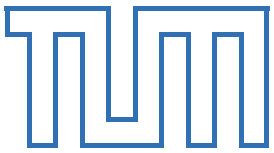


Fakultät für Informatik
der Technischen Universität München

Lehrstuhl für Sicherheit in der Informatik

System Architectures for Data Confidentiality and Frameworks for Main Memory Extraction

Manuel Huber

Dissertation

Fakultät für Informatik
der Technischen Universität München

Lehrstuhl für Sicherheit in der Informatik

System Architectures for Data Confidentiality and Frameworks for Main Memory Extraction

Manuel Huber

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. Jörg Ott
Prüfer der Dissertation:	1. Prof. Dr. Claudia Eckert
	2. Prof. Dr. Georg Sigl

Die Dissertation wurde am 25.09.2019 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 02.12.2019 angenommen.

Abstract

The sensitive data our modern computing systems process, transmit, and store poses a highly valuable target for cybercriminals and forensic investigators. Ranging from mobile devices and embedded devices in the internet of things to desktop and server environments, all systems expose characteristic vulnerabilities exploitable by remote and physical attack vectors of different sophistication. A resulting disclosure of confidential data, such as key material, passwords or classified documents, can lead to severe consequences for individuals, organizations and governmental bodies. The goal of this work is both the systematic protection of confidential data and the investigation of attack vectors to acquire confidential data. For the systematic protection, we design system architectures which assure data confidentiality on modern systems in the presence of both remote and physical adversaries. For the investigation of attack vectors, we develop sophisticated data extraction frameworks to demonstrate the potential of adversaries and to urge the need for carefully designed system architectures.

We pursue a systematic approach where we first design a secure architecture to isolate system resources based on OS-level virtualization. The architecture enables different isolated execution environments, called containers, to restrict remote adversaries exploiting a vulnerable container from accessing other containers possibly processing confidential data. While the design is generally platform-independent, we first tailor the architecture to mobile devices. This enables us to operate multiple virtualized Android containers on a single device having one container run in foreground and the others in background. We further demonstrate the applicability of this architecture in real-world ecosystems by providing a holistic security concept. This concept covers identity management and device provisioning and enrollment based on a public key infrastructure and secure elements for users, such as smartcards. In the next step, we tailor the architecture to embedded use cases, such as to platforms for the internet of things or cyber-physical systems. We demonstrate how to integrate the architecture into productive environments at the example of a cross-domain data exchange platform connecting organizations.

While our virtualization architectures for resource isolation defend against remote adversaries, these do not remediate physical memory attack scenarios, such as cold boot or DMA attacks. These attacks make it possible to extract the confidential data stored in main memory while circumventing OS protection mechanisms. To emphasize the urge for the protection of data against physical adversaries, we design a framework for cold boot attacks on mobile devices. Our framework enables the systematic extraction of data from the main memory of mobile devices. Since the main memory is not encrypted and thus stored in plaintext on common systems, memory extraction leveraging the cold boot attack is not only limited to mobile devices.

To counter the threat of memory attacks from physical attack vectors, we design main memory encryption architectures for different types of systems. For notebooks and desktop systems, we introduce an architecture to encrypt the main memory during device suspension. This makes memory attacks futile on suspended, likely unattended devices. While we also design this architecture for environments virtualized with a hypervisor, it does not apply to the different usage model of mobile devices. The operating systems of mobile devices do not fully suspend to support background activities in the absence or inactivity of the users. To also enable memory protection on mobile devices, we develop a main memory encryption architecture on the granularity of process groups. We combine our secure virtualization and main memory encryption architectures, making it possible to encrypt main memory of individual containers. We keep actively used containers in normal operation and encrypt suspended containers in background. The combination of the architectures not only protects the Android containers from remote adversaries, but also from physical attackers once suspended. For systems which do not necessarily suspend, such as embedded IoT devices, we present an alternate architecture for transparent runtime memory encryption based on a minimal hypervisor.

When encrypting memory at runtime, attackers may still fully interact with the non-suspended systems. We point out that such encryption techniques require particularly careful design by developing a memory extraction framework targeting virtual machines on server systems that leverage hardware extensions for transparent runtime memory encryption. Our framework makes it possible to fully extract the memory of encrypted virtual machines possibly being under heavy load, such as virtual machines hosting productive web servers open to the public. We additionally equip our framework with techniques to specifically extract targeted resources in short time, such as TLS, SSH, or disk encryption keys, requiring only minimal interaction with the system and leaving only a small footprint.

Kurzfassung

Die sensitiven Daten, die unsere heutigen Systeme verarbeiten, übertragen und speichern stellen für Kriminelle und Forensiker ein äußerst attraktives Ziel dar. Zu diesen Systemen zählen beispielsweise mobile Endgeräte, eingebettete Geräte für das Internet der Dinge, aber auch Desktop- und Serverumgebungen. All diese Systeme weißen charakteristische Schwachstellen auf, welche beiderseits von physischen Angreifern und Angreifern über entfernte Schnittstellen hinweg mittels Angriffsvektoren verschiedener Komplexität ausgenutzt werden können. Eine resultierende Enthüllung vertraulicher Daten, wie etwa Schlüsselmaterial, Passwörter oder klassifizierter Dokumente, kann zu starken Konsequenzen für Einzelpersonen, aber auch für Organisationen und Behörden führen. Das Ziel dieser Arbeit ist beiderseits der systematische Schutz vertraulicher Daten, als auch das Untersuchen von Angriffsvektoren, um an vertrauliche Daten zu gelangen. Für den systematischen Schutz entwerfen wir Systemarchitekturen, die die Vertraulichkeit von Daten auf modernen Rechengeräten unter der Bedrohungssituation von beiderseits entfernten und physisch präsenten Angreifern sicherstellen. Zur Untersuchung von Angriffsvektoren entwickeln wir fortgeschrittene Rahmenwerke zur Extrahierung von Daten, um das Potential von Angreifern zu demonstrieren und um den Bedarf für sorgfältig entworfene Systemarchitekturen zu motivieren.

Wir verfolgen hierbei einen systematischen Ansatz, bei dem wir zunächst eine Sicherheitsarchitektur entwerfen, welche Systemressourcen auf Basis von Virtualisierung auf Betriebssystem-Ebene isoliert. Diese Architektur ermöglicht es, verschiedene voneinander isolierte Ausführungsumgebungen, sogenannte Container, zu betreiben, um Angreifer, die eine Software-Schwachstelle eines verwundbaren Containers ausnutzen, innerhalb des Containers zu isolieren. Dies bewahrt andere Container mit deren vertraulichen Daten vor den Auswirkungen des Angriffs. Während der Architekturentwurf generell plattformunabhängig ist, schneiden wir die Architektur zunächst auf mobile Endgeräte zu, auf denen wir mehrere, virtualisierte Android-Container betreiben. Dabei läuft ein Container für den Nutzer sichtbar im Vordergrund und die anderen Container im Hintergrund. Des weiteren demonstrieren wir die Anwendbarkeit der Architektur innerhalb produktiv genutzter Ökosysteme durch Bereitstellen eines ganzheitlichen Sicherheitskonzepts. Dies beinhaltet das Verwalten von Identitäten, sowie Geräteprovisionierung und -ausrollung, basierend auf einer Infrastruktur mit öffentlichen Schlüsselpaaren und Sicherheitselementen, wie Chipkarten, für Endnutzer. Im nächsten Schritt schneiden wir die Architektur auf Benutzungsszenarien im Bereich eingebetteter Systeme zu, wie zum Beispiel auf Plattformen im Internet der Dinge, oder auf Cyber-physische Systeme. Wir beschreiben darüber hinaus die Integration dieser Architektur in produktive Umgebungen am Beispiel einer Bereichsgrenzen überschreitenden Plattform zum Datenaustausch zwischen Organisationen.

