_{ID}$
Generate: PK_{ID}, SK_{ID}
TIP-Token-Channel($PK_{ID}, info_{ID}$)
TIP-Token-Channel($Cert_{SK_{TIP}}(PK_{ID}, info_{ID})$)

Figure 7.10: TrustID identity derivation protocol.

denote the calculation as function f:

$$PIN_{Token+RootID} = f(PIN_{Token}, PIN_{RootID})$$

A function f should be chosen that is easily calculable by the user. The calculation must be invertible by the token with knowledge of its own PIN_{Token}. Since a function with two arguments is not formally invertible, we formally parameterize f with PIN_{Token} into a token-specific function $f_{PIN_{Token}}$ only taking PIN_{RootID} as single argument. Since this function requires knowledge of PIN_{Token}, only the corresponding token can calculate it. The token receives the combined $PIN_{Token+RootID}$ and calculates PIN_{RootID} with its token-specific inverse function $f^{-1}_{PIN_{Token}}$. This means that only the token the user intends to store the new ID in is able to calculate the secret necessary to establish a channel to the user's RootID. After this calculation, token and RootID share a secret password in form of PIN_{RootID} which never went through the possibly malicious host in plaintext. This password is then used in the next step to establish the RootID-Token-Channel. The combined PIN entry binds the token and RootID to each other for the protocol run, preventing certain relay attacks as discussed in the evaluation in Section 7.3.7.

To clarify the PIN mechanism further, we discuss the following simple but realistic example for the PIN calculation function f:

$$\begin{aligned} f(x,y) &:= x + y \\ f_{PIN_{Token}}(x) &:= PIN_{Token} + x \\ f^{-1}_{PIN_{Token}}(y) &:= y - PIN_{Token} \end{aligned}$$

In a concrete example with this specific PIN function, we assume $PIN_{Token} = 123456$ and $PIN_{RootID} = 345678$. The user uses the introduced function and calculates

$$PIN_{Token+RootID} = f_{PIN_{Token}}(345678) = 469134$$

and enters it through the untrusted host. The token then calculates

$$PIN_{RootID} = f^{-1}_{PIN_{Token}}(469134) = 345678$$

to obtain the shared secret for the channel establishment with the RootID. This example also shows that the calculation by the user does *not* have to be complicated, especially when enforcing typical protections for the PINs. To this end, both, token and RootID, should implement try counters for the PIN entry to prevent brute force attacks.

RootID-Token-Channel Establishment. After the secure combined PIN entry, the token and RootID share a secret password PIN_{RootID}, which they use in the next step to establish the secure RootID-Token-Channel. For that, a balanced Password-Authenticated Key Agreement (PAKE) protocol is used [BM92; BMP00; GL01]. Such a protocol allows the two parties to derive a common cryptographic key based on a common human-memorable password. In our aforementioned example and implementation scenario where the RootID is the German ID card nPA, the PAKE protocol used is the Password Authenticated Connection Establishment (PACE) protocol [Fed12]. In this context, the token takes the role of a terminal.

Read-out of the RootID Information. After the establishment of the RootID-Token-Channel and the TIP-Token-Channel, the RootID can be reached by the TIP through a relay in the token without data being sent unencrypted or unauthenticated at any point. The token acts as a channel endpoint for both channels and tunnels the communication accordingly. This tunnel is used for establishing another channel, the RootID-TIP-Channel, which allows end-to-end encrypted and mutually authenticated communication between RootID and TIP, i.e., without even the token being able to eavesdrop information. Just as for the TIP-Token-Channel, the available key material in RootID and TIP allow a certificate-based channel establishment, for example, with TLS. The newly set up RootID-TIP-Channel then replaces the two channels ending in the token for all communication between RootID and TIP and is used to read the user's identity information stored in the RootID.

ID Generation, Validation and Transmission. As soon as the TIP has read the RootID identity information, the actual generation of the derived ID is initiated by the host. For that, the host encapsulates information regarding the identity requested by the user into an $info_{ID}$ data structure and sends it to the token. The token stores the $info_{ID}$ object and generates a new key pair (PK_{ID}, SK_{ID}) for the new ID. It is important that the key pair is generated directly on the token and that the private key *never* leaves the token. Afterwards, the public key PK_{ID} and the $info_{ID}$ object are sent to the TIP encrypted and authenticated through the previously established TIP-Token-Channel. The channel ensures that the TIP *only* accepts a public key for certification which was generated in the particular token with which the protocol was initiated.

The TIP is now responsible for validating the ID request based on the information read from the RootID and the information to be included into the newly derived ID, i.e., the $info_{ID}$ object. The actual decision mechanism for granting or refusing an ID request is highly context-dependent and therefore not further specified here. The TIP could, for example, consult a back-end at a bank before granting a virtual bank card ID to a user.

If the TIP grants the request, it generates a certificate containing the public key for the newly derived ID together with the $info_{ID}$ object. Thus, the ID is always bound to the specific context it was created for. The certificate is then sent via the TIP-Token-Channel to the token where it is stored together with the ID's key pair.

7.3.5 ID Usage

With the asymmetric key pair and the corresponding certificate, containing context-specific information ($info_{ID}$), the newly derived ID can be used in diverse, application-specific ways. In the following, we focus on scenarios in which the host and its token are used to authenticate the user towards a physical *terminal*, for example, to open a door lock or access a bank account. Basically, there are two different cases for the authentication with a physical terminal: Either only the ID authenticates itself towards the terminal or the authentication is done mutually. In the first case, it is sufficient for the terminal to have access to the certificate chain with which the ID was certified to be able to verify the ID. In the second case, the terminal infrastructure must be included into the PKI (see Figure 7.9), so that every terminal gets an own certificate that can be verified by the token using its pre-installed public root key, e.g., PK_{TIP} in our concept.

Depending on their value, IDs stored on the token might additionally require protection from unauthorized usage, especially considering that our attacker model (see Section 7.1) assumes a possibly malicious host. To this end, we propose two different PIN-based schemes, described in the following.

Basic Protection Scheme. In the first scheme, the usage of the IDs stored on the token is protected by a *single* PIN named PIN_{Use}. This PIN is different from the PIN_{Token} used during ID derivation. When an ID is requested by a terminal, the user approves the request by entering this PIN_{Use} on the *terminal*, which is assumed to be trustworthy. The PIN_{Use} can then be used as shared secret for a PAKE protocol between token and terminal as described before for the ID derivation protocol. The untrusted host is not able to gain knowledge of PIN_{Use}.

Reverse Combined PIN Entry. The first scheme requires the terminal to be fully trusted as it gains knowledge of the single PIN_{Use} used for authorizing requests for all IDs stored in the token. The second scheme allows us to restrict this trust in the terminal with the *reverse combined PIN entry*. This mechanism basically reverses the PIN entry mechanism of the ID derivation. The steps are the following:

1. The user initiates the usage of a specific ID. The terminal and/or the host send a request for the ID to the token.

2. The token generates a random *one-time valid* PIN_{ID} and calculates $f^{-1}_{PIN_{Token}}(PIN_{ID})$ as previously discussed in Section 7.3.4 to produce a combined $PIN_{Token+ID}$.

3. The combined PIN is presented to the user via the host. The user is able to reverse the calculation with the knowledge of the secret PIN_{Token} using $f_{PIN_{Token}}$.

4. The user enters the calculated PIN_{ID} on the terminal, which is then able to use it in a PAKE protocol to request the *specific* ID on the token.

The first approach is easily usable requiring the user only to memorize an additional PIN_{Use} but requires the terminal to be trustworthy. The second approach has the advantage that the terminal only gains knowledge of a one-time valid shared secret. It furthermore restricts the access to only one specific ID. Nevertheless, the user *cannot* be sure that the used ID is the one he expects without any trust in either the terminal display or the host display. A partial solution to this problem could be to require both entities to show which ID is about to be accessed. Furthermore, it must be ensured that an attacker might not gain access to *both*, the $PIN_{Token+ID}$ displayed on the host and the PIN_{ID} entered on the terminal. This would allow the attacker to calculate the secret PIN_{Token} only known to the user and the token. Both attacks require control over the terminal *and* the host at the same time, significantly reducing their potential.

7.3.6 Implementation

We implemented a prototype of our main TrustID application scenario with a smartphone as host and an SE as token. Our prototype uses the government-issued German ID card nPA as RootID, an Android smartphone running the TrustID App and a JavaCard-based microSD card SE.

Trusted Identity Provider. Due to requirements imposed by the eID protocol [Fed12] for accessing data stored in the nPA, the TIP must cooperate with another entity, the eID-Server, which reads the nPA data on behalf of the TIP. All communication is conducted via HTTP(s) requests/responses, which the TrustID app on the smartphone translates into APDUs to be sent to the SE token or RootID and vice versa. The TLS protection on this layer is optional as the security of the communication is enforced by the TrustID protocol.

Applet on the SE. The TrustID prototype applet is running on a G&D Mobile Security Card SE 1.0, using the JavaCard API in version 2.2.1. The main task of the TrustID applet is to store IDs and to provide functions to derive, manage and use them. An ID in the applet basically consists of an asymmetric RSA key pair (currently 2048 bit) and a certificate (X.509) issued by the TIP for the public key, additionally containing information about the ID ($info_{ID}$) as X.509 extensions.

As limitation of the current version of the prototype, the RootID-Token-Channel does not terminate in the JavaCard applet but directly in the TrustID Android app. This is mainly due to the fact that the JavaCard API of currently available microSD card SEs does *not* support the specific ECC operations necessary for the PACE protocol.

Android App. The TrustID Android prototype app runs on a Samsung Galaxy SIII (i9300) device with Cyanogenmod, a custom Android distribution. The usage of IDs is realized via the Card Emulation feature provided by the Cyanogenmod 10 release for the Galaxy SIII. This feature allows the smartphone to act as a contactless, NFC-enabled, smart card forwarding all incoming communication to software and vice versa. The TrustID app is the endpoint for this function and, in turn, relays the communication to the SE. In order to use one of the IDs stored on the SE, the user has to activate it and place the smartphone on a terminal. The app's Graphical User Interface (GUI) shows a list of IDs stored on the SE and allows the user to delete, activate and derive new IDs.

The establishment of the secure channel between eID-Server and nPA is realized in the app using the open source library eIDClientCore [BeI]. Because of incompatibilities between the NFC hardware of the nPA and the Galaxy SIII, the prototype uses a *relay host*

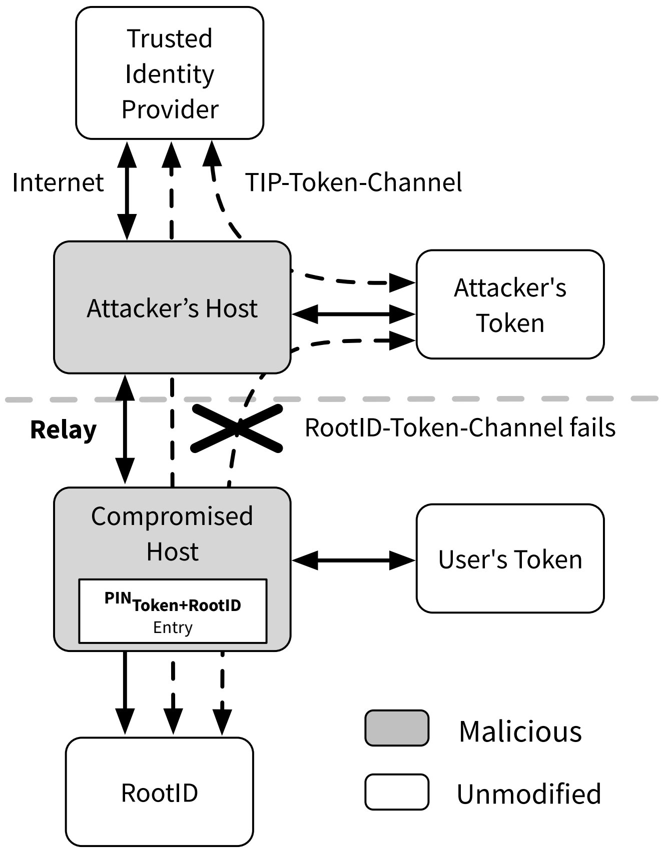


Figure 7.11: Relay attack and TrustID defense.

to enable their connection. An off-the-shelf smart card reader is connected to this relay host, which reads the nPA and relays communication to the smartphone via WLAN.