Während die Sicherheitsarchitekturen zur Isolation von Ausführungsumgebungen gegen Angreifer wirkt, welche Softwareschwachstellen ausnutzen, verhindert diese aber nicht die Angriffsszenarien auf den Hauptspeicher durch physische Angreifer. Ein Beispiel dafür sind Kaltstart- oder DMA-Attacken, welche es ermöglichen, vertrauliche, im Hauptspeicher abgelegte Daten unter umgehen der Schutzmechanismen der Betriebssysteme zu extrahieren. Um die Notwendigkeit des Schutzes von Daten gegen physische Angreifer aufzuzeigen, entwerfen wir ein Rahmenwerk für Kaltstartangriffe auf mobilen Endgeräten. Unser Rahmenwerk ermöglicht es, systematisch Daten aus dem Hauptspeicher von mobilen Endgeräten zu extrahieren. Da der Hauptspeicher auf herkömmlichen Systemen unverschlüsselt ist und deshalb im Klartext vorliegt, ist die Extrahierung von Hauptspeicher mittels Kaltstartangriffen nicht auf mobile Endgeräte beschränkt.

Um der Gefahr von Speicherangriffen durch physische Angreifer entgegenzuwirken, entwerfen wir Architekturen zur Hauptspeicherverschlüsselung für verschiedenartige Systeme. Für Klapprechner und Desktop-Geräte führen wir eine Architektur zur Verschlüsselung des Hauptspeichers während des Einleiten des Ruhemodus ein, was Speicherangriffe auf ruhenden, dann oftmals unbeaufsichtigten Geräten zwecklos macht. Während wir diese Architektur auch für Umgebungen, welche mit einem Hypervisor virtualisiert werden, entwerfen, lässt sich diese nicht auf das Nutzungsmodell mobiler Endgeräte transferieren. Betriebssysteme auf mobilen Endgeräten setzen das System nicht vollständig in den Ruhemodus, um Hintergrundaktivitäten während der Absenz oder Inaktivität des Nutzers zu ermöglichen. Um ebenso Speicherschutz für mobile Endgeräte zu ermöglichen, entwerfen wir eine Architektur zur Hauptspeicherverschlüsselung auf Basis von Prozessgruppen. Wir kombinieren unsere Virtualisierungs- und Hauptspeicherverschlüsselungs-Architekturen, was es ermöglicht den Hauptspeicher auf individueller Container zu verschlüsseln. Wir halten dabei aktiv genutzte Container im normalen, unverschlüsselten Betrieb, und verschlüsseln sich im Ruhemodus befindliche Hintergrund-Container. Die Kombination der Architekturen schützt die Android-Container nicht nur vor entfernten Angreifen, sondern auch von physischen Angreifern, sobald sich diese im Ruhemodus befinden. Für Systeme, auf welchen nicht notwendigerweise ein Ruhemodus vorgesehen ist, wie zum Beispiel eingebettete Geräte im Internet der Dinge, präsentieren wir eine alternative Architektur zur transparenten Laufzeitverschlüsselung basierend auf einem minimalen Hypervisor.

Bei transparenter Verschlüsselung des Hauptspeichers zur Laufzeit können Angreifer allerdings im vollen Umfang mit dem nicht ruhenden System interagieren. Wir zeigen auf, dass der Entwurf solcher Verschlüsselungstechniken besonders gründlicher Konzeption bedarf, indem wir ein Rahmenwerk zur Extrahierung von Speicher von virtuellen Maschinen auf Serversystemen, welche Hardware-Erweiterungen für die Laufzeit-basierte Hauptspeicherverschlüsselung nutzen, entwerfen. Unser Rahmenwerk ermöglicht es, den kompletten Hauptspeicher verschlüsselter virtueller Maschinen, welche auf derartigen Servern betrieben werden und welche möglicherweise unter hoher Last stehen, zu extrahieren. Ein Beispiel sind öffentlich erreichbare, produktiv eingesetzte Web Server. Zusätzlich statten wir dieses Rahmenwerk mit Techniken aus, um spezifische Ressourcen, wie TLS- oder SSH-Schlüssel, oder Schlüssel zur Festplattenverschlüsselung, innerhalb kürzester Zeit gezielt zu extrahieren. Dabei bedarf es nur minimaler Interaktion mit dem System, auf welchem dadurch ein lediglich geringer digitaler Fußabdruck entsteht.

Acknowledgments

I would like to express my sincere gratitude to all of those who accompanied me on the long way with all the ups and downs a PhD brings with it. This way led from the initial idea of pursing a PhD while completing my master's degree in December 2013 over publishing a decent amount of papers throughout the following years to completing the thesis in 2019. The many steps in between these stages were important experiences and by far not only centered around writing papers and attending conferences. As a matter of fact, my personal daily work routine turned out to be guided by the principle that there exists no fixed routine. With the predominant project work at Fraunhofer AISEC, with business travels, with research, with occasionally attending TUM Graduate School, and with the attempt to lead a near-normal personal life with versatile hobbies and some madness, the last few years, while rich in experiences, passed in the blink of an eye.

Having all this in mind, my particular gratitude goes to the following people: My supervisor Prof. Dr. Claudia Eckert, as well as Prof. Dr. Georg Sigl, both directors of Fraunhofer AISEC, for providing me the opportunity and support to pursue the PhD. The manager of my department for secure operating systems, Sascha Wessel, who especially supported me during the early stages of the PhD at which it is difficult to gain momentum. To those colleagues at Fraunhofer AISEC and to all other researchers and students with whom I was fortunate to collaborate in both project and research works. With the obscure and guilty feeling of not mentioning all people who I am supposed to, I thus thank the following peers in alphabetic order: Gerd Brost, Paul England, Julian Horsch, Mathias Morbitzer, Marcus Peinado, Mykolai Protsenko, Kai Samelin, Benjamin Taubmann, Michael Velten, and Michael Weiß. Last but not least, my thank goes to my family and friends who are equally important, especially when it comes to sustain the balance between getting lost in details and the liberation of mind from bits and bytes.

Contents

CHAPTER 1

Introduction

The first part of the introduction motivates the importance of protecting confidential data on modern systems and highlights possible strategies against attackers. Based on that, we formulate a problem statement, which involves different challenges that arise when aiming to reliably protect data on modern systems and when seeking to extract confidential data. Subsequently, we address these challenges by briefly presenting the scientific publications and their contributions relevant for this thesis. The introduction ends with an outline describing the structure of the thesis, oriented along the elaborated challenges.

1.1 Motivation

During the past decades, more and more sensitive data has spread into the digital world. For example, individuals starting with setting up passwords for mail accounts and exchanging private mail contents have increased their digital footprint by moving more and more parts of their former non-digital identity onto their computers and the internet. The trend of the digitalization was initiated and continuously pushed forward by IT enterprises and innovative start-ups. This led not only individuals, but also companies and institutions of all kind to digitalize their infrastructures and workflows and to offer various types of services via the internet. The amount of sensitive data stored, processed and distributed on our nowadays diverse computing systems thus increased and digital resources of high value, such as banking credentials, intellectual property, or classified documents, were rapidly created. This data poses a highly valuable target for both sides of the law, forensic investigators and cybercriminals, where disclosure can lead to severe consequences for individuals, organizations and governmental bodies. With the quick innovation cycles of both software and hardware, the diverse types of devices on the market are capable of offering and consuming an abundance of services, but also exhibit their weaknesses. This embraces, for instance, smallest Internet of Things (IoT) devices used in industrial environments or in smart homes, mobile devices, notebooks and desktop computers used for business and private purposes, up to large-scale server and cloud systems of enterprises.

Despite efforts being made to increase system and network security, critical and characteristic vulnerabilities of devices are nevertheless frequently identified and exploited by criminals, launching attacks of different complexity. For example, smartphones pose attractive targets for attackers [Bec11; Fei15] even though many approaches to mitigate their susceptibilities were proposed [Alm14; Bac14; Chi11; Enc09; Ong09]. The over and over occurring security issues on their software layers make the devices vulnerable to a large number of remote attacks [Fel11; Pen14; Poe14; Zho12]. Various system architectures making use of additional or specifically designed hardware to improve system security

were proposed, for instance, in [Wag18; Wei16] for autonomous, embedded environments. This includes infrastructures for secure boot [Arb97], leveraging Trusted Platform Modules (TPMs) as hardware trust anchors for measured boot [Tru] up to the application layer [Sai04] for remote attestation [Eng03; Gar03], or Trusted Execution Environments (TEEs) for executing sensitive tasks in environments isolated from the rest of the system [ARM09; Cos16]. These architectures prevent attackers, for instance, from unnoticed overwrites of boot code and allow system designers to move specific tasks to the isolated TEE. This can be sufficient to protect confidential data on embedded systems fulfilling a very specific purpose where all sensitive data can be ensured to be relocated to the TEE. However, end user systems are usually more complex regarding the amount of sensitive data. This means that these architectures are not sufficient for protecting all the confidential data possibly found on a system. Also, proposed approaches for dynamic attestation [Dav09; Kil09], isolated execution of security-sensitive code combined with attested execution [McC10; McC08], or system monitoring combined with integrity verification [Vel17], all possibly building up on TPMs or on TEEs, do not primarily defend against the theft of confidential data. As a result, there is a clear need for system architectures designed to protect confidential data and to understand how attacks can be carried out.

Attackers pursuing the goal of data theft do not necessarily care about preserving certain imposed integrity conditions, but must be assumed to be primarily interested in finding any means to access sensitive data. While also TEEs and secure boot infrastructures were integrated onto mobile devices to increase system security, malicious applications installed with certain permissions have still shown to be capable of providing sensitive data to third parties. The same holds for other platforms. Despite being possibly equipped with a TPM for code integrity and remote attestation, the vast and diverse amounts of possibly confidential data on those devices are not safe from software-level attacks during the runtime. This is due to the incomplete isolation of possibly sensitive data from integral and possibly compromised parts of the system. While some sensitive data or code can be processed in a TEE, such as specific key material in the ARM TrustZone [ARM09] or in enclaves with Intel Software Guard Extensions (SGX) [Cos16; McK13], the systems cannot ensure to secure all the vast amounts of other possibly sensitive data from access. Exacerbating this problem, attacks vectors like Rowhammer [Kim14], or Spectre [Koc18] and Meltdown [Lip18b] have recently demonstrated that such software-level attacks are hard to fully defend against, even in cases where attackers have only minimal access to the system [Gru16; Lip18a; Tat18]. The nature of these so far unconventional attack vectors rendered traditional OS security mechanisms ineffective, as these exploited flaws in hardware, such as on Dynamic Random Access Memory (DRAM) modules or on modern CPUs, leading to unauthorized data access or privilege escalation. These are problems that need to be overcome in hardware design or microcode.

In this thesis we consider attackers that try to remotely exploit vulnerable code to gain access to confidential data as well as attackers aiming to gain access via physical interfaces. An important step to protect confidential data is thus the design of data-centric security architectures to isolate full execution environments on user-level with all their data from other environments possibly compromised by attackers. Such architectures are especially valuable when practically usable on a wide range of device types in different environments,

such as smartphones for end users in corporations or embedded devices in industrial facilities. A data-centric system securely isolating resources, resistant against software-layer threats of remote attackers is however only the first part for increasing overall system security. The system likely remains susceptible to physical attackers on hardware-layer. As a protection against physical attacks, many systems have security measures in place, such as tamper-proof hardware-locations for key material or Full Disk Encryption (FDE). When the system properly encrypts persistent storage, an attacker with physical access only to the storage medium must pay considerable effort to decrypt the data. However, there still remains considerable attack surface on sensitive data in main memory. The key used for FDE during the system's runtime is, like vast amounts of other possibly sensitive data, stored in main memory in plaintext [Apo13; Nta14; Pet07; Tan12]. Confidential data, where key material is only one example, accessed from main memory by a physical attacker could be reused in order to decrypt storage volumes, conduct transactions or to spoof an identity. Attacks yielding main memory access are called memory attacks and can, for example, be achieved by physical attackers via Direct Memory Access (DMA) [Ste13], Joint Test Action Group (JTAG) [Wei12], or cold boot [Gru13; Gut01; Hal09; Mül13]. For DMA attacks, the PCIe [Dev09], Firewire [Bec05; Boi06], or Thunderbolt [Maa12; Sev13] interfaces have shown to be exploitable on various platforms.

It is an essential aspect to investigate memory attacks, for example, cold boot attacks on mobile devices, to raise awareness for the threat and for the possibility of their forensic application. A likewise important goal is to develop mechanisms that counter such memory attacks and integrate them into secure system architectures. A way to address this is to not only encrypt the persistent storage, but also to encrypt the sensitive contents of main memory. It is in the next step essential to investigate the design of main memory encryption mechanisms for shortcomings to emphasize the requirement for their thorough design.

To summarize, while many other lines of work focus on architectures preventing attacks or on protecting specific secrets, this work focuses on preserving data confidentiality in the event of both remote and physical attacks taking place. Further, this work investigates methods for main memory extraction.

1.2 Problem Statement

We pointed out that the design of many common systems does not sufficiently protect the confidentiality of sensitive data against remote or physical attackers. This results in the need for improvements in system security by developing system architectures applicable in practice, and which address existing systems' shortcomings. There is also a need to investigate the attack vectors on data confidentiality, for forensic purposes, as well as for raising awareness and recognizing the design problems that need to be overcome. The goal of this work is to systematically find solutions to the challenges we list and briefly describe in the following.

Challenge 1: Design of Architectures for the Secure Isolation of System Resources
Remote or local attackers exploiting existing architectural shortcomings and software vulnerabilities must be prevented from accessing confidential data. An adversary gaining

root privileges by exploiting resources must be restricted to a defined part of the system, but not affect other, isolated environments. Sensitive data part of isolated environments then remains inaccessible to the attacker. To increase overall system security, the design of secure architectures should enable the protection of integral parts of a system and comply with embedded systems, mobile devices, but also desktop platforms. A particular challenge are mobile device platforms, which pose a high value target, are widely distributed, communicate over lots of interfaces and process vast amounts of sensitive data.

One of our goals is to realize a secure virtualization architecture for mobile devices to increase system security on mobile platforms. Not only mobile device platforms are widespread, but also platforms found in IoT ecosystems and other distributed embedded environments. Protecting such platforms is especially challenging, because the end user is often not part of the usage model and the systems operate mostly autonomously. For this reason, we also seek to realize a secure virtualization architecture for IoT platforms and usage scenarios.

Challenge 2: Applicability of Architectures for Secure Resource Isolation in Operational Environments

A secure architecture for mobile devices, or for IoT ecosystems, should not only result in a technical proof of concept prototype, but also be applicable in real-world scenarios. This requires the definition of an ecosystem in which the devices and their software are deployed. In turn, this requires concepts for deploying the virtualization architecture as a secure solution within productive infrastructures, especially when devices are located in untrusted environments. This includes overcoming the challenges of secure device provisioning and enrollment, secure software updates, as well as identity management for all entities in the ecosystem, such as a backend for remote management, the devices, their users, or operators. The specific field of application particularly influences the challenge and requires for different models. An example are devices without end users autonomously deployed in infrastructures, compared to smartphones being assigned to employees in an organization.

We address this challenge by introducing the relevant ecosystems and concepts for our secure virtualization architectures for both mobile devices for use in organizations, and for embedded devices for use in industrial IoT environments.

Challenge 3: Main Memory Extraction on Conventional Platforms

While the design of secure virtualization architectures defends against attacks on software layer, it does not remediate physical memory attack scenarios. Common hardware platforms ranging from embedded devices over mobile phones to traditional desktop devices keep their main memory in plaintext. Security mechanisms implemented in software may not be sufficient to protect the sensitive data in main memory when it comes to physical attacks circumventing the security mechanisms of the OS by directly accessing the main memory. This can be demonstrated with cold boot, JTAG, or DMA attacks, for instance. The challenge is thus to investigate the feasibility of such attacks on modern systems and to evaluate their usability for forensic frameworks. Especially mobile devices are a valuable target as they can easily be stolen or found to be unattended.