7.3.7 Security Discussion

In the following, we analyze the security of our TrustID ID derivation concept. Following our attacker model (see Section 7.1), TIP, token and RootID are assumed to be untampered but the host might be completely compromised. In the TrustID architecture (see Figure 7.8), we differentiate between *direct* channels, i.e., the Token-Channel, the TIP-Channel and the RootID-Channel (solid lines in Figure 7.8), and *indirect* channels, i.e., the TIP-Token-Channel, the RootID-Token-Channel and the RootID-TIP-Channel (dashed lines). Their security is discussed in the following:

Direct Channels. All direct channels are unprotected, i.e., provide neither authentication nor encryption or integrity protection. They can be thought of as physical connections and attacks are prevented by the logical secure channels on top of them.

TIP-Token-Channel. This channel is established using the static key material in a TLS-similar protocol. It is therefore protected against eavesdropping, manipulation and replay attacks. With the authentication of the token, the TIP can be sure that the channel ends in a valid token but at this point there is *no* association to a specific user. This might lead to a relay attack as depicted in Figure 7.11. The attacker modifies the user's host to relay the communication remotely to an attacker's host also containing a valid token. The goal of the attack is to have the user authenticate an ID derivation (with RootID and secret PINs) but storing the new ID in the attacker's token. Since the TIP-Token-Channel is only guaranteed to be established with a *valid* token, the attacker is able to set it up to his own token. Still, the attack fails with the RootID-Token-Channel and the secure combined PIN entry as described in the following.

RootID-Token-Channel. TrustID uses the combined PIN entry mechanism (see Section 7.3.4) to establish the RootID-Token-Channel. The mechanism ensures that the shared secret PIN_{RootID} can only be obtained by the token for which the user calculated the combined $PIN_{Token+RootID}$. This effectively binds a token chosen by the user and his RootID to each other for one protocol run. Since the attacker's token contains a different PIN_{Token}, it is *not* able to calculate the shared secret based on the intercepted $PIN_{Token+RootID}$. Hence, it cannot establish the RootID-Token-Channel. In other words, the RootID will not accept any connection attempt from the attacker's token, effectively preventing the relay attack as shown in Figure 7.11. By entering the combined PIN, the user furthermore guarantees the physical proximity of the three entities host, token and RootID, preventing also other relay attacks between these. Eavesdropping, replay and manipulation are prevented by the PAKE protocol establishing the channel.

RootID-TIP-Channel. This channel is set up *through* the other secure channels using a TLS-similar, mutually authenticated certificate-based protocol. By establishing the channel through the RootID-Token-Channel and TIP-Token-Channel, the different protocol steps are bound together, effectively preventing interleaving attacks.

The attacker might be able to modify the ID request generated by the user, resulting in a different derived ID than intended by the user. Even if the attack is not detected by the TIP when validating the request against the RootID information, the wrongly derived ID would still be safely stored in the user's token. The attacker might be able to continuously generate valid IDs on a compromised host containing a valid token as soon as he gains knowledge of $PIN_{Token+RootID}$. However, this attack is only possible as long as the RootID is in reach and, again, can be considered uncritical as the attacker does not gain control over the derived IDs. The attacker might also manipulate the displaying of IDs to the user. We consider this attack uncritical with the same argument as for the previous attacks. The same holds for all kinds of manipulations of communication data going through the host, including $PIN_{Token+RootID}$.

7.4 Summary

In this chapter, we explored ways to protect cryptographic keys against advanced attacks on the target platform by leveraging physical separation with our token. Our advanced attacker model for this chapter assumes that our target platform can be completely compromised, including all logical, in-CPU separations. Our strong attacker model is motivated by advanced side channel attacks, such as *Meltdown* and *Spectre*, showing that shared resources can almost always be exploited to cross separation boundaries.

The protection of symmetric and asymmetric keys poses different challenges. Symmetric cryptography is often performance-critical and, hence, externalizing symmetric keys to our token requires a concept with an efficient trade-off. Asymmetric cryptography is often already completely externalized, for example, when using smart cards for authentication. Therefore, in this context, we focused on the secure generation of asymmetric keys for a user's identities in our token.

First, we presented CoKey, a concept and design for secure and fast external symmetric cryptography using a detachable token. Our concept combines the security of external cryptography with the performance of in-host cryptography. It allows a host to encrypt large amounts of data, such as for FDE, in a fast and efficient way while binding the security of the encrypted data to the presence of a specific hardware token. The evaluation of our Linux-based prototype shows that the system is able to provide encryption ten times as fast as a comparable conventional token-based solution.

In the context of asymmetric cryptography, we introduced the TrustID architecture and protocol. Our approach allows secure storage, derivation and usage of multiple context-specific IDs with a possibly malicious host, i.e., target platform, utilizing our token as credential store. As core of our concept, we introduced the secure combined PIN entry mechanism for authorizing ID derivations without relying on the trustworthiness of the host. Furthermore, we introduced a reverse combined PIN entry able to protect the newly derived IDs from unauthorized access. We presented our prototype running on a Samsung Galaxy SIII host utilizing a microSD card SE as token and the German identity card nPA as RootID. In our systematic security discussion, we showed that TrustID's ID derivation is secure against a strong attacker controlling the host.

Conclusion 8

Mobile devices, such as smartphones, show a combination of properties that makes securing them a particularly challenging task. Their mobility and connectivity exposes them to different, possibly hostile environments and, hence, to a variety of remote and local, physical attacks. Their versatility, achieved through a variety of sensors and the ability for user customization, ensures their wide adoption and usage in a multitude of use cases, which, in turn, results in a large amount of valuable data being processed on them.

Advanced attacks emerging in the recent years have shown that mobile devices are indeed becoming a prime target for attackers. Remote software attacks, such as Stagefright, threaten the run-time integrity of devices, and physical attacks, for example, via cold boot, threaten the confidentiality of main memory data. On the other hand, mobile devices and their CPUs get more powerful and gain more security features. This also includes new logical separations, i.e., isolated code execution contexts, such as TEEs or additional modes for hardware-assisted virtualization. While those separations allow for novel security schemes, in some cases, their isolation can be overcome by physical attacks or microarchitectural attacks raising the need for stronger protection mechanisms, for example, using physical separation, for securing high value secrets.

In this thesis, we explored ways to leverage both logical and physical separation for mobile security. More specifically, we explored novel approaches for monitoring and protecting the run-time integrity of mobile devices and for protecting the confidentiality of main memory data and of valuable secrets, such as keys and identities.

8.1 Contributions

As basis for our security concepts, we first introduced a *generic, modular system architecture* in Chapter 3. Our architecture consists of two physical entities, the target platform, a mobile device providing several logical separations, and the security token, a smaller device under physical protection by its user, as additional physical separation. Instead of defining a full software architecture and requiring specific hardware platforms for these two entities, we focused on specifying required basic hardware features for each of our security concepts. This makes our architecture modular and adaptable to different scenarios with different hardware devices and different combinations of our security concepts. Furthermore, we introduced a *generic attacker model*, specifying a remote and a physical attacker as basis for the specific threat models defined for each security concept in the thesis.

As foundation for our token-based security mechanisms protecting keys and identities, we want the security token to be able to protect itself against a target platform whose boot-time integrity has been compromised. To achieve this, in Chapter 4, we introduced a *software-based trusted boot process*, which is able to establish this initial trust in the target platform's integrity without relying on pre-provisioned key material or specific hardware features. As basis for this concept, acting as trust anchor for the software-based trusted boot, we introduced *SobTrA*, a timing-based primitive that allows the token to obtain a guarantee that a piece of code has been executed untampered on a target platform with an ARM Cortex-A8 CPU. With our prototype implementation and evaluation, we showed that our software-based trusted boot is able to reliably detect a tampered target platform.

In order to harden the target platform against remote attacks and, thus, enable it to maintain the initial trust of the token, in Chapter 5, we explored new ways to monitor and eventually protect the target platform's run-time integrity by leveraging the logical separation offered by hardware-assisted virtualization. We first introduced a *framework for gathering page-granular control flow data* of the target platform's kernel and user space from a minimal, custom ARM hypervisor. By restricting the executable pages of the guest and analyzing the resulting page faults in the hypervisor, our framework is able to gather control flow data transparently and without modifications to the software in higher layers. Additionally, we presented an example application using the framework to enforce a predetermined page-granular control flow in the guest to eventually harden the target platform against control flow hijacking attacks, such as ROP. We showed the feasibility of our concept with the implementation of a full prototype on an ARM Cortex-A15 development board running Android. The detailed performance analysis of our prototype showed that the mechanism performs comparably well and, furthermore, gives an impression how lightweight ARM's hardware virtualization extensions are in general, enabling novel security mechanisms relying on frequent context switches.

Mobile devices are threatened not only by remote, software-based attacks but also by physical attacks. Ensuring a target platform's software integrity typically does not protect it against physical attacks, such as cold boot. Therefore, in Chapter 6, we explored software-based mechanisms for memory encryption, protecting the target platform's data confidentiality against strong physical attackers. To this end, we proposed the combination of a run-time and a suspend-time encryption scheme. First, we proposed *TransCrypt*, a virtualization-based mechanism that is able to transparently encrypt kernel and user space at run-time. TransCrypt is implemented in the hypervisor and works by restricting the guest to a small working set of unencrypted pages while keeping most of the other pages encrypted. The mechanism works transparently, without exposing an interface to the guest, and independently of the guest's OS. With our fully functional prototype on an ARM Cortex-A15 development board and our detailed evaluation, we showed that TransCrypt is feasible and able to effectively and efficiently protect actual secrets, such as a user's E-mail password, in the target platform's main memory against strong physical attackers.

With TransCrypt, a working set of memory pages remains unencrypted, and the encryption key, since it is required constantly, is only logically separated inside the target platform using a TEE. Based on the observation that many functions of a typical mobile device are not required most of the time, we hence proposed a combination of TransCrypt with a complementary *suspend-time memory encryption scheme*, able to provide stronger protections for a suspended system or suspended groups of processes, representing a specific function of the device. The mechanism is implemented as an extension to the Linux process suspension system and forces processes to encrypt themselves during suspension. The respective keys for the encryption of a process group are only available during encryption and decryption and securely physically separated on our token otherwise. Therefore, data of a suspended group of processes is strongly protected, even against an attacker gaining software control of the target platform. We discussed in detail how to combine both run- and suspend-time encryption to achieve a target platform whose data confidentiality is optimally protected against physical attackers.

Both our proposed memory encryption schemes are software-based and target typical mobile devices, i.e., ARM platforms running, for example, Android. Recent server platforms feature hardware mechanisms for memory encryption. As last part of Chapter 6 and as comparison to our combined software-based encryption system, we presented an analysis of AMD's SEV feature for encrypting VMs, whose goal is not only to protect VMs against a physical attacker but also against a malicious hypervisor. With *SEVered*, our attack able to successfully extract memory from encrypted VMs, we showed that such an unconventional trust model is hard to realize securely.

Our previous, target platform-based security concepts mostly rely on logical, in-CPU separation. Logically separated contexts typically still share many resources, such as caches and physical memory. Microarchitectural attacks, for example, using cache timing, are able to exploit the sharing of physical resources to break logical separations. Therefore, while the security offered by logical separations is acceptable for the bulk of data, stronger protection is required and achievable for small, highly valuable secrets. In Chapter 7, we explored ways to leverage the physical separation offered by our token in order to protect valuable secrets, even in the presence of a fully compromised target platform. To this end, we first proposed *CoKey*, a concept using our token to protect keys for symmetric cryptography. Symmetric cryptography is often performance-critical and externalizing the entire cryptographic operation is too inefficient for data-intensive use cases, such as FDE. CoKey combines key material of the target platform and the token, binding encrypted data to the specific token used for its encryption, and forces both entities to cooperate during encryption and decryption. We showed the feasibility of our concept with the implementation of a fully functional prototype. In our detailed performance evaluation and security discussion, we compared CoKey to other key protection schemes and showed that it offers an efficient and effective trade-off against different remote and physical attackers.

As second part of Chapter 7, we introduced *TrustID*, a concept for securing user IDs, leveraging the physical separation of our token. In our concept, user IDs are represented by asymmetric key pairs whose private key is securely stored in our token. TrustID extends our architecture with a secure RootID device from which it is able to securely derive trusted IDs into the token with the target platform establishing the physical connection between both entities. With our *secure combined PIN entry*, the TrustID derivation protocol is secure even in the presence of a fully compromised target platform.

All in all, with our contributions we were able to improve the security of mobile devices with regard to several key aspects. We leveraged both logical and physical separation as features of modern mobile platforms to build novel concepts protecting the integrity and confidentiality of mobile devices against remote and physical attackers.