We aim to build a forensic framework for main memory extraction using the cold boot

attack. The framework enables the systematic extraction of sensitive system resources from main memory. With this framework, we urge the need to design effective countermeasures against physical attackers.

Challenge 4: Design of Architectures for Main Memory Encryption
The design of architectures for main memory encryption can enable an efficient defense against memory attacks. Main memory encryption does not prevent memory attacks themselves, but rather renders their effects futile. The architectures should not allow attackers to access any sensitive data in plaintext to sustain data confidentiality in case of a physical memory attack. This also requires secure key management for the main memory encryption key. The different usage models on the versatile systems must be considered as well. Smartphones operate in background and do not fully suspend compared to desktop computers and laptops where a full suspension is initiated. Some systems do not suspend at all. Main memory encryption must be accordingly designed to increase overall system security. For example, transparent encryption during a system's runtime causes performance overhead, whereas encryption during suspension protects only when the device suspends.

We address this challenge by introducing main memory encryption architectures for different device and usage types and by combining our secure virtualization architecture for mobile devices with main memory encryption to establish a system both resistant against software-layer and memory attacks.

Challenge 5: Main Memory Extraction on Platforms with Hardware-Based Memory Encryption
It is important to design memory encryption architectures with care, keeping in mind that a physical attacker may not only carry out memory attacks, but also gain privileges on the running system. An important aspect is to analyze existing solutions for architectural weaknesses. An example are virtualized server systems using hardware extensions for full-memory runtime encryption of their Virtual Machines (VMs). This relieves, for instance, customers from fully trusting their server providers, otherwise being able to access all the VM's memory in plaintext from the Hypervisor (HV).

We address this challenge by developing a framework that nevertheless allows to extract the full memory contents of the encrypted VMs in plaintext. Our framework is further capable of efficiently identifying and extracting specific resources, such as private and symmetric keys, with a minimal footprint left behind. This is even feasible in productive scenarios with the VM, such as a web server, being under high load. Our work raises the awareness for the limitations of runtime memory encryption platforms. Further, our work motivates the design of future architectures thwarting memory extraction vectors, an aspect we cover as well by proposing possible solution approaches.

1.3 Publications and Contributions

The following lists and briefly summarizes the scientific publications and their contributions with which we address the described challenges in the subsequent chapters of this work.

Publication 1: A Secure Architecture for Operating System-Level Virtualization on Mobile Devices

[Hub16a] Huber, Manuel et al.: 'A Secure Architecture for Operating System-Level Virtualization on Mobile Devices'. *Revised Selected Papers of the 11th International Conference on Information Security and Cryptology - Volume 9589*. Inscrypt 2015. Beijing, China: Springer-Verlag New York, Inc., 2016: pp. 430–450. isbn: 978-3-319-38897-7. doi: 10.1007/978-3-319-38898-4_25. url: http://dx.doi.org/10.1007/978-3-319-38898-4_25

With this work, we address Challenge 1. We designed a secure virtualization architecture, which isolates system resources to inhibit potentially compromised parts of the system from accessing other, protected resources. Our architecture builds on the OS-level virtualization solution from [Wes13]. The virtualization allows to instantiate different isolated execution environments, which we call *containers* and which share the same OS kernel. We systematically isolated and constrained the containers to restrict adversaries exploiting a container from accessing other containers' resources. The platform ensures that the entities exclusively communicate over well-defined and secured channels.

We realized the architecture with the implementation of a prototype for mobile devices on the ARM platform [Fraa]. We leveraged and combined the underlying Linux kernel's security mechanisms. Our prototype operates multiple virtualized Android containers where one container is running at the foreground and others in background. Our prototype and its architecture comprise a fully working implementation where we also solved the challenges of secure hardware resource virtualization and secure foreground-background container switching, always considering that one or more containers may be compromised. In our security evaluation, we showed how the architecture efficiently protects containers with confidential data from remote attackers trying to extract resources.

Contribution 1: A secure OS-level virtualization architecture for resource isolation.

Contribution 2: Implementation of the secure architecture for mobile devices running Android containers.

Publication 2: Improving Mobile Device Security with Operating System-level Virtualization

[Wes15] Wessel, Sascha et al.: 'Improving Mobile Device Security with Operating System-level Virtualization'. *Computers & Security*. Vol. 52. C. Oxford, UK: Elsevier Advanced Technology Publications, July 2015: pp. 207–220. doi: 10.1016/j.cose.2015.02.005. url: https://doi.org/10.1016/j.cose.2015.02.005

This work represents an extended version of the former conference paper [Wes13] under the same title. With this work, we address Challenge 2. The conference paper presented the OS-level based concept for realizing Android containers sharing the same OS kernel. We extended the conference paper with a holistic security concept to apply the mobile

device architecture in real-world ecosystems. We introduced a Secure Element (SE) for two factor authentication to the system architecture, which each user must unlock to start containers. Furthermore, we classified the different physical entities and software components, for example, the backend, users, containers, and devices, and systematically secured their relationships based on a Public Key Infrastructure (PKI). An example is to achieve the confidentiality, integrity and authenticity of containers with a concept for container software signatures and container storage encryption. We furthermore presented an applicable provisioning, enrollment and management process for devices and their assigned users.

Contribution 3: Security concept for the application of the secure virtualization architecture for mobile devices in operational end user ecosystems.[1]

Publication 3: An Ecosystem and IoT Device Architecture for Building Trust in the Industrial Data Space

[Bro18] Brost, Gerd S. et al.: 'An Ecosystem and IoT Device Architecture for Building Trust in the Industrial Data Space'. *Proceedings of the 4th ACM Workshop on Cyber-Physical System Security.* CPSS '18. Incheon, Republic of Korea: ACM, 2018: pp. 39–50. isbn: 978-1-4503-5755-5. doi: 10.1145/3198458.3198459. url: https://doi.org/10.1145/3198458.3198459

This work transfers the secure virtualization architecture and its application concepts previously designed for the mobile domain [Hub16a; Wes15] to embedded IoT platforms, and thus addresses Challenge 1 and Challenge 2. The work also focuses on a specific use case, the Industrial Data Space, a data exchange platform for organizations, where embedded devices gather and exchange data with other devices in a distributed network, for example, sensor or manufacturing data in Cyber-Physical Systems (CPS). We called these devices trusted connectors, which can be regarded as IoT devices part of an untrusted industrial ecosystem. We isolated the trusted connectors' different services which gather and process the data and which possibly originate from third parties. As the exchange of data is a central point in IoT ecosystems, we provided a secure protocol to establish mutual trust between connectors for the secure exchange of the confidential data between connectors likely dispersed over different organizations. As end users are not part of the usage model, we did not use a passphrase-based SE as trust anchor for the connectors, but instead used a TPM with its sealing and attestation functionalities. Addressing Challenge 2, the work also focuses on identity management for the whole ecosystem, on connector and service provisioning and their lifecycle to integrate the architecture into productive industrial environments.

Contribution 4: Implementation of the secure architecture for embedded platforms and application in industrial IoT environments.[2]

1 Joint work with Sascha Wessel who is the main contributor of the paper.
2 The first authors Gerd Brost and Manuel Huber contributed equally to the paper.

Publication 4: A Flexible Framework for Mobile Device Forensics Based on Cold Boot Attacks

[Hub16b] Huber, Manuel et al.: 'A Flexible Framework for Mobile Device Forensics Based on Cold Boot Attacks'. *EURASIP Journal on Information Security*. Vol. 2016. 1. New York, NY, United States: Hindawi Publishing Corp., Dec. 2016: 41:1–41:13. doi: 10.1186/s13635-016-0041-4. url: https://doi.org/10.1186/s13635-016-0041-4

In this work, we developed a framework for mobile device forensics based on the cold boot attack addressing Challenge 3. For the memory extraction, we reboot devices into the bootloader and start a minimalistic application instead of the Linux kernel on Android-based systems, for example. The minimalistic application fills no more than a single page in main memory mapped to a region where only constant and publicly known data of the previously running system resides. From the perspective of memory attackers but also forensic experts, the framework provides the possibility to systematically analyze and extract the full and unaltered state of the previously running system. Based on a serial connection, our minimalistic application takes memory requests from a remote forensic host device and returns the requested memory chunks via the serial interface.

We showed how analysts can, for instance, first retrieve the list of running processes and analyze the processes' memory mappings to quickly access the regions where sensitive material is located, such as username and password combinations for exchange accounts. We demonstrated the practicality of our framework with prototypes for two different devices, the Samsung Galaxy S4 and Nexus 5 phones. We also extracted sensitive key material of the containers from the secure virtualization architecture for mobile devices. With this, the framework emphasizes the urge for the protection of system resources against physical adversaries.

Contribution 5: A forensic framework for memory extraction on mobile devices based on the cold boot attack.[1]

Publication 5: Protecting Suspended Devices from Memory Attacks

[Hub17b] Huber, Manuel et al.: 'Protecting Suspended Devices from Memory Attacks'. *Proceedings of the 10th European Workshop on Systems Security*. EuroSec'17. Belgrade, Serbia: ACM, 2017: 10:1–10:6. isbn: 978-1-4503-4935-2. doi: 10.1145/3065913.3065914. url: http://doi.acm.org/10.1145/3065913.3065914

In this work, we developed a main memory encryption architecture for platforms which support full OS suspension and resumption to tackle Challenge 4. We primarily focused on the x86 architecture with Linux-based OSs, and aimed to protect the main memory of devices left unattended. During suspension, the architecture ensures the encryption of

1 Joint contribution with Benjamin Taubmann.

the full memory of all processes on the system and only decrypts when the user provides correct credentials upon resumption. This protects devices, such as laptops or desktop computers, from memory attacks once fully suspended.

We implemented our prototype for the Linux kernel and provided a stable and real-life applicable solution. Our design and implementation both leverage core concepts of OSs, such as cryptographic functionalities, the memory management infrastructure and process freezing capabilities. This makes the prototype easily deployable onto systems and can be combined with a TPM for the hardware-based protection of the encryption key, preventing brute-force attacks. The system automatically encrypts the memory when the device is suspended or left idle for a certain amount of time. In order to restore the system, the user has to provide the FDE passphrase. We also developed a concept to protect the full memory of guest OSs, like Windows, based on our Linux prototype by transferring our concepts into a Linux-based HV.

Contribution 6: A main memory encryption architecture for platforms with suspension features.

Publication 6: Freeze and Crypt: Linux Kernel Support for Main Memory Encryption

[Hub18] Huber, Manuel et al.: 'Freeze and Crypt: Linux Kernel Support for Main Memory Encryption'. *Computers & Security* (2018), vol. 86: pp. 420–436. issn: 0167-4048. doi: 10.1016/j.cose.2018.08.011. url: http://www.sciencedirect.com/science/article/pii/S0167404818310435

To further address Challenge 4, we presented a main memory encryption architecture for arbitrary process groups, for example, to protect VMs and container-based environments from memory attacks. The architecture ensures to encrypt the memory of process groups when freezing, i.e., suspending, a process group and ensures to decrypt the memory when thawing the group, i.e., when resuming it. While our design is platform-independent, we implemented a prototype for the ARM architecture on a Linux-based system. We combined the prototype with our secure virtualization platform to encrypt suspended Android containers. This results in a system protecting against both remote and physical adversaries. We encrypt idle or background containers with an SE-backed key that is only present during ongoing en- and decryption. Since encrypted containers are frozen and not responsive, we extended the virtualization architecture to handle and display encrypted containers' incoming events, such as phone calls or data packets, to preserve the usability of the system. To verify the effectivity of our memory encryption architecture, we acquired memory with our cold boot-based memory extraction framework from the devices. The extraction of sensitive data from suspended devices was no longer possible.

Contribution 7: A main memory encryption architecture on the granularity of process groups, suitable for encryption of containers.

Contribution 8: Combination of the memory encryption architecture for containers with the secure virtualization architecture for mobile devices.

Publication 7: TransCrypt: Transparent Main Memory Encryption Using a Minimal ARM Hypervisor

[Hor17] Horsch, Julian et al.: 'TransCrypt: Transparent Main Memory Encryption Using a Minimal ARM Hypervisor'. *Proceedings of the 16th International Conference on Trust, Security and Privacy in Computing and Communications.* TrustCom '17. Sydney, Australia: IEEE, Aug. 2017: pp. 408–417. DOI: 10.1109/Trustcom/BigDataSE/ICESS.2017.232

To address Challenge 4 with regard to platforms which do not necessarily suspend or which do not even support suspension, we proposed an architecture for transparent runtime memory encryption on the ARM platform. The architecture ensures the encryption of the full memory of a single guest OS with a minimal HV transparent to the guest. The HV, almost completely agnostic to the guest OS, leaves only a small amount of the currently utilized memory unencrypted, the so-called working set of pages. The ephemeral key resides in hardware-protected memory, which it never leaves, such as memory protected by the ARM TrustZone. This architecture allows making a choice on the trade-off between security and performance by keeping the amount of memory left unencrypted adaptable. We implemented and evaluated the prototype on an embedded board running Android and determined a plausible trade-off between security and performance.

Contribution 9: A runtime memory encryption architecture using a minimal hypervisor.[1]

Publication 8: SEVered: Subverting AMD's Virtual Machine Encryption

[Mor18] Morbitzer, Mathias et al.: 'SEVered: Subverting AMD's Virtual Machine Encryption'. *Proceedings of the 11th European Workshop on Systems Security.* EuroSec'18. Porto, Portugal: ACM, 2018: 1:1–1:6. ISBN: 978-1-4503-5652-7. DOI: 10.1145/3193111.3193112. URL: http://doi.acm.org/10.1145/3193111.3193112

AMD Secure Encrypted Virtualization (SEV) provides a hardware extension for main memory encryption for VMs. Mainly designed for VMs in (cloud) server systems, AMD SEV includes a protocol to ensure clients that their VM runs encrypted on the host system with SEV enabled. This protects the confidential data on guest VMs from a possibly malicious HV and from physical attackers. Addressing Challenge 5 with this work, we undermined this hardware security mechanism with a framework capable of extracting arbitrary encrypted main memory pages of the protected guest VMs in plaintext. Our framework makes it even possible to extract memory when the VM is under heavy load, possibly in productive environments where many remote peers simultaneously request resources. Our framework allows to fully extraction the memory of encrypted VMs running on the servers. We implemented a prototype which realized the attack on real hardware, AMD's EPYC processor. With the framework, we showed that the AMD SEV technology

1 Joint work with Julian Horsch who is the main contributor.

exposes design weaknesses exploitable in practice and pointed out the need to carefully design memory encryption schemes withstanding attackers with HV privileges on the system. We discussed several mitigation strategies to overcome the weaknesses.

Contribution 10: A main memory extraction framework for platforms with hardware-based runtime memory encryption.[1]

Publication 9: Extracting Secrets from Encrypted Virtual Machines

[Mor19] Morbitzer, Mathias et al.: 'Extracting Secrets from Encrypted Virtual Machines'. *Proceedings of the Ninth ACM on Conference on Data and Application Security and Privacy*. CODASPY '19. Richardson, Texas, USA: ACM, 2019: p. 10. ISBN: 978-1-4503-6099-9. DOI: 10.1145/3292006.3300022. URL: https://doi.org/10.1145/3292006.3300022

To further address Challenge 5, we extended the main memory extraction framework for hardware-based runtime encryption platforms with a method to specifically extract a VM's most valuable resources, such as Transport Layer Security (TLS), Secure Shell (SSH) or FDE keys. When targeting specific resources, the prior framework for memory extraction likely requires a considerable amount of time until the desired secrets are extracted and leaves a large footprint due to the many repeated requests for data. We extended the framework with the capability to extract targeted resources while requiring minimal interaction with the target system and leaving only a minimal footprint. By observing the page access types and timings of the guest VM in the HV, we recognized the relevant memory pages and specifically extracted them. An example is to observe a single TLS handshake from the outside while recording the VM's page accesses, and to then extract a specific subset of the pages the VM accessed during the recording phase. This extension of the framework makes it possible to extract sensitive key material in less than a minute with a drastically reduced amount of requests to the VM, making the attack inconspicuous and especially feasible in productive server environments.