8.2 Future Research Directions

As the importance of mobile devices will continue to surge, attacks will get more and more advanced. This increases the need for advanced mobile security concepts with a variety of corresponding research challenges. In the following, we discuss some possible future research directions taking our contributions as a starting point.

Mobile and embedded devices heavily rely on unsafe, fast programming languages, such as C and C++. As discussed in Chapter 5, in order to protect mobile devices against remote attacks, it is crucial to ensure their software's run-time integrity against advanced attack techniques, such as ROP. Research from recent years has produced promising compiler-based techniques that use instrumentation to ensure run-time integrity for unsafe languages from within the program itself. CFI and similar compiler-based approaches typically rely on metadata for their functionality. While the integrity of this metadata is typically crucial for the security of an approach, it is often only protected by hiding it in the address space of the process. In order to improve this situation, the virtualization-based techniques presented in this thesis could be combined with compiler-based approaches to realize a run-time integrity approach not relying on information hiding.

CPUs gain more and more security functions. Intel Control-flow Enforcement Technology (CET) and ARM pointer authentication [Qua17] are recent features whose goal is the hardware-assisted protection of the program control flow. Intel Memory Protection Extensions (MPX) is a feature whose goal is not only to protect the control flow but to help implement full memory safety for unsafe languages. As a future research opportunity, one could analyze the security those features can achieve and explore ways to combine them with existing software-, virtualization- and TEE-based approaches.

Mobile devices are at high risk of getting lost or stolen and hence being subject to physical attacks. As discussed in Chapter 6, encryption of the main memory can prevent memory attacks, a large subclass of physical attacks whose goal is the extraction of data from main memory, for example, via DMA or cold boot. While our concepts already provide a relatively complete protection considering the fact that they are software-based, a future research opportunity could be to optimize their performance. A large performance gain could, for example, be achieved by using hardware-accelerated symmetric cryptography of the ARMv8-A architecture. Nonetheless, full hardware-based solutions for memory encryption, such as AMD SME and SEV and Intel's announced TME technology [Int17], will most certainly gain more relevance in the future and their security should be further analyzed in future research.

Most of the security concepts proposed in this thesis rely on the security of logical separation inside the target platform's CPU, for example, on a TEE being able to protect a key against a software attack. As discussed in Chapter 7, this might not always be the case. Even assuming bug-free system software, an attacker might still be able to extract secrets from logically separated contexts, such as higher privilege levels, using microarchitectural attacks. In order to protect against those attacks, they must be studied in detail in future research. Based on the results, software-based defenses should be researched to bridge the gap until CPU manufacturers deliver usable hardware-based solutions.

Finally, as another research opportunity, one could investigate further uses and concepts for physical separation. As discussed in Chapter 7, our physically separated security token can help to protect highly valuable secrets against all kinds of threats, including microarchitectural attacks. As a basic mechanism helping the token to protect itself from a compromised target platform, in Chapter 4, we proposed a software-based trusted boot process, which could be generalized to support more target platforms in future research. Furthermore, new token-based functions could be researched, providing secure physical separation for even more secrets of the target platform.

Acronyms

AEAD	Authenticated Encryption with Associated Data 104
AES	Advanced Encryption Standard 97, 101, 103, 104, 110, 114, 144, 145
AMD-SP	AMD Secure Processor 23, 24, 121, 126, 129
APDU	Application Protocol Data Unit 156, 161
API	Application Programming Interface 138–140, 144–146
ASID	Address Space Identifier 22, 23, 77
ASLR	Address Space Layout Randomization 3, 85
BTS	Branch Trace Store 79
CA	Certificate Authority 155
CBC	Cipher Block Chaining 104, 141
CET	Control-flow Enforcement Technology 168
CF	Control Flow 65, 67–70, 73–75, 77, 78, 80, 82, 84, 85, 87–89, 92
CFG	Control Flow Graph 68
CFI	Control Flow Integrity 70, 84, 168
CPSR	Current Program Status Register 15, 50, 54, 57–60, 77, 78
CPU	Central Processing Unit iii–v, 4–6, 8, 11–15, 17–19, 21–31, 33, 35, 42–44, 48–50, 52, 54, 55, 60, 61, 65, 66, 70, 71, 74, 79, 80, 89, 90, 92–96, 107, 108, 110, 114, 115, 120–122, 130–133, 136, 138, 164–169
CRA	Code-Reuse Attack 3, 67, 68, 84

CRC	Cyclic Redundancy Check 89
CRTM	Core Root of Trust for Measurement 32
CTR	Counter 141, 145
CVE	Common Vulnerabilities and Exposures 2
DEP	Data Execution Prevention 3, 65, 67, 84
DMA	Direct Memory Access iii, vi, 2–5, 7, 23, 27–29, 39, 43, 71, 93, 94, 96, 98, 100, 101, 103, 104, 107–109, 111, 115, 129, 132–134, 136, 169
DOP	Data-Oriented Programming 68
DRAM	Dynamic Random Access Memory 4, 23, 24, 132
DRM	Digital Rights Management 24
EAP	Extensible Authentication Protocol 152
ECB	Electronic Code Book 140, 142, 144, 145
ECC	Elliptic Curve Cryptography 155, 161
EIC	Electronic Identity Card 151, 154
EL	Exception Level 15–19, 22, 26, 27, 30, 31, 56, 57, 77
EPC	Enclave Page Cache 28, 29
EPCM	Enclave Page Cache Map 28, 29
ESSIV	Encrypted Salt-Sector Initialization Vector 104, 141
FDE	Full Disk Encryption iv, 5, 8, 35, 93, 96, 104, 115, 121, 133, 137, 146, 148, 149, 164, 167
FIQ	Fast Interrupt 60
fTPM	Firmware Trusted Platform Module 32
GPA	Guest-Physical Address 18, 22, 70, 72, 122, 123, 128, 129
GPS	Global Positioning System 2
GPU	Graphics Processing Unit 14, 90, 96, 98, 104, 107, 111, 114, 132, 134

GUI	Graphical User Interface 161
HCR	Hyp Configuration Register 109
HSM	Hardware Security Module 136
HSR	Hyp Syndrome Register 77
HTTP	Hypertext Transfer Protocol 156
HVC	Hypervisor Call 22, 72, 100
I/O	Input/Output 10, 35, 107, 114
ID	Identity iv, vi, 8, 9, 29, 115, 133, 134, 150–156, 158–164, 168
IOMMU	Input-Output Memory Management Unit 23, 71, 96
IoT	Internet of Things 1
IPA	Intermediate Physical Address 18, 22, 70, 72, 73, 77, 80, 81, 101, 102, 109, 110, 122
iRAM	Internal RAM 97
IRQ	Interrupt Request 60
ISA	Instruction Set Architecture 15, 21
IV	Initialization Vector 103, 104, 110, 118, 128, 135, 139–142, 144–150
JIT	Just-In-Time 67, 75, 85, 88
JTAG	Joint Test Action Group 43, 67, 72, 112
JVM	Java Virtual Machine 20
KRACK	Key Reinstallation Attack 2
KVM	Kernel-based Virtual Machine 126
LBR	Last Branch Record 79
LED	Light-Emitting Diode 37, 139, 145, 149, 150
LOC	Lines of Code 80, 84, 109, 143
LPAE	Large Physical Address Extension 81

LR	Link Register 15, 57, 59, 77
LRM	Least Recently Mapped 102, 110
LRU	Least Recently Used 102
MEE	Memory Encryption Engine 28
MitM	Man-in-the-Middle 2, 44
MMU	Memory Management Unit 13, 17, 22, 50, 60, 79
MNO	Mobile Network Operator 152, 153
MPS	Mapped Page Set 101–106, 110–112, 115
MPU	Memory Protection Unit 13
MPX	Memory Protection Extensions 168
NFC	Near Field Communication 151, 153, 154, 161
NIC	Network Interface Controller 96, 132, 134
nPA	Neuer Personalausweis 154, 158, 161, 162, 164
NS	Non-Secure 19, 26, 27
OCRAM	On-Chip RAM 97, 103, 115
OS	Operating System 2, 5, 7, 10, 16, 17, 19–22, 26, 27, 31, 57–59, 65–67, 79, 86, 89, 94, 95, 97–99, 102, 104, 111, 138, 167
OTP	One-Time Password 30, 137
PA	Physical Address 17, 18, 22, 70, 72, 73, 77, 80, 81, 101, 102, 109, 123
PACE	Password Authenticated Connection Establishment 158, 161
PAKE	Password-Authenticated Key Agreement 158, 160, 163
PC	Program Counter 15, 49–53, 55, 57, 77
PIC	Position-Independent Code 3, 84
PID	Process Identifier 20, 77

PIN	Personal Identification Number 36, 37, 41, 118, 139, 151, 153, 155, 156, 158, 160, 163, 164, 168
PKI	Public Key Infrastructure 150, 153, 155, 156, 159
PL	Privilege Level 15, 17, 56–59, 77, 80–82
PoC	Proof of Concept 110, 111, 122, 126–128
PRM	Processor Reserved Memory 28
PTE	Page Table Entry 19, 124
PXN	Privileged Execute Never 19
RAM	Random Access Memory 5, 25, 27, 29, 81, 90, 93–99, 103, 104, 107, 109, 114, 116, 119–121, 133, 134, 136, 137, 146
RCE	Remote Code Execution 2, 3, 38, 96, 132, 134
REE	Rich Execution Environment 24, 25
RISC	Reduced Instruction Set Computing 13
ROM	Read-Only Memory 27, 30, 31, 41
ROP	Return-Oriented Programming 3, 10, 68, 70, 79, 88, 166, 168
SCC	Security Controller 138, 143, 145, 146
SCR	Secure Configuration Register 58
SCTLR	System Control Register 15
SDIO	Secure Digital Input Output 36, 45, 154
SE	Secure Element 24, 39, 116, 151–154, 161, 164
SEV	Secure Encrypted Virtualization iv, vi, 12, 23, 24, 95, 97, 121–123, 126, 128, 130, 167, 169
SEV-ES	SEV Encrypted State 23, 24
SGX	Software Guard Extensions 4, 26–29, 34, 96, 136
SIM	Subscriber Identification Module 153
SIMD	Single Instruction Multiple Data 132

SLAT	Second Level Address Translation 18, 22, 23, 35, 66, 69–77, 80, 92, 99–102, 105, 106, 109, 114, 123, 124, 126, 129
SMC	Secure Memory Card 26, 31
SME	Secure Memory Encryption 23, 24, 97, 121, 169
SMM	Secure Management Mode 24
SMMU	System Memory Management Unit 71, 96
SMP	Symmetric Multiprocessing 80
SobTrA	Software-based Trust Anchor 11, 34, 36, 42–47, 49, 51–64, 166
SoC	System on a Chip 1, 14, 24, 25, 27, 45, 60, 89, 97, 132, 136, 138, 143, 145, 146
SP	Stack Pointer 15, 53, 54, 77, 78
SPI	Serial Peripheral Interface 36, 45, 60–62
SPS	Special Page Set 101–105, 107, 112
SPSR	Saved Program Status Register 57, 77
SR	Status Register 49–52
SRAM	Static Random-access Memory 25, 27, 61, 97, 136
SSD	Solid-State Drive 146
SSO	Single Sign On 152
TCB	Trusted Computing Base 6, 7, 35, 39, 40, 65, 66, 70, 72, 80, 92, 93, 99, 130, 153
TEE	Trusted Execution Environment iii–vi, 4, 6, 11, 16, 24–27, 34, 35, 41, 43, 64, 95, 97, 101, 120, 131, 132, 134, 136, 165, 167–169
TIP	Trusted Identity Provider 134, 151, 153–156, 158, 159, 161–163
TLB	Translation Lookaside Buffer 22, 27, 55, 76, 110, 114
TLS	Transport Layer Security 152–154, 156, 159, 161, 163
TME	Total Memory Encryption 97, 121, 169
TPM	Trusted Platform Module 32, 41, 43, 137

TSME	Transparent SME 23
TZASC	TrustZone Address Space Controller 27
TZMA	TrustZone Memory Adapter 27
UEFI	Unified Extensible Firmware Interface 31
UICC	Universal Integrated Circuit Card 152, 153
UPS	Unencrypted Page Set 99, 101, 104, 105, 112
USB	Universal Serial Bus 36, 45, 136, 137, 143–148
UXN	Unprivileged Execute Never 19
VA	Virtual Address 17, 18, 22, 28, 70, 75, 77, 78, 80, 81, 101, 102, 109, 111, 112, 122
VM	Virtual Machine 18, 20–24, 80, 95, 97, 109, 116, 121–124, 126–130, 150, 167
VMCB	Virtual Machine Control Block 24
VMID	Virtual Machine Identifier 22
VMM	Virtual Machine Monitor 16, 20, 65, 66, 68–70, 92
VPN	Virtual Private Network 136
W^X	Write XOR Execute 3, 75, 84, 85
WLAN	Wireless Local Area Network 162
WPA2	Wi-Fi Protected Access 2 2
XN	Execute Never 19, 70, 72, 81, 82

Bibliography

[Aba+05] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. "Control-flow Integrity". In: *Proceedings of the 12th ACM Conference on Computer and Communications Security*. CCS '05. Alexandria, VA, USA: ACM, 2005, pp. 340–353. DOI: 10.1145/1102120.1102165.