Contribution 11: A method for the efficient extraction of targeted secrets from platforms with hardware-based memory encryption.[2]

Publication 10: A Lightweight Framework for Cold Boot Based Forensics on Mobile Devices

[Tau15] Taubmann, Benjamin et al.: 'A Lightweight Framework for Cold Boot Based Forensics on Mobile Devices'. *Proceedings of the 2015 10th International Conference on Availability, Reliability and Security*. ARES '15. Washington, DC, USA: IEEE Computer Society, 2015: pp. 120–128. ISBN: 978-1-4673-6590-1. DOI: 10.1109/ARES.2015.47. URL: https://doi.org/10.1109/ARES.2015.47

1 Joint work with Mathias Morbitzer who is the main contributor.
2 The first authors Mathias Morbitzer and Manuel Huber contributed equally to the paper.

This is the conference paper version of the journal paper on our cold boot framework [Hub16b]. The conference paper covers a subset of the contents from the previously described journal paper.

Publication 11: Freeze & Crypt: Linux Kernel Support for Main Memory Encryption

[Hub17a] Huber, Manuel et al.: 'Freeze & Crypt: Linux Kernel Support for Main Memory Encryption'. *14th International Conference on Security and Cryptography*. SECRYPT 2017. Madrid, Spain: ScitePress, 2017: pp. 17–30. isbn: 978-989-758-259-2. doi: 10.5220/0006378400170030

This is the conference paper version of the journal paper on our main memory encryption architecture for process groups [Hub18]. The conference paper covers a subset of the contents from the previously described journal paper.

Publication 12: Dominance as a New Trusted Computing Primitive for the Internet of Things

[Xu19] Xu, Meng et al.: 'Dominance as a New Trusted Computing Primitive for the Internet of Things'. *Proceedings of the 2019 IEEE Symposium on Security and Privacy*. SP '19. San Francisco, CA: IEEE Computer Society, 2019

This paper defines a new trusted computing primitive, called dominance, for the internet of things. The central goals of dominance are device recoverability and availability. Dominance allows a remote administrator to recover compromised or malfunctioning devices within a bounded amount of time. Dominance even enables the eviction of malware on devices, which runs on the highest privilege level and which actively refuses updates. This is of special importance in settings where a large number of devices is spatially dispersed. While manual repair can lead to a time-consuming and expensive recovery process, an automatted remote recovery can effectively reduce cost, time and bring otherwise bricked devices back to their original mission.

We decomposed dominance into two simpler primitives, the *gated boot* and the *reset trigger* primitives. Gated boot ensures that only the software that is authorized by the administrator is ever booted on the device. In order to protect the code and data of gated boot, we introduced so-called *latches*. Latches are either in locked or unlocked state. Once locked, latches effectively fulfill a security function until the next reset of the device. In the simplest case, the functionality of a latch is to prevent accesses to certain memory range until the next reset. Once gated boot passes control to untrusted software, it latches its memory regions.

The reset trigger primitive enforces that the device can always be reset by its administrator within a bounded amount of time. A reset trigger can be realized with an Authenticated Watchdog Timer (AWT). An AWT resets the device after a certain amount of time unless the administrator issues the AWT an authenticated ticket, which extends the time until reset. The AWT is isolated from possibly malicious software at runtime, for instance, as a separate hardware implementation, or implemented in TrustZone. We implemented

dominance on three popular IoT devices, on the i.MX6 HummingBoard, the Raspberry Pi 3, and an STM32L4 microcontroller, ranging from high to low end devices. In our evaluation, we showed that the overhead of our prototypes is negligible and that our concept can be realized in practice.

Publication 13: Cryptographically Enforced Four-Eyes Principle

[Bil16] Bilzhause, Arne et al.: 'Cryptographically Enforced Four-Eyes Principle'. *11th International Conference on Availability, Reliability and Security (ARES)*. IEEE, Aug. 2016: pp. 760–767. DOI: 10.1109/ARES.2016.28

In this work, we introduced a formal framework, which enforces the four eyes principle (4EP), a control and authorization principle to minimize the likelihood of forging bogus data in networks. Our framework comprises cryptographic security definitions capturing the main idea of the 4EP and uses Sanitizable Signature Schemes (SSS) to design a provably secure construction meeting the requirements of the 4EP.

1.4 Outline

This thesis is organized as follows. We first introduce a system architecture and attacker model in Chapter 2 to give an overview on the general system layout serving as the basis for the subsequent chapters, as well as to provide a unified understanding of the attacker and of the problems to be solved. Afterwards, we focus on the previously described challenges in Chapter 3 to Chapter 6. Each of those chapters introduces the specific challenge and contributions in more depth and provides an overview on related work. Where necessary, the chapters additionally provide a brief background.

Chapter 3 presents our OS-level virtualization-based architecture for the isolation of system resources to tackle Challenge 1. We first realize the architecture for mobile devices and describe its integration into productive end user environments, providing a solution for Challenge 2. Based on that, we also realize an architecture for embedded use cases without end users and describe its integration into untrusted environments in the industrial IoT.

We present our main memory extraction framework based on the cold boot attack, referring to Challenge 3, in Chapter 4.

Chapter 5 focuses on Challenge 4, the architectural design to defend against memory attacks. We start with a main memory encryption architecture for devices, which support suspension, such as notebooks and desktop computers. Then, we introduce our main memory encryption architecture for process groups and combine it with the secure virtualization architecture for mobile devices to protect suspended Android containers. This results in an architecture for main memory encryption for systems which are not designed to fully suspend, such as smartphones processing background events during phases of inactivity. We also develop a runtime memory encryption architecture that additionally supports systems not suspending.

In Chapter 6, we cover Challenge 5, the extraction of main memory from hardware-based runtime memory encryption platforms. We first present the framework we designed for the extraction of full memory dumps from encrypted VMs. We extend that framework by

introducing methods to directly extract specific assets with a reduced footprint instead of acquiring full memory dumps.

We finally draw our conclusions and point out future research directions in Chapter 7.

CHAPTER 2

System Architecture and Attacker Model

In the following, we introduce a generic system architecture and attacker model as basis for the remainder of the thesis. Based on that, we also sketch existing defense mechanisms and briefly relate the defense mechanisms we propose in this thesis to the attacker model.

2.1 System Architecture

With the introduction of a generic system architecture, we provide a unified view on a computing device's hardware components and software stack used as common basis for the later chapters. Furthermore, this helps relate the attack vectors resulting from the attacker model we introduce to the architecture.

Our system architecture considers computing devices that are both remotely and physically accessible to attackers. These are widely used devices based on the prevalent architectures, such as x86- or ARM-based devices. We focus on devices with sufficient storage and memory, such as on Cortex-A processors, that can host a complex execution environment, for instance, Linux-based OSs. We differentiate between devices designed for end users, i.e., smartphones, tablets, desktop computers, or notebooks, and devices applied in environments without end users. The latter can be devices deployed in an industrial setting, such as gateway devices connecting enterprises and facilities to the internet of things. We assume that the devices store and process confidential data that needs to be protected from adversaries. These data can be manufacturing plans, cryptographic key material, sensitive documents, or user credentials, just to name a few.