[Adv18] Advanced Micro Devices. *Secure Encrypted Virtualization API Version 0.16. Technical Preview*. Publication Nr. 55766, Revision: 3.06. Feb. 2018.

[AKm15] Gunnar Alendal, Christian Kison, and modg. *got HW crypto? On the (in)security of a Self-Encrypting Drive series*. IACR Cryptology ePrint Archive, Report 2015/1002. 2015. URL: http://eprint.iacr.org/2015/1002.

[Ali11] Asad M. Ali. "Seamless Fusion of Secure Software and Trusted USB Token for Protecting Enterprise & Government Data". In: *Proceedings of the Sixth International Conference on Availability, Reliability and Security*. ARES '11. Vienna, Austria: IEEE, Aug. 2011, pp. 409–414. DOI: 10.1109/ARES.2011.67.

[AF04] Tiago Alves and Don Felton. *TrustZone: Integrated Hardware and Software Security – Enabling Trusted Computing in Embedded Systems*. ARM Limited, 2004.

[AnT] AnTuTu Developers. *AnTuTu*. URL: http://www.antutu.com/en/index.shtml (Accessed: 14 Mar. 2019).

[AFS97] William A. Arbaugh, David J. Farber, and Jonathan M. Smith. "A Secure and Reliable Bootstrap Architecture". In: *Proceedings of the 1997 IEEE Symposium on Security and Privacy*. S&P '97. Oakland, CA, USA: IEEE, 1997, pp. 65–71. DOI: 10.1109/SECPRI.1997.601317.

[ARM10] ARM Limited. *Cortex-A8 – Technical Reference Manual*. Version r3p2. ARM DDI 0344K (ID060510). May 2010.

[ARM14] ARM Limited. *ARM Architecture Reference Manual – ARMv7-A and ARMv7-R edition*. ARM DDI 0406C.c (ID051414). May 2014.

[ARM17] ARM Limited. *ARM Architecture Reference Manual – ARMv8-A, for ARMv8-A architecture profile*. ARM DDI 0487C.a (ID121917). Dec. 2017.

[ARM18] ARM Limited. *Cache Speculation Side-channels*. White Paper. Version 2.4. Oct. 2018. URL: `https://developer.arm.com/support/arm-security-updates/speculative-processor-vulnerability/download-the-whitepaper` (Accessed: 14 Mar. 2019).

[Art17] Nitay Artenstein. *Broadpwn: Remotely Compromising Android and iOS via a Bug in Broadcom's Wi-Fi Chipsets*. July 2017. URL: `https://blog.exodusintel.com/2017/07/26/broadpwn/` (Accessed: 14 Mar. 2019).

[Avi+10] Adam J. Aviv, Katherine L. Gibson, Evan Mossop, Matt Blaze, and Jonathan M. Smith. "Smudge Attacks on Smartphone Touch Screens". In: *Proceedings of the 4th USENIX Workshop on Offensive Technologies*. WOOT '10. Washington, DC, USA: USENIX Association, Aug. 2010.

[ADB15] Zuk Avraham, Joshua Drake, and Nikias Bassen. *Experts Found a Unicorn in the Heart of Android*. Zimperium. July 2015. URL: `https://blog.zimperium.com/experts-found-a-unicorn-in-the-heart-of-android/` (Accessed: 14 Mar. 2019).

[Aza+14] Ahmed M. Azab, Peng Ning, Jitesh Shah, Quan Chen, Rohan Bhutkar, Guruprasad Ganesh, Jia Ma, and Wenbo Shen. "Hypervision Across Worlds: Real-time Kernel Protection from the ARM TrustZone Secure World". In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. CCS '14. Scottsdale, Arizona, USA: ACM, Nov. 2014, pp. 90–102. DOI: `10.1145/2660267.2660350`.

[Baz+16] Max Bazaliy, Seth Hardy, Michael Flossman, Kristy Edwards, Andrew Blaich, and Mike Murray. *Technical Analysis of Pegasus Spyware*. White Paper. Lookout, Aug. 2016. URL: `https://info.lookout.com/rs/051-ESQ-475/images/lookout-pegasus-technical-analysis.pdf` (Accessed: 18 Mar. 2019).

[Bea09] Beagleboard.org. *BeagleBoard System Reference Manual*. Revision C4. Dec. 2009. URL: `http://beagleboard.org/static/BBSRM_latest.pdf` (Accessed: 14 Mar. 2019).

[BDK05] Michael Becher, Maximillian Dornseif, and Christian N. Klein. "FireWire: All Your Memory Are Belong To Us". In: *Proceedings of CanSecWest*. 2005.

[BeI] BeID - Berlin electronic IDentity laboratory. *eIDClientCore*. URL: `http://sar.informatik.hu-berlin.de/BeID-lab/eIDClientCore/` (Accessed: 18 Mar. 2019).

[BM92] Steven M. Bellovin and Michael Merritt. “Encrypted Key Exchange: Password-based Protocols Secure Against Dictionary Attacks”. In: *Proceedings of the 1992 IEEE Computer Society Symposium on Research in Security and Privacy*. Oakland, CA, USA: IEEE, May 1992, pp. 72–84. DOI: `10.1109/RISP.1992.213269`.

[Ben17] Gal Beniamini. *Trust Issues: Exploiting TrustZone TEEs*. Google Project Zero. July 24, 2017. URL: `https://googleprojectzero.blogspot.com/2017/07/trust-issues-exploiting-trustzone-tees.html` (Accessed: 18 Mar. 2019).

[Ber05] Daniel J. Bernstein. *Cache-timing attacks on AES*. Tech. rep. The University of Illinois at Chicago, 2005.

[Bit17] Bitkom e.V. *Mobile Steuerungszentrale für das Internet of Things*. Feb. 22, 2017. URL: `https://www.bitkom.org/Presse/Presseinformation/Mobile-Steuerungszentrale-fuer-das-Internet-of-Things.html` (Accessed: 18 Mar. 2019).

[BR12] Erik-Oliver Blass and William Robertson. “TRESOR-HUNT: Attacking CPU-bound Encryption”. In: *Proceedings of the 28th Annual Computer Security Applications Conference*. ACSAC ’12. Orlando, Florida, USA: ACM, Dec. 2012, pp. 71–78. DOI: `10.1145/2420950.2420961`.

[Blu18] Bluetooth SIG, Inc. *Bluetooth market update 2018*. 2018.

[Boi06] Adam Boileau. *Hit by a Bus: Physical Access Attacks with Firewire*. Presentation, Ruxcon. 2006.

[Bos+08] Abhijit Bose, Xin Hu, Kang G. Shin, and Taejoon Park. “Behavioral Detection of Malware on Mobile Handsets”. In: *Proceedings of the 6th International Conference on Mobile Systems, Applications, and Services*. MobiSys ’08. Breckenridge, CO, USA: ACM, June 2008, pp. 225–238. DOI: `10.1145/1378600.1378626`.

[BMP00] Victor Boyko, Philip D. MacKenzie, and Sarvar Patel. “Provably Secure Password-Authenticated Key Exchange Using Diffie-Hellman”. In: *Advances in Cryptology - EUROCRYPT 2000, International Conference on the Theory and Application of Cryptographic Techniques, Bruges, Belgium, May 14-18, 2000, Proceeding*. Vol. 1807. Lecture Notes in Computer Science. Springer, 2000, pp. 156–171. DOI: `10.1007/3-540-45539-6_12`.

[Bro] David Brownell. *USB Gadget API for Linux*. URL: `https://www.kernel.org/doc/htmldocs/gadget/` (Accessed: 14 Mar. 2019).

[Bun14] Bundesregierung der Bundesrepublik Deutschland. *Pläne zur Einführung eines zentralen Handy-Registers*. Drucksache 18/2236. Deutscher Bundestag, 18. Wahlperiode, July 30, 2014. URL: `http://dip21.bundestag.de/dip21/btd/18/022/1802236.pdf`.

[Car17] Pierre Carru. *Attack TrustZone with Rowhammer*. Presentation. Apr. 2017.

[Cas+09] Claude Castelluccia, Aurélien Francillon, Daniele Perito, and Claudio Soriente. "On the Difficulty of Software-based Attestation of Embedded Devices". In: *Proceedings of the 16th ACM Conference on Computer and Communications Security*. CCS '09. Chicago, Illinois, USA: ACM, Nov. 2009, pp. 400–409. DOI: `10.1145/1653662.1653711`.

[Che+18] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H. Lai. *SgxPectre Attacks: Leaking Enclave Secrets via Speculative Execution*. Feb. 2018. arXiv: `1802.09085 [cs.CR]`. URL: `https://arxiv.org/abs/1802.09085`.

[Che+09] Ping Chen, Hai Xiao, Xiaobin Shen, Xinchun Yin, Bing Mao, and Li Xie. "DROP: Detecting Return-Oriented Programming Malicious Code". In: *Information Systems Security, 5th International Conference, ICISS 2009, Kolkata, India, December 14-18, 2009, Proceedings*. Vol. 5905. Lecture Notes in Computer Science. Springer, 2009, pp. 163–177. DOI: `10.1007/978-3-642-10772-6_13`.

[Che+11] Wei-Dar Chen, Keith E. Mayes, Yuan-Hung Lien, and Jung-Hui Chiu. "NFC Mobile Payment with Citizen Digital Certificate". In: *Proceedings of the 2nd International Conference on Next Generation Information Technology*. ICNIT '11. Gyeongju, South Korea: IEEE, June 2011, pp. 120–126.

[CDC08] Xi Chen, Robert P. Dick, and Alok Choudhary. "Operating System Controlled Processor-memory Bus Encryption". In: *Proceedings of the Conference on Design, Automation and Test in Europe*. DATE '08. Munich, Germany: ACM, Mar. 2008, pp. 1154–1159. DOI: `10.1145/1403375.1403657`.

[Che+08] Xiaoxin Chen, Tal Garfinkel, E. Christopher Lewis, Pratap Subrahmanyam, Carl A. Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan R.K. Ports. "Overshadow: A Virtualization-based Approach to Retrofitting Protection in Commodity Operating Systems". In: *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XIII. Seattle, WA, USA: ACM, 2008, pp. 2–13. DOI: `10.1145/1346281.1346284`.

[Cis19] Cisco. *Cisco Visual Networking Index: Forecast and Trends, 2017–2022*. White Paper. Document ID:1551296909190103. Feb. 2019. URL: `https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/white-paper-c11-741490.html` (Accessed: 18 Mar. 2019).

[Col+15] Patrick Colp, Jiawen Zhang, James Gleeson, Sahil Suneja, Eyal de Lara, Himanshu Raj, Stefan Saroiu, and Alec Wolman. “Protecting Data on Smartphones and Tablets from Memory Attacks”. In: *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS ’15. Istanbul, Turkey: ACM, Mar. 2015, pp. 177–189. DOI: `10.1145/2694344.2694380`.

[Con] Embedded Microprocessor Benchmark Consortium. *CoreMark*. URL: `https://www.eembc.org/coremark/` (Accessed: 18 Mar. 2019).

[CN05] Mark D. Corner and Brian D. Noble. “Protecting File Systems with Transient Authentication”. In: *Wireless Networks* 11.1 (2005), pp. 7–19. DOI: `10.1007/s11276-004-4743-z`.

[CD16] Victor Costan and Srinivas Devadas. *Intel SGX Explained*. IACR Cryptology ePrint Archive, Report 2016/086. 2016. URL: `https://eprint.iacr.org/2016/086`.

[CVE] CVE Details. *Top 50 Products By Total Number Of Distinct Vulnerabilities in 2017*. URL: `https://www.cvedetails.com/top-50-products.php?year=2017` (Accessed: 18 Mar. 2019).

[Dal+16] Christoffer Dall, Shih-Wei Li, Jin Tack Lim, Jason Nieh, and Georgios Koloventzos. “ARM Virtualization: Performance and Architectural Implications”. In: *Proceedings of the 43rd International Symposium on Computer Architecture*. ISCA ’16. Seoul, Republic of Korea: IEEE, June 2016, pp. 304–316. DOI: `10.1109/ISCA.2016.35`.

[DN14] Christoffer Dall and Jason Nieh. “KVM/ARM: The Design and Implementation of the Linux ARM Hypervisor”. In: *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS ’14. Salt Lake City, Utah, USA: ACM, Feb. 2014, pp. 333–348. DOI: `10.1145/2541940.2541946`.

[Dav+10a] Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, and Marcel Winandy. “Privilege Escalation Attacks on Android”. In: *Information Security - 13th International Conference, ISC 2010, Boca Raton, FL, USA, October 25-28, 2010, Revised Selected Papers*. Vol. 6531. Lecture Notes in Computer Science. Springer, 2010, pp. 346–360. DOI: `10.1007/978-3-642-18178-8_30`.

[Dav+10b] Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, and Marcel Winandy. *Return-oriented Programming Without Returns on ARM*. Tech. rep. Ruhr-Universität Bochum, 2010.

[DSW11] Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. “ROPdefender: A Detection Tool to Defend Against Return-oriented Programming Attacks”. In: *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*. ASIACCS ’11. Hong Kong, China: ACM, Mar. 2011, pp. 40–51. DOI: 10.1145/1966913.1966920.

[Den+12] Zhui Deng, Dongyan Xu, Xiangyu Zhang, and Xuxiang Jiang. “Introlib: Efficient and Transparent Library Call Introspection for Malware Forensics”. In: *Digital Investigation* 9 (2012), pp. 13–23.

[DZX13] Zhui Deng, Xiangyu Zhang, and Dongyan Xu. “SPIDER: Stealthy Binary Program Instrumentation and Debugging via Hardware Virtualization”. In: *Proceedings of the 29th Annual Computer Security Applications Conference*. ACSAC ’13. New Orleans, Louisiana, USA: ACM, Dec. 2013, pp. 289–298. DOI: 10.1145/2523649.2523675.

[Din+08] Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. “Ether: Malware Analysis via Hardware Virtualization Extensions”. In: *Proceedings of the 15th ACM Conference on Computer and Communications Security*. CCS ’08. Alexandria, Virginia, USA: ACM, Oct. 2008, pp. 51–62. DOI: 10.1145/1455770.1455779.

[Din+12] Jiun-Hung Ding, Chang-Jung Lin, Ping-Hao Chang, Chieh-Hao Tsang, Wei-Chung Hsu, and Yeh-Ching Chung. “ARMvisor: System Virtualization for ARM”. In: *Proceedings of the Ottawa Linux Symposium (OLS)*. 2012, pp. 93–107.

[Dmi+12] Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Sandeep Tamrakar, and Christian Wachsmann. “SmartTokens: Delegable Access Control with NFC-Enabled Smartphones”. In: *Trust and Trustworthy Computing - 5th International Conference, TRUST 2012, Vienna, Austria, June 13-15, 2012. Proceedings*. Vol. 7344. Lecture Notes in Computer Science. Springer, 2012, pp. 219–238. DOI: 10.1007/978-3-642-30921-2_13.

[DK06] Guillaume Duc and Ronan Keryell. “CryptoPage: An Efficient Secure Architecture with Memory Encryption, Integrity and Information Leakage Protection”. In: *Proceedings of the 22th Annual Computer Security Applications Conference*. ACSAC ’06. Dec. 2006, pp. 483–492. DOI: 10.1109/ACSAC.2006.21.

[Fed14] Federal Communications Commission (FCC). *Report of Technological Advisory Council (TAC) Subcommittee on Mobile Device Theft Prevention (MDTP)*. Version 1.0. Dec. 2014.

[Fed12] Federal Office for Information Security (BSI). *BSI TR-03110, Advanced Security Mechanisms for Machine Readable Travel Documents*. Mar. 2012. URL: `https://www.bsi.bund.de/EN/Publications/TechnicalGuidelines/TR03110/BSITR03110.html` (Accessed: 18 Mar. 2019).

[Fre12] Freescale Semiconductor Inc. *i.MX53 Multimedia Applications Processor Reference Manual*. Rev. 2.1. June 2012. URL: `https://cache.freescale.com/files/32bit/doc/ref_manual/iMX53RM.pdf` (Accessed: 14 Mar. 2019).

[Fri+18] Pietro Frigo, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. "Grand Pwning Unit: Accelerating Microarchitectural Attacks with the GPU". In: *Proceedings of the 2018 IEEE Symposium on Security and Privacy*. S&P '18. San Jose, California, USA, May 2018, pp. 357–372. DOI: `10.1109/SP.2018.00022`.

[Fru05] Clemens Fruhwirth. *New Methods in Hard Disk Encryption*. Tech. rep. Vienna University of Technology, July 2005.

[GGR09] Ryan W. Gardner, Sujata Garera, and Aviel D. Rubin. "Detecting Code Alteration by Creating a Temporary Memory Bottleneck". In: *IEEE Trans. Information Forensics and Security* 4.4 (Dec. 2009), pp. 638–650. DOI: `10.1109/TIFS.2009.2033231`.

[Gar+07] Tal Garfinkel, Keith Adams, Andrew Warfield, and Jason Franklin. "Compatibility Is Not Transparency: VMM Detection Myths and Realities". In: *Proceedings of HotOS'07: 11th Workshop on Hot Topics in Operating Systems*. HotOS '07. San Diego, California, USA: USENIX Association, May 2007.

[GR03] Tal Garfinkel and Mendel Rosenblum. "A Virtual Machine Introspection Based Architecture for Intrusion Detection". In: *Proceedings of the 7th Annual Network and Distributed System Security Symposium*. NDSS '03. San Diego, California, USA: The Internet Society, 2003.

[GVJ14] Xinyang Ge, Hayawardh Vijayakumar, and Trent Jaeger. *Sprobes: Enforcing Kernel Code Integrity on the TrustZone Architecture*. Oct. 2014. arXiv: `1410.7747 [cs.CR]`. URL: `http://arxiv.org/abs/1410.7747`.

[GCK05] Jonathon T. Giffin, Mihai Christodorescu, and Louis Kruger. "Strengthening Software Self-Checksumming via Self-Modifying Code". In: *Proceedings of the 21st Annual Computer Security Applications Conference*. ACSAC '05. Tucson, AZ, USA: IEEE, Dec. 2005, pp. 23–32. DOI: `10.1109/CSAC.2005.53`.

[Gif82] David K. Gifford. "Cryptographic Sealing for Information Secrecy and Authentication". In: *Commun. ACM* 25.4 (Apr. 1982), pp. 274–286. DOI: `10.1145/358468.358493`.

[GLQ99] Tanguy Gilmont, Jean-Didier Legat, and Jean-Jacques Quisquater. “Enhancing Security in the Memory Management Unit”. In: *Proceedings of the 25th EUROMICRO Conference*. Milan, Italy: IEEE, Sept. 1999, pp. 449–456. DOI: `10.1109/EURMIC.1999.794507`.

[Glo17] GlobalPlatform Inc. *TEE System Architecture*. Version 1.1. Jan. 2017.

[Gol18] Nico Golde. *There’s Life in the Old Dog Yet: Tearing New Holes into Intel/iPhone Cellular Modems*. Comsecuris UG. Apr. 4, 2018. URL: `https://comsecuris.com/blog/posts/theres_life_in_the_old_dog_yet_tearing_new_holes_into_inteliphone_cellular_modems` (Accessed: 18 Mar. 2019).

[GL01] Oded Goldreich and Yehuda Lindell. “Session-Key Generation Using Human Passwords Only”. In: *Advances in Cryptology - CRYPTO 2001, 21st Annual International Cryptology Conference, Santa Barbara, California, USA, August 19-23, 2001, Proceedings*. Vol. 2139. Lecture Notes in Computer Science. Springer, 2001, pp. 408–432. DOI: `10.1007/3-540-44647-8_24`.

[Göt+16a] Johannes Götzfried, Nico Dörr, Ralph Palutke, and Tilo Müller. “HyperCrypt: Hypervisor-Based Encryption of Kernel and User Space”. In: *Proceedings of the 11th International Conference on Availability, Reliability and Security*. ARES ’16. Aug. 2016, pp. 79–87. DOI: `10.1109/ARES.2016.13`.

[GM13] Johannes Götzfried and Tilo Müller. “ARMORED: CPU-Bound Encryption for Android-Driven ARM Devices”. In: *Proceedings of the Eigth International Conference on Availability, Reliability and Security*. ARES ’13. Regensburg, Germany: IEEE, Sept. 2013, pp. 161–168. DOI: `10.1109/ARES.2013.23`.

[Göt+16b] Johannes Götzfried, Tilo Müller, Gabor Drescher, Stefan Nürnberger, and Michael Backes. “RamCrypt: Kernel-based Address Space Encryption for User-mode Processes”. In: *Proceedings of the 11th ACM Asia Conference on Computer and Communications Security*. ASIACCS ’16. Xi’an, China: ACM, May 2016, pp. 919–924. DOI: `10.1145/2897845.2897924`.

[Gut99] Peter Gutmann. “The Design of a Cryptographic Security Architecture”. In: *Proceedings of the 8th USENIX Security Symposium*. Washington, DC, USA: USENIX Association, Aug. 1999.

[Gut00] Peter Gutmann. “An Open-Source Cryptographic Coprocessor”. In: *Proceedings of the 9th USENIX Security Symposium*. SEC ’00. Denver, Colorado, USA: USENIX Association, Aug. 2000.

[Gut01] Peter Gutmann. “Data Remanence in Semiconductor Devices”. In: *Proceedings of the 10th USENIX Security Symposium*. SEC ’01. Washington, DC, USA: USENIX Association, Aug. 2001.

[Hal+09] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten. "Lest We Remember: Cold-boot Attacks on Encryption Keys". In: *Commun. ACM* 52.5 (May 2009), pp. 91–98. DOI: 10.1145/1506409.1506429.

[HWF15] Daniel M. Hein, Johannes Winter, and Andreas Fitzek. "Secure Block Device – Secure, Flexible, and Efficient Data Storage for ARM TrustZone Systems". In: *Proceedings of the 14th International Conference on Trust, Security and Privacy in Computing and Communications*. TrustCom '15. Helsinki, Finland: IEEE, Aug. 2015, pp. 222–229. DOI: 10.1109/Trustcom.2015.378.

[Hei09] G. Heiser. "Hypervisors for Consumer Electronics". In: *Proceedings of the 6th IEEE Consumer Communications and Networking Conference*. CCNC '09. Las Vegas, NV, USA: IEEE, Jan. 2009, pp. 1–5. DOI: 10.1109/CCNC.2009.4784922.

[HT13] Michael Henson and Stephen Taylor. "Beyond Full Disk Encryption: Protection on Security-Enhanced Commodity Processors". In: *Applied Cryptography and Network Security - 11th International Conference, ACNS 2013, Banff, AB, Canada, June 25-28, 2013. Proceedings*. Vol. 7954. Lecture Notes in Computer Science. Springer, 2013, pp. 307–321. DOI: 10.1007/978-3-642-38980-1_19.

[Hig] High Density Devices AS. *Hiddn coCrypt*. URL: https://hiddn.no/products/encrypted-usb-flash-stick (Accessed: 14 Mar. 2019).

[Hit08] Hitachi Global Storage Technologies. *Safeguarding Your Data with Hitachi Bulk Data Encryption*. White Paper. July 2008.

[HFS98] Steven A. Hofmeyr, Stephanie Forrest, and Anil Somayaji. "Intrusion Detection Using Sequences of System Calls". In: *Journal of Computer Security* 6.3 (July 1998), pp. 151–180.

[Hor18] Jann Horn. *Reading privileged memory with a side-channel*. Google Project Zero. Jan. 3, 2018. URL: https://googleprojectzero.blogspot.de/2018/01/reading-privileged-memory-with-side.html (Accessed: 18 Mar. 2019).

[Hor12] Julian Horsch. "Software-based Trust Anchors for ARM Cortex Application Processors". Master's Thesis. Techniche Universität München, May 2012.

[Hor+14a] 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. San Antonio, Texas, USA: ACM, Mar. 2014, pp. 281–288. DOI: 10.1145/2557547.2557593.

[HHW17a] Julian Horsch, Manuel Huber, and Sascha Wessel. “TransCrypt: Transparent Main Memory Encryption Using a Minimal ARM Hypervisor”. In: *Proceedings of the 16th International Conference on Trust, Security and Privacy in Computing and Communications*. TrustCom ’17. Sydney, Australia: IEEE, Aug. 2017, pp. 152–161. DOI: `10.1109/Trustcom/BigDataSE/ICESS.2017.232`.