Despite the devices possibly being used in different ways and environments, we abstract with our system architecture from particular usage characteristics or peripherals. We assume that devices typically interact with their users or communicate with other devices. For this purpose, the devices expose interfaces for the communication to the outside, either for direct interaction with users, for remote communication within a network, or to communicate with other devices in the proximity. These interfaces may be Wi-Fi, USB, LAN, bluetooth, or radio interfaces, for instance. This not only makes the devices remotely available and possibly connect to untrusted networks, such as the internet, but also makes them physically attachable.

Figure 2.1 illustrates an abstract hard- and software stack for our system architecture. The software layer consists of a complex OS with middleware and different applications on top. This may, for instance, be an Android OS with several apps running, or a Linux kernel with several user space functionalities. The Trusted Computing Base (TCB) typically involves the layers highlighted in gray colors. Secrets relevant for attackers can be processed and stored on any layer of the software stack, such as by user space processes or in the form

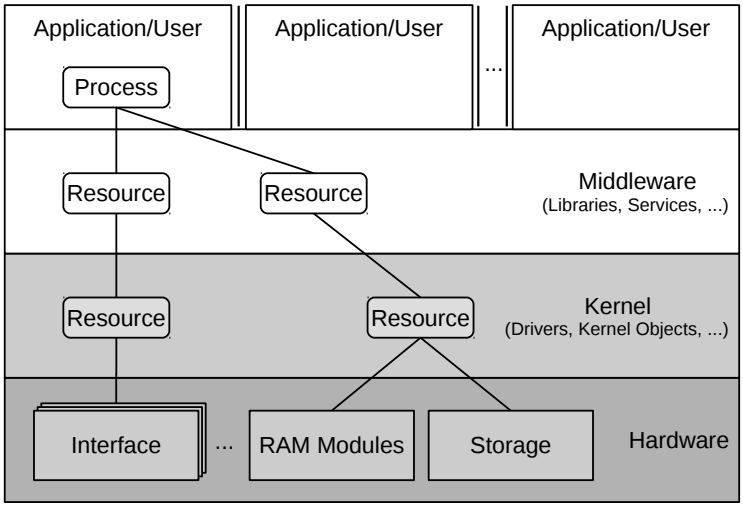


Figure 2.1: Abstract view on the hard- and software stack of our system serving as basis for the proposed security architectures and memory attacks.

of kernel objects. From the hardware-perspective, Figure 2.1 illustrates various generic interfaces of a computing device, as well as RAM modules and persistent storage, where we assume that the sensitive secrets are stored. In case of RAM modules, the stored data is volatile, but present as long as the system is powered and as long as data is not specifically erased or overwritten. For persistent storage, we assume for the remainder of the work that storage volumes with confidential data are properly secured with state-of-the-art FDE. This makes an attacker with physical access to the storage drive unable to read out the plaintext contents of the storage volume, unless not in possession of the FDE key. Secrets may also be temporarily present in other locations, such as registers or caches, which are aspects this thesis does not cover.

Figure 2.2 provides an abstract, hardware-centric view of our system architecture, showing a device with several interfaces and hardware peripherals. With the illustration, we put the focus on sensitive data possibly stored in the RAM modules. Data in main memory can be accessed via software running on the main CPU, or via firmware running on peripheral components with access to the system bus, such as the Graphics Processing Unit (GPU). Peripheral components can also represent the interfaces to the outside, such as a Wi-Fi chip, PCIe controller or a JTAG/UART interface, as shown in in Figure 2.2. Depending on the type of interface and on the control the attacker has over the interface, the interfaces can either allow attackers direct access to main memory, or allow to interact with the OS and its user space components. Note that we refine this generic system architecture in subsequent chapters where necessary and that we assume that the OS encrypts persistent storage.

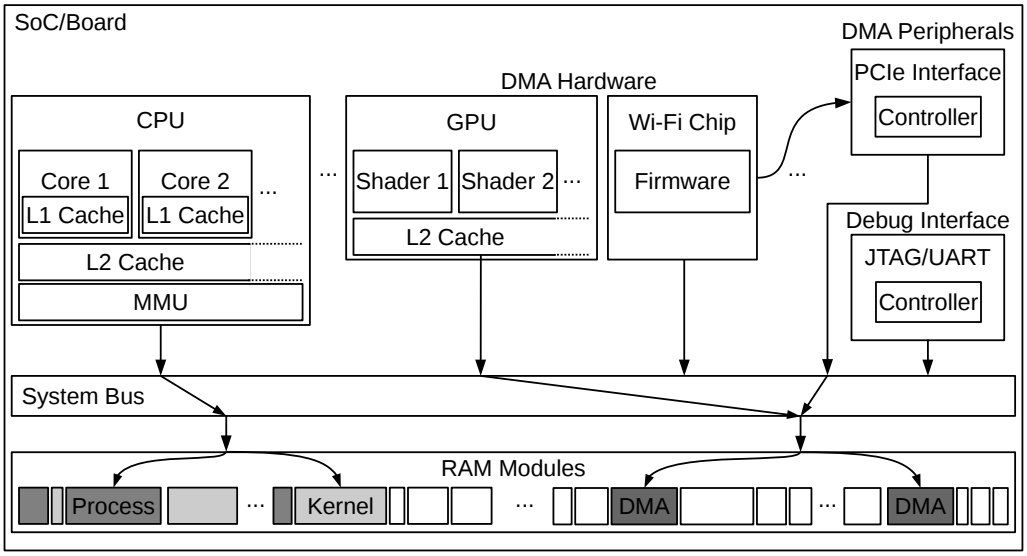


Figure 2.2: Hardware-centric view of our system architecture.

2.2 Attacker Model

This thesis focuses on the protection of the confidentiality of data stored on computing devices. We thus do not seek to defend against Denial of Service (DoS) attacks, and not explicitly against attacks on control flow integrity, for example. Note that to reach the goal of obtaining confidential data, an attacker might violate control flow integrity, though. The protection of confidential data is retained as long as an integrity violation does not yield the attacker access to confidential data.

In turn, this means that the goal of the attacker is to read out confidential data in plaintext from main memory or persistent storage on our devices. Confidential data may, for example, entail cryptographic key material, documents, credentials, or images. The disclosure of confidential data has an adverse impact across the boundaries of the system. From this point on, we consider confidential data to form a subset of a system's *sensitive* data. Sensitive data refers to segments of main memory which are, for instance, crucial to the security of the system itself. An example is data that needs to be protected to preserve the integrity of the OS, such as kernel structures.

In Section 2.2.1, we define the different attacker types relevant for the thesis. We describe resulting attack scenarios in Section 2.2.2.

2.2.1 Attacker Types

In our attacker model, we differentiate between three attacker types aiming at confidential data, each of which has different capabilities.

Physical Attacker. This type of attacker has physical access to the full system. The attacker can launch cold boot and DMA attacks and access all the physical interfaces the system exposes, such as debug interfaces, or interfaces for peripheral hardware. Furthermore, the attacker can dismount removable components. Our assumption

that persistent storage is properly encrypted prevents physical attackers from directly reading out all contents on persistent storage. In order to violate data confidentiality, the attacker thus targets the contents of main memory.

Local Attacker. The local attacker has local privileges on the running system, but is due to common access control mechanisms restricted from accessing the desired confidential data. This means that the attacker can execute code and access data within the privilege boundary enforced by the OS. The attacker can try to escalate privileges by exploiting software flaws, i.e., to execute runtime attacks on the TCB. Only when this succeeds, the attacker can read out confidential memory according to the privileges achieved. The local attacker may not only act from a remote position, but may in addition be *physically present*. However, the attacker can only benefit from the physical presence by making use of the common interfaces for end users, such as the touchscreen, keyboard strokes, or common USB interface functionality. This restriction separates the definition of the local attacker from the physical attacker's.