[HW15] Julian Horsch and Sascha Wessel. “Transparent Page-Based Kernel and User Space Execution Tracing from a Custom Minimal ARM Hypervisor”. In: *Proceedings of the 14th International Conference on Trust, Security and Privacy in Computing and Communications*. TrustCom ’15. Helsinki, Finland: IEEE, Aug. 2015, pp. 408–417. DOI: `10.1109/Trustcom.2015.401`.

[HWE16] Julian Horsch, Sascha Wessel, and Claudia Eckert. “CoKey: Fast Token-based Cooperative Cryptography”. In: *Proceedings of the 32nd Annual Conference on Computer Security Applications*. ACSAC ’16. Los Angeles, California, USA: ACM, Dec. 2016, pp. 314–323. DOI: `10.1145/2991079.2991117`.

[Hor+14b] Julian Horsch, Sascha Wessel, Frederic Stumpf, and Claudia Eckert. “SobTrA: A Software-based Trust Anchor for ARM Cortex Application Processors”. In: *Proceedings of the 4th ACM Conference on Data and Application Security and Privacy*. CODASPY ’14. San Antonio, Texas, USA: ACM, Mar. 2014, pp. 273–280. DOI: `10.1145/2557547.2557569`.

[Hu+16] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. “Data-Oriented Programming: On the Expressiveness of Non-control Data Attacks”. In: *Proceedings of the 2016 IEEE Symposium on Security and Privacy*. S&P ’16. San Jose, California, USA: IEEE, May 2016, pp. 969–986. DOI: `10.1109/SP.2016.62`.

[HS12] Jingyu Hua and Kouichi Sakurai. “Barrier: A Lightweight Hypervisor for Protecting Kernel Integrity via Memory Isolation”. In: *Proceedings of the 27th Annual ACM Symposium on Applied Computing*. SAC ’12. Trento, Italy: ACM, Mar. 2012, pp. 1470–1477. DOI: `10.1145/2245276.2232011`.

[Hub+17] Manuel Huber, Julian Horsch, Junaid Ali, and Sascha Wessel. “Freeze & Crypt: Linux Kernel Support for Main Memory Encryption”. In: *14th International Conference on Security and Cryptography (SECRYPT 2017)*. INSTICC. ScitePress, 2017.

[Hub+18] Manuel Huber, Julian Horsch, Junaid Ali, and Sascha Wessel. “Freeze and Crypt: Linux Kernel Support for Main Memory Encryption”. In: *Computers & Security* (2018). DOI: `10.1016/j.cose.2018.08.011`.

[Hub+15] Manuel Huber, Julian Horsch, Michael Velten, Michael Weiß, and Sascha Wessel. “A Secure Architecture for Operating System-Level Virtualization on Mobile Devices”. In: *Information Security and Cryptology - 11th International Conference, Inscrypt 2015, Beijing, China, November 1-3, 2015, Revised Selected Papers*. Vol. 9589. Lecture Notes in Computer Science. Springer, 2015, pp. 430–450. DOI: 10.1007/978-3-319-38898-4_25.

[HHW17b] Manuel Huber, Julian Horsch, and Sascha Wessel. “Protecting Suspended Devices from Memory Attacks”. In: *Proceedings of the 10th European Workshop on Systems Security*. EuroSec’17. Belgrade, Serbia: ACM, Apr. 2017, 10:1–10:6. DOI: 10.1145/3065913.3065914.

[Hyp08] Konstantin Hyppönen. “An Open Mobile Identity Tool: An Architecture for Mobile Identity Management”. In: *Public Key Infrastructure, 5th European PKI Workshop: Theory and Practice, EuroPKI 2008, Trondheim, Norway, June 16-17, 2008, Proceedings*. Vol. 5057. Lecture Notes in Computer Science. Springer, 2008, pp. 207–222. DOI: 10.1007/978-3-540-69485-4_15.

[Int17] Intel Corp. *Intel Architecture – Memory Encryption Technologies Specification*. Rev: 1.1. Dec. 2017.

[Int18] Intel Corp. *Intel 64 and IA-32 Architectures Software Developer’s Manual. Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D and 4*. Order Number: 325462-067US. May 2018.

[Inv] Inverse Path S.r.l. *USB armory – Open Source Flash-Drive Sized Computer*. URL: https://inversepath.com/usbarmory (Accessed: 14 Mar. 2019).

[JJ11] Markus Jakobsson and Karl-Anders Johansson. “Practical and Secure Software-Based Attestation”. In: *Proceedings of the 2011 Workshop on Lightweight Security Privacy: Devices, Protocols and Applications*. LightSec ’11. Mar. 2011, pp. 1–9. DOI: 10.1109/LightSec.2011.8.

[Jia+11] Jun Jiang, Xiaoqi Jia, Dengguo Feng, Shengzhi Zhang, and Peng Liu. “HyperCrop: A Hypervisor-Based Countermeasure for Return Oriented Programming”. In: *Information and Communications Security - 13th International Conference, ICICS 2011, Beijing, China, November 23-26, 2011. Proceedings*. Vol. 7043. Lecture Notes in Computer Science. Springer, 2011, pp. 360–373. DOI: 10.1007/978-3-642-25243-3_29.

[JWX07] Xuxian Jiang, Xinyuan Wang, and Dongyan Xu. “Stealthy Malware Detection Through Vmm-based "Out-of-the-box" Semantic View Reconstruction”. In: *Proceedings of the 14th ACM Conference on Computer and Communications Security*. CCS ’07. Alexandria, Virginia, USA: ACM, 2007, pp. 128–138. DOI: 10.1145/1315245.1315262.

[JNR02] Rajeev Joshi, Greg Nelson, and Keith Randall. “Denali: A Goal-directed Superoptimizer”. In: *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*. PLDI ’02. Berlin, Germany: ACM, June 2002, pp. 304–314. DOI: 10.1145/512529.512566.

[JH10] Li Jun and Yu Huiping. “Trusted Full Disk Encryption Model Based on TPM”. In: *Proceedings of the 2nd International Conference on Information Science and Engineering*. ICISE ’10. IEEE, Dec. 2010, pp. 1–4. DOI: 10.1109/ICISE.2010.5689500.

[Kao+12] Yung-Wei Kao, Xin Zhang, Ahren Studer, and Adrian Perrig. “Mobile encryption for laptop data protection (MELP)”. In: *IET Information Security* 6.4 (Dec. 2012), pp. 291–298. DOI: 10.1049/iet-ifs.2011.0347.

[KPW16] David Kaplan, Jeremy Powell, and Tom Woller. *AMD Memory Encryption*. White Paper. Apr. 2016.

[KPK14] Vasileios P. Kemerlis, Michalis Polychronakis, and Angelos D. Keromytis. “ret2dir: Rethinking Kernel Isolation”. In: *Proceedings of the 23rd USENIX Security Symposium*. SEC ’14. San Diego, CA, USA: USENIX Association, Aug. 2014, pp. 957–972.

[KJ03] Rick Kennell and Leah H. Jamieson. “Establishing the Genuinity of Remote Computer Systems”. In: *Proceedings of the 12th USENIX Security Symposium*. SEC ’03. Washington, DC, USA: USENIX Association, Aug. 2003.

[Kim+14] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. “Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors”. In: *Proceeding of the 41st Annual International Symposium on Computer Architecture*. ISCA ’14. Minneapolis, Minnesota, USA: IEEE Press, June 2014, pp. 361–372.

[KBA02] Vladimir Kiriansky, Derek Bruening, and Saman P. Amarasinghe. “Secure Execution via Program Shepherding”. In: *Proceedings of the 11th USENIX Security Symposium*. SEC ’02. San Francisco, CA, USA: USENIX Association, Aug. 2002, pp. 191–206.

[KS04] Alexander Klimov and Adi Shamir. “New Cryptographic Primitives Based on Multiword T-Functions”. In: *Fast Software Encryption, 11th International Workshop, FSE 2004, Delhi, India, February 5-7, 2004, Revised Papers*. Vol. 3017. Lecture Notes in Computer Science. Springer, 2004, pp. 1–15. DOI: 10.1007/978-3-540-25937-4_1.

[Koc+18] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. *Spectre Attacks: Exploiting Speculative Execution*. Jan. 2018. arXiv: 1801.01203 [cs.CR]. URL: https://arxiv.org/abs/1801.01203.

[Kol+09] Clemens Kolbitsch, Paolo Milani Comparetti, Christopher Kruegel, Engin Kirda, Xiao-yong Zhou, and XiaoFeng Wang. “Effective and Efficient Malware Detection at the End Host”. In: *Proceedings of the 18th USENIX Security Symposium*. SEC ’09. Montreal, Canada: USENIX Association, Aug. 2009, pp. 351–366.

[Kor10] Tim Kornau. “Return Oriented Programming for the ARM Architecture”. Master’s Thesis. Ruhr-Universität Bochum, Dec. 2010.

[Lam73] Butler W. Lampson. “A Note on the Confinement Problem”. In: *Commun. ACM* 16.10 (Oct. 1973), pp. 613–615. DOI: 10.1145/362375.362389.

[LSS12] Andreas Leicher, Andreas U. Schmidt, and Yogendra Shah. “Smart OpenID: A Smart Card Based OpenID Protocol”. In: *Information Security and Privacy Research - 27th IFIP TC 11 Information Security and Privacy Conference, SEC 2012, Heraklion, Crete, Greece, June 4-6, 2012. Proceedings*. Vol. 376. IFIP Advances in Information and Communication Technology. Springer, 2012, pp. 75–86. DOI: 10.1007/978-3-642-30436-1_7.

[Lie+00] David Lie, Chandramohan Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John Mitchell, and Mark Horowitz. “Architectural Support for Copy and Tamper Resistant Software”. In: *SIGPLAN Not.* 35.11 (Nov. 2000), pp. 168–177. DOI: 10.1145/356989.357005.

[LYQ07] Kun Lin, Lin Yuan, and Gang Qu. “SecureGo: A Hardware-Software Co-Protection against Identity Theft in Online Transaction”. In: *Proceedings of the 2007 ECSIS Symposium on Bio-inspired, Learning, and Intelligent Systems for Security*. BLISS ’07. Edinburgh, United Kingdom: IEEE, Aug. 2007, pp. 59–64. DOI: 10.1109/BLISS.2007.30.

[Lin] Linux Developers. *CGroup Freezer Documentation*. URL: https://www.kernel.org/doc/Documentation/cgroup-v1/freezer-subsystem.txt (Accessed: 18 Mar. 2019).

[Lip+18a] Moritz Lipp, Misiker Tadesse Aga, Michael Schwarz, Daniel Gruss, Clémentine Maurice, Lukas Raab, and Lukas Lamster. *Nethammer: Inducing Rowhammer Faults through Network Requests*. May 2018. arXiv: 1805.04956 [cs.CR]. URL: https://arxiv.org/abs/1805.04956.

[Lip+18b] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. *Meltdown*. Jan. 2018. arXiv: `1801.01207 [cs.CR]`. URL: `https://arxiv.org/abs/1801.01207`.

[Mar+19] Athanasios Markettos, Colin Rothwell, Brett F. Gutstein, Allison Pearce, Peter G. Neumann, Simon W. Moore, and Robert N. M. Watson. "Thunderclap: Exploring Vulnerabilities in Operating System IOMMU Protection via DMA from Untrustworthy Peripherals". In: *26th Annual Network and Distributed System Security Symposium*. NDSS '19. San Diego, California, USA: The Internet Society, Feb. 2019.

[MPB10] Lorenzo Martignoni, Roberto Paleari, and Danilo Bruschi. "Conqueror: Tamper-Proof Code Execution on Legacy Systems". In: *Detection of Intrusions and Malware, and Vulnerability Assessment, 7th International Conference, DIMVA 2010, Bonn, Germany, July 8-9, 2010. Proceedings*. Vol. 6201. Lecture Notes in Computer Science. Springer, 2010, pp. 21–40. DOI: `10.1007/978-3-642-14215-4_2`.

[McK+13] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. "Innovative Instructions and Software Model for Isolated Execution". In: *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*. HASP '13. Tel-Aviv, Israel: ACM, June 2013, 10:1–10:1. DOI: `10.1145/2487726.2488368`.

[MG18] Carlo Meijer and Bernard van Gastel. *Self-encrypting deception: weaknesses in the encryption of solid state drives (SSDs)*. Tech. rep. Radboud University and Open University of the Netherlands, Nov. 2018. URL: `https://www.ru.nl/publish/pages/909275/draft-paper_1.pdf` (Accessed: 5 Mar. 2019).

[MN11] Roberto Mijat and Andy Nightingale. *Virtualization is Coming to a Platform Near You*. White Paper. ARM Limited, Jan. 2011.

[MHH19] Mathias Morbitzer, Manuel Huber, and Julian Horsch. "Extracting Secrets from Encrypted Virtual Machines". In: *Proceedings of the 9th ACM Conference on Data and Application Security and Privacy*. CODASPY '19. Richardson, Texas, USA: ACM, Mar. 2019. DOI: `10.1145/3292006.3300022`.

[Mor+18] Mathias Morbitzer, Manuel Huber, Julian Horsch, and Sascha Wessel. "SEVered: Subverting AMD's Virtual Machine Encryption". In: *Proceedings of the 11th European Workshop on Systems Security*. EuroSec'18. Porto, Portugal: ACM, Apr. 2018. DOI: `10.1145/3193111.3193112`.

[MFD11] Tilo Müller, Felix C. Freiling, and Andreas Dewald. “TRESOR Runs Encryption Securely Outside RAM”. In: *Proceedings of the 20th USENIX Security Symposium*. SEC ’11. San Francisco, CA, USA: USENIX Association, Aug. 2011.

[MS13] Tilo Müller and Michael Spreitzenbarth. “FROST - Forensic Recovery of Scrambled Telephones”. In: *Applied Cryptography and Network Security - 11th International Conference, ACNS 2013, Banff, AB, Canada, June 25-28, 2013. Proceedings*. Vol. 7954. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2013, pp. 373–388. DOI: 10.1007/978-3-642-38980-1_23.

[MTF12] Tilo Müller, Benjamin Taubmann, and Felix C. Freiling. “TreVisor - OS-Independent Software-Based Full Disk Encryption Secure against Main Memory Attacks”. In: *Applied Cryptography and Network Security - 10th International Conference, ACNS 2012, Singapore, June 26-29, 2012. Proceedings*. Vol. 7341. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2012, pp. 66–83. DOI: 10.1007/978-3-642-31284-7_5.

[Ngu+09] Anh M. Nguyen, Nabil Schear, HeeDong Jung, Apeksha Godiyal, Samuel T. King, and Hai D. Nguyen. “MAVMM: Lightweight and Purpose Built VMM for Malware Analysis”. In: *Proceedings of the 2009 Annual Computer Security Applications Conference*. ACSAC ’09. IEEE, Dec. 2009, pp. 441–450.

[NXP16] NXP. *UM10360 – LPC176x/5x User manual*. Version 4.1. Dec. 2016. URL: http://www.nxp.com/documents/user_manual/UM10360.pdf (Accessed: 14 Mar. 2019).

[Off16] Office for National Statistics. *Focus on property crime: year ending March 2016*. Nov. 24, 2016. URL: https://www.ons.gov.uk/peoplepopulationandcommunity/crimeandjustice/bulletins/focusonpropertycrime/yearendingmarch2016 (Accessed: 18 Mar. 2019).

[Ona+10] Kaan Onarlioglu, Leyla Bilge, Andrea Lanzi, Davide Balzarotti, and Engin Kirda. “G-Free: Defeating Return-oriented Programming Through Gadget-less Binaries”. In: *Proceedings of the 26th Annual Computer Security Applications Conference*. ACSAC ’10. Austin, Texas, USA: ACM, Dec. 2010, pp. 49–58. DOI: 10.1145/1920261.1920269.

[OOY08] Koichi Onoue, Yoshihiro Oyama, and Akinori Yonezawa. “Control of System Calls from Outside of Virtual Machines”. In: *Proceedings of the 2008 ACM Symposium on Applied Computing*. SAC ’08. Fortaleza, Ceara, Brazil: ACM, Mar. 2008, pp. 2116–1221. DOI: 10.1145/1363686.1364196.

[PPK13] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. "Transparent ROP Exploit Mitigation Using Indirect Branch Tracing". In: *Proceedings of the 22th USENIX Security Symposium*. SEC '13. Washington, DC, USA: USENIX Association, Aug. 2013, pp. 447–462.

[PBG15] Mathias Payer, Antonio Barresi, and Thomas R. Gross. "Fine-Grained Control-Flow Integrity Through Binary Hardening". In: *Detection of Intrusions and Malware, and Vulnerability Assessment, 12th International Conference, DIMVA 2015, Milan, Italy, July 9-10, 2015. Proceedings*. Springer, 2015, pp. 144–164. DOI: 10.1007/978-3-319-20550-2_8.

[PG11] Mathias Payer and Thomas R. Gross. "Fine-grained User-space Security Through Virtualization". In: *Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. VEE '11. Newport Beach, California, USA: ACM, Mar. 2011, pp. 157–168. DOI: 10.1145/1952682.1952703.

[Pay+08] Bryan D. Payne, Martim Carbone, Monirul I. Sharif, and Wenke Lee. "Lares: An Architecture for Secure Active Monitoring Using Virtualization". In: *Proceedings of the 2008 IEEE Symposium on Security and Privacy*. S&P '08. Oakland, California, USA: IEEE, May 2008, pp. 233–247. DOI: 10.1109/SP.2008.24.

[PD10] Adrian Perrig and Leendert van Doorn. *Refutation of "On the Difficulty of Software-Based Attestation of Embedded Devices"*. Aug. 2010. URL: http://www.netsec.ethz.ch/publications/papers/perrig-ccs-refutation.pdf (Accessed: 14 Mar. 2019).

[Pet10] Peter A. H. Peterson. "Cryptkeeper: Improving security with encrypted RAM". In: *Proceedings of the 2010 IEEE International Conference on Technologies for Homeland Security*. HST '10. IEEE, Nov. 2010, pp. 120–126.

[PH07] Nick L. Petroni Jr. and Michael Hicks. "Automated Detection of Persistent Kernel Control-flow Attacks". In: *Proceedings of the 14th ACM Conference on Computer and Communications Security*. CCS '07. Alexandria, Virginia, USA: ACM, Oct. 2007, pp. 103–115. DOI: 10.1145/1315245.1315260.

[PSE11] Jonas Pfoh, Christian A. Schneider, and Claudia Eckert. "Nitro: Hardware-Based System Call Tracing for Virtual Machines". In: *Advances in Information and Computer Security - 6th International Workshop, IWSEC 2011, Tokyo, Japan, November 8-10, 2011. Proceedings*. Vol. 7038. Lecture Notes in Computer Science. Springer, 2011, pp. 96–112. DOI: 10.1007/978-3-642-25141-2_7.

[PG74] Gerald J. Popek and Robert P. Goldberg. "Formal Requirements for Virtualizable Third Generation Architectures". In: *Commun. ACM* 17.7 (July 1974), pp. 412–421. DOI: 10.1145/361011.361073.

[Qua17] Qualcomm Technologies, Inc. *Pointer Authentication on ARMv8.3. Design and Analysis of the New Software Security Instructions*. White Paper. Jan. 2017. URL: https://www.qualcomm.com/media/documents/files/whitepaper-pointer-authentication-on-armv8-3.pdf (Accessed: 18 Mar. 2019).

[Raj+16] Himanshu Raj, Stefan Saroiu, Alec Wolman, Ronald Aigner, Jeremiah Cox, Paul England, Chris Fenner, Kinshuman Kinshumann, Jork Loeser, Dennis Mattoon, Magnus Nystrom, David Robinson, Rob Spiger, Stefan Thom, and David Wooten. "fTPM: A Software-Only Implementation of a TPM Chip". In: *Proceedings of the 25th USENIX Security Symposium*. SEC '16. Austin, TX, USA: USENIX Association, Aug. 2016, pp. 841–856.

[RLX11] Junghwan Rhee, Zhiqiang Lin, and Dongyan Xu. "Characterizing Kernel Malware Behavior with Kernel Data Access Patterns". In: *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*. ASIACCS '11. Hong Kong, China: ACM, 2011, pp. 207–216. DOI: 10.1145/1966913.1966940.

[Rhe+09] Junghwan Rhee, Ryan Riley, Dongyan Xu, and Xuxian Jiang. "Defeating Dynamic Data Kernel Rootkit Attacks via VMM-Based Guest-Transparent Monitoring". In: *Proceedings of the The Forth International Conference on Availability, Reliability and Security*. ARES '09. Fukuoka, Japan: IEEE, Mar. 2009, pp. 74–81. DOI: 10.1109/ARES.2009.116.

[Rie+11] Konrad Rieck, Philipp Trinius, Carsten Willems, and Thorsten Holz. "Automatic Analysis of Malware Behavior Using Machine Learning". In: *Journal of Computer Security* 19.4 (Dec. 2011), pp. 639–668. DOI: 10.3233/JCS-2010-0410.

[Ros14] Dan Rosenberg. *QSEE TrustZone Kernel Integer Overflow Vulnerability*. July 2014. URL: https://wikileaks.org/sony/docs/05/docs/Hacks/us-14-Rosenberg-Reflections-On-Trusting-TrustZone-WP.pdf (Accessed: 18 Mar. 2019).

[Sch+18] Michael Schwarz, Martin Schwarzl, Moritz Lipp, and Daniel Gruss. *NetSpectre: Read Arbitrary Memory over Network*. July 2018. URL: https://misc0110.net/web/files/netspectre.pdf.

[Sea15] Mark Seaborn. *Exploiting the DRAM rowhammer bug to gain kernel privileges*. Google Project Zero. Mar. 9, 2015. URL: https://googleprojectzero.blogspot.com/2015/03/exploiting-dram-rowhammer-bug-to-gain.html (Accessed: 18 Mar. 2019).

[Sea06] Seagate Corporation. *Drivetrust Technology: A Technical Overview*. White Paper. Oct. 2006. URL: https://www.seagate.com/docs/pdf/whitepaper/TP564_DriveTrust_Oct06.pdf (Accessed: 14 Mar. 2019).

[SV17] Ben Seri and Gregory Vishnepolsky. *The dangers of Bluetooth implementations: Unveiling zero day vulnerabilities and security flaws in modern Bluetooth stacks.* White Paper. 2017. URL: `http://go.armis.com/blueborne-technical-paper` (Accessed: 14 Mar. 2019).

[Ses09] Arvind Seshadri. "A Software Primitive for Externally-verifiable Untampered Execution and Its Applications to Securing Computing Systems". AAI3382437. PhD thesis. Pittsburgh, PA, USA: Carnegie Mellon University, 2009.

[Ses+06] Arvind Seshadri, Mark Luk, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. "SCUBA: Secure Code Update By Attestation in Sensor Networks". In: *Proceedings of the 5th ACM Workshop on Wireless Security*. WiSe '06. Los Angeles, California: ACM, Sept. 2006, pp. 85–94. DOI: `10.1145/1161289.1161306`.

[Ses+07] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. "SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes". In: *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*. SOSP '07. Stevenson, Washington, USA: ACM, Oct. 2007, pp. 335–350. DOI: `10.1145/1294261.1294294`.

[Ses+05] Arvind Seshadri, Mark Luk, Elaine Shi, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. "Pioneer: Verifying Code Integrity and Enforcing Untampered Code Execution on Legacy Systems". In: *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*. SOSP '05. Brighton, United Kingdom: ACM, Oct. 2005, pp. 1–16. DOI: `10.1145/1095810.1095812`.

[Ses+04] Arvind Seshadri, Adrian Perrig, Leendert van Doorn, and Pradeep K. Khosla. "SWATT: SoftWare-based ATTestation for Embedded Devices". In: *Proceedings of the 2004 IEEE Symposium on Security and Privacy*. S&P '04. Berkeley, CA, USA: IEEE, May 2004, pp. 272–282. DOI: `10.1109/SECPRI.2004.1301329`.

[Sha07] Hovav Shacham. "The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86)". In: *Proceedings of the 14th ACM Conference on Computer and Communications Security*. CCS '07. Alexandria, Virginia, USA: ACM, 2007, pp. 552–561. DOI: `10.1145/1315245.1315313`.

[Sha+04] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. "On the Effectiveness of Address-space Randomization". In: *Proceedings of the 11th ACM Conference on Computer and Communications Security*. CCS '04. Washington DC, USA: ACM, Oct. 2004, pp. 298–307. DOI: `10.1145/1030083.1030124`.

[SCT04] Umesh Shankar, Monica Chew, and J. D. Tygar. "Side Effects Are Not Sufficient to Authenticate Software". In: *Proceedings of the 13th USENIX Security Symposium*. SEC '04. San Diego, CA, USA: USENIX Association, Aug. 2004, pp. 89–102.

[Sha+09] Monirul I. Sharif, Wenke Lee, Weidong Cui, and Andrea Lanzi. "Secure in-VM Monitoring Using Hardware Virtualization". In: *Proceedings of the 16th ACM Conference on Computer and Communications Security*. CCS '09. Chicago, Illinois, USA: ACM, Nov. 2009, pp. 477–487. DOI: 10.1145/1653662.1653720.

[Sim11] Patrick Simmons. "Security Through Amnesia: A Software-based Solution to the Cold Boot Attack on Disk Encryption". In: *Proceedings of the 27th Annual Computer Security Applications Conference*. ACSAC '11. Orlando, Florida, USA: ACM, Dec. 2011, pp. 73–82. DOI: 10.1145/2076732.2076743.

[Sri+11] Deepa Srinivasan, Zhi Wang, Xuxian Jiang, and Dongyan Xu. "Process Outgrafting: An Efficient "out-of-VM" Approach for Fine-grained Process Execution Monitoring". In: *Proceedings of the 18th ACM Conference on Computer and Communications Security*. CCS '11. Chicago, Illinois, USA: ACM, Oct. 2011, pp. 363–374. DOI: 10.1145/2046707.2046751.

[SG11] Abhinav Srivastava and Jonathon T. Giffin. "Efficient Monitoring of Untrusted Kernel-Mode Execution". In: *Proceedings of the 18th Annual Network and Distributed System Security Symposium*. NDSS 2011. San Diego, California, USA: The Internet Society, Feb. 2011.

[Sta11] Frank Stajano. "Pico: No More Passwords!" In: *Security Protocols XIX - 19th International Workshop, Cambridge, UK, March 28-30, 2011, Revised Selected Papers*. Vol. 7114. Lecture Notes in Computer Science. Springer, 2011, pp. 49–81. DOI: 10.1007/978-3-642-25867-1_6.

[Sta16] StatCounter. *Mobile and tablet internet usage exceeds desktop for first time worldwide*. Nov. 1, 2016. URL: http://gs.statcounter.com/press/mobile-and-tablet-internet-usage-exceeds-desktop-for-first-time-worldwide (Accessed: 18 Mar. 2019).

[SB12] Patrick Stewin and Iurii Bystrov. "Understanding DMA Malware". In: *Detection of Intrusions and Malware, and Vulnerability Assessment - 9th International Conference, DIMVA 2012, Heraklion, Crete, Greece, July 26-27, 2012, Revised Selected Papers*. Vol. 7591. Lecture Notes in Computer Science. Springer, 2012, pp. 21–41. DOI: 10.1007/978-3-642-37300-8_2.

[Suh+03] G. Edward Suh, Dwaine Clarke, Blaise Gassend, Marten van Dijk, and Srinivas Devadas. “AEGIS: Architecture for Tamper-evident and Tamper-resistant Processing”. In: *Proceedings of the 17th Annual International Conference on Supercomputing*. ICS ’03. San Francisco, CA, USA: ACM, June 2003, pp. 160–171. DOI: 10.1145/782814.782838.

[Sze+13] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. “SoK: Eternal War in Memory”. In: *Proceedings of the 2013 IEEE Symposium on Security and Privacy*. S&P ’13. Berkeley, CA, USA: IEEE, May 2013, pp. 48–62. DOI: 10.1109/SP.2013.13.

[Tan+12] Yang Tang, Phillip Ames, Sravan Bhamidipati, Ashish Bijlani, Roxana Geambasu, and Nikhil Sarda. “CleanOS: Limiting Mobile Data Exposure with Idle Eviction”. In: *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation*. OSDI ’12. Hollywood, CA, USA: USENIX Association, Oct. 2012, pp. 77–91.

[Tat+18] Andrei Tatar, Radhesh Krishnan, Elias Athanasopoulos, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. “Throwhammer: Rowhammer Attacks over the Network and Defenses”. In: *Proceedings of the 2018 USENIX Annual Technical Conference*. ATC ’18. Boston, MA, USA: USENIX Association, July 2018.

[Tau+15] Benjamin Taubmann, Manuel Huber, Sascha Wessel, Lukas Heim, Hans Peter Reiser, and Georg Sigl. “A Lightweight Framework for Cold Boot Based Forensics on Mobile Devices”. In: *Proceedings of the 10th International Conference on Availability, Reliability and Security*. ARES ’15. Toulouse, France: IEEE, Aug. 2015, pp. 120–128. DOI: 10.1109/ARES.2015.47.

[Tex12] Texas Instruments. *OMAP35x Applications Processor – Technical Reference Manual*. SPRUF98Y. Dec. 2012. URL: http://www.ti.com/lit/pdf/spruf98 (Accessed: 18 Mar. 2019).

[Tic+14] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. “Enforcing Forward-edge Control-flow Integrity in GCC & LLVM”. In: *Proceedings of the 23rd USENIX Security Symposium*. SEC ’14. San Diego, CA: USENIX Association, Aug. 2014, pp. 941–955.

[Tir07] Kris Tiri. “Side-channel Attack Pitfalls”. In: *Proceedings of the 44th Annual Design Automation Conference*. DAC ’07. San Diego, California: ACM, June 2007, pp. 15–20. DOI: 10.1145/1278480.1278485.

[Tru11] Trusted Computing Group. *Trusted Platform Module Main Specification 1.2 Revision 116*. Mar. 2011.

[TCS05] J. P. Tual, A. Couchard, and L. Sourgen. “USB Full Speed Enabled Smart Cards for Consumer Electronics Applications”. In: *Proceedings of the Ninth International Symposium on Consumer Electronics*. ISCE ’05. IEEE, June 2005, pp. 230–236. DOI: 10.1109/ISCE.2005.1502376.

[UMK11] Pascal Urien, Estel Marie, and Christophe Kiennert. “A New Convergent Identity System Based on EAP-TLS Smart Cards”. In: *Proceedings of the 2011 Conference on Network and Information Systems Security*. SAR-SSI ’11. La Rochelle, France: IEEE, May 2011, pp. 1–6. DOI: 10.1109/SAR-SSI.2011.5931373.

[USB00] USB Implementers Forum. *Universal Serial Bus Specification Revision 2.0*. Apr. 2000.

[VP17] Mathy Vanhoef and Frank Piessens. “Key Reinstallation Attacks: Forcing Nonce Reuse in WPA2”. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’17. Dallas, Texas, USA: ACM, Oct. 2017, pp. 1313–1328. DOI: 10.1145/3133956.3134027.

[VH11] Prashant Varanasi and Gernot Heiser. “Hardware-supported Virtualization on ARM”. In: *Proceedings of the Second Asia-Pacific Workshop on Systems*. APSys ’11. Shanghai, China: ACM, July 2011, 11:1–11:5. DOI: 10.1145/2103799.2103813.

[VY05] Amit Vasudevan and Ramesh Yerraballi. “Stealth Breakpoints”. In: *Proceedings of the 21st Annual Computer Security Applications Conference*. ACSAC ’05. IEEE, Dec. 2005.

[Vee+15] Victor van der Veen, Dennis Andriesse, Enes Göktaş, Ben Gras, Lionel Sambuc, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. “Practical Context-Sensitive CFI”. In: *Proceedings of the 22nd ACM Conference on Computer and Communications Security*. CCS ’15. Denver, Colorado, USA: ACM, Oct. 2015, pp. 927–940. DOI: 10.1145/2810103.2813673.

[Vee+16] Victor van der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clementine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. “Drammer: Deterministic Rowhammer Attacks on Mobile Platforms”. In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’16. Vienna, Austria: ACM, Oct. 2016, pp. 1675–1689. DOI: 10.1145/2976749.2978406.

[Wan+13] Tielei Wang, Kangjie Lu, Long Lu, Simon P. Chung, and Wenke Lee. “Jekyll on iOS: When Benign Apps Become Evil”. In: *Proceedings of the 22th USENIX Security Symposium*. SEC ’13. Washington, DC, USA: USENIX Association, 2013, pp. 559–572.

[Wei12] Ralf-Philipp Weinmann. “Baseband Attacks: Remote Exploitation of Memory Corruptions in Cellular Protocol Stacks”. In: *Proceedings of the 6th USENIX Workshop on Offensive Technologies*. WOOT ’12. Bellevue, WA, USA: USENIX Association, Aug. 2012, pp. 12–21.

[WS12] Sascha Wessel and Frederic Stumpf. “Page-based Runtime Integrity Protection of User and Kernel Code”. In: *Proceedings of the 5th European Workshop on Systems Security*. EuroSec’12. New York, NY, USA: ACM, 2012.

[Wil+12] Carsten Willems, Ralf Hund, Andreas Fobian, Dennis Felsch, Thorsten Holz, and Amit Vasudevan. “Down to the Bare Metal: Using Processor Features for Binary Analysis”. In: *Proceedings of the 28th Annual Computer Security Applications Conference*. ACSAC ’12. Orlando, Florida, USA: ACM, Dec. 2012, pp. 189–198. DOI: 10.1145/2420950.2420980.

[WOS05] Glenn Wurster, P. C. van Oorschot, and Anil Somayaji. “A Generic Attack on Checksumming-Based Software Tamper Resistance”. In: *Proceedings of the 2005 IEEE Symposium on Security and Privacy*. S&P ’05. Oakland, CA, USA: IEEE, May 2005, pp. 127–138. DOI: 10.1109/SP.2005.2.

[Xia+12] Yubin Xia, Yutao Liu, Haibo Chen, and Binyu Zang. “CFIMon: Detecting Violation of Control Flow Integrity Using Performance Counters”. In: *Proceedings of the 2012 IEEE/IFIP International Conference on Dependable Systems and Networks*. DSN ’12. Boston, MA, USA: IEEE, June 2012, pp. 1–12. DOI: 10.1109/DSN.2012.6263958.

[XTL11] Xi Xiong, Donghai Tian, and Peng Liu. “Practical Protection of Kernel Integrity for Commodity OS from Untrusted Extensions”. In: *Proceedings of the 18th Annual Network and Distributed System Security Symposium*. NDSS 2011. San Diego, California, USA: The Internet Society, Feb. 2011.

[YY12] Lok-Kwong Yan and Heng Yin. “DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis”. In: *Proceedings of the 21th USENIX Security Symposium*. SEC ’12. Bellevue, WA, USA: USENIX Association, Aug. 2012, pp. 569–584.

[Yan+11] Qiang Yan, Jin Han, Yingjiu Li, Robert H. Deng, and Tieyan Li. “A Software-based Root-of-trust Primitive on Multicore Platforms”. In: *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*. ASIACCS ’11. Hong Kong, China: ACM, 2011, pp. 334–343. DOI: 10.1145/1966913.1966957.

[YS08] Jisoo Yang and Kang G. Shin. “Using Hypervisor to Provide Data Secrecy for User Applications on a Per-page Basis”. In: *Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. VEE ’08. Seattle, WA, USA: ACM, 2008, pp. 71–80. DOI: 10.1145/1346256.1346267.

[YF14] Yuval Yarom and Katrina Falkner. “FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack”. In: *Proceedings of the 23rd USENIX Security Symposium*. SEC ’14. San Diego, CA, USA: USENIX Association, Aug. 2014, pp. 719–732.

[Yoo+08] Seehwan Yoo, Yunxin Liu, Cheol-Ho Hong, Chuck Yoo, and Yongguang Zhang. “MobiVMM: A Virtual Machine Monitor for Mobile Phones”. In: *Proceedings of the First Workshop on Virtualization in Mobile Computing*. MobiVirt ’08. Breckenridge, Colorado: ACM, June 2008, pp. 1–5. DOI: 10.1145/1622103.1622109.

[Yuba] Yubico Inc. *YubiHSM*. URL: https://www.yubico.com/products/yubihsm (Accessed: 14 Mar. 2019).

[Yubb] Yubico Inc. *YubiKeys*. URL: https://www.yubico.com/products/yubikey-hardware (Accessed: 14 Mar. 2019).

[Zha14] Dongli Zhang. *TrustFA: TrustZone-Assisted Facial Authentication on Smartphone*. Tech. rep. Dec. 2014. DOI: 10.13140/RG.2.1.1507.5441.

[ZS13] Mingwei Zhang and R. Sekar. “Control Flow Integrity for COTS Binaries”. In: *Proceedings of the 22nd USENIX Security Symposium*. SEC ’13. Washington, D.C.: USENIX Association, Aug. 2013, pp. 337–352.

[ZM16] Lianying Zhao and Mohammad Mannan. “Hypnoguard: Protecting Secrets Across Sleep-wake Cycles”. In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’16. Vienna, Austria: ACM, 2016, pp. 945–957. DOI: 10.1145/2976749.2978372.