Remote Attacker. In contrast to the local attacker, the remote attacker starts without any privileges on a running, non-compromised system. Like the local attacker, the remote attacker may also be physically present. The remote attacker can try to gain privileges with remote exploits targeting the different layers of the system's software stack. The attacker can communicate with the system via its communication interfaces, for example, via remote or LAN connections, or via protocols over a radio, Wi-Fi, or bluetooth hardware interface. When the attacker gains control over an application on user-level, for example, because it incorrectly processes incoming data packets, this results in local privileges providing the capabilities of a local attacker. Note that the attacker may not only try to compromise the runtime environment on the application processor, but also send malicious data to peripheral devices in order to obtain access to confidential data. These may have their own runtime environment, such as firmware on a Wi-Fi chip, and have access to main memory.

We assume the following capabilities to be out of scope for all described attacker types:

Breaking cryptographic primitives. The attacker is not able to break properly applied state-of-the-art cryptographic primitives. This capability is reserved to parties with access to extraordinary computational resources, such as governmental bodies, or parties with undisclosed knowledge about cryptographic flaws and backdoors.

Physical lab and complex side-channel attacks. The attacker is unable to execute sophisticated physical lab and side-channel attacks on hardware elements. This exclusion comprises other physical attack vectors than those described. We thus exclude, for example, cache attacks, fault attacks, microprobing attacks, as well as complex software- or hardware-based side-channel attacks like attacks exploiting speculative execution or power consumption and timing. It is important to provide countermeasures against these attacks acting on the specifics of the hardware layer and design. There exists a vast body of work in this topic, which we leave out of scope and which may complement the mechanisms we propose.

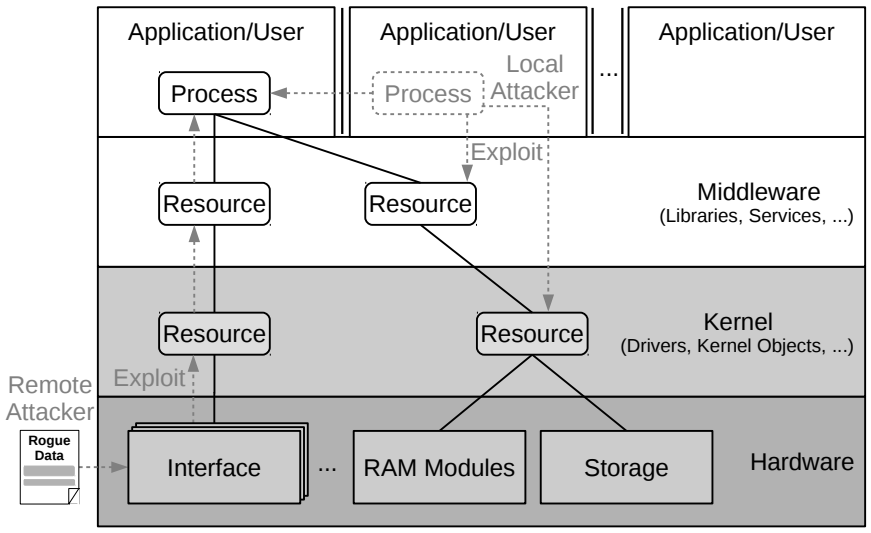


Figure 2.3: Possible exploitation paths for acquiring privileges as a local or remote attacker.

Together with having storage properly encrypted, this implies that the attacker can target confidential data only in main memory. The attacker can access the data either via a physical channel or in software with sufficient privileges. This also implies that we may assume that hardware protection mechanisms securely store confidential data. An example is data in caches, on-chip memory, such as in iRAM, or data protected with the ARM TrustZone, or key material located in an SE.

2.2.2 Attack Scenarios

In the following, we present the different attack scenarios on confidential data on the basis of the capabilities and limitations of the introduced attacker types. We also provide illustrations with different views on our system architecture to emphasize the shortcomings of common systems.

2.2.2.1 Scenario 1: Software Exploits

Figure 2.3 comes back to the software-centric view on a device with a user-level, or application-layer, a middleware, a kernel and an abstract hardware layer. User applications run on the topmost layer and usually have least privileges. This is the starting point of a local attacker trying to exploit other processes or layers. The local attacker, as depicted on the top middle layer, can try to exploit other application-layer processes, the middleware or the kernel. The interaction with other layers involves their typical resources, for example, libraries, services, drivers, or kernel objects. The interaction can be made via system or library calls, Inter-Process Communication (IPC) or or kernel interfaces, for instance. The middleware layer consists of usually higher privileged system services or library functionalities, which, for instance, facilitate the communication with the kernel or with external entities. Control over the kernel layer entails the full set of privileges. Depending on the targeted data, gaining control over another user space process may be

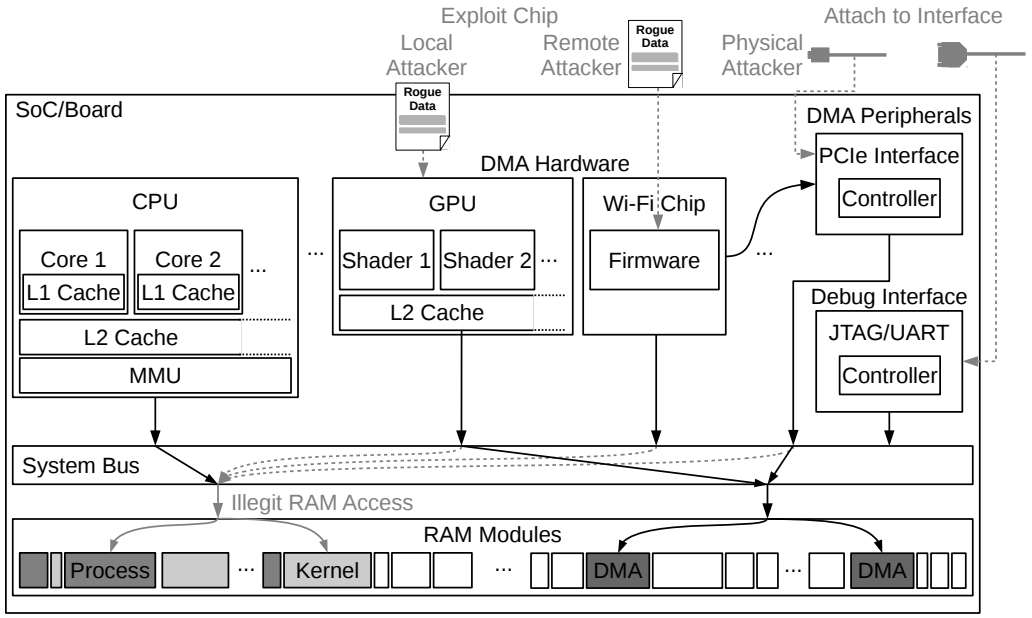


Figure 2.4: Scenarios for memory attacks circumventing OS and CPU protection mechanisms.

sufficient, or require a kernel exploit.

Communication from outside via a hardware interface usually involves the kernel including drivers and the middleware layer. In case an application on user-level communicates, data and code flow first traverses these layers, as depicted in Figure 2.3. A remote attacker, bottom left, can thus try send rogue data to exploit one of the involved software layers over any available interface. We consider software exploits on the runtime environment of the system running on the application processor as *indirect* attacks on memory. The attacker first acquires a higher set of privileges and then uses the acquired privileges to access memory. This means that the local or remote attackers first have to bypass certain security mechanisms of the OS to access the targeted data.

Figure 2.3 omits direct attacks on memory via software exploits, because it focuses on the scenario of exploiting the runtime environment on the application processor. In general, a remote attacker can also try to exploit peripheral firmware on the lowest layer, for example, to reach code execution on a Wi-Fi chip when it processes incoming data. We consider these *direct* attacks on memory, which we cover in the following two scenarios.

2.2.2.2 Scenario 2: DMA Attack

A DMA attack refers to the scenario where an attacker makes use of DMA-capable peripheral devices to directly access the system's memory without involving the CPU. This path circumvents the security mechanisms of the OS, not requiring to gain privileges on the running OS. This scenario mainly refers to physical attackers, but with remote exploits for externally communicating DMA devices, DMA attacks can also be carried out by remote and local attackers. Figure 2.4 illustrates this setting, showing attack scenarios on DMA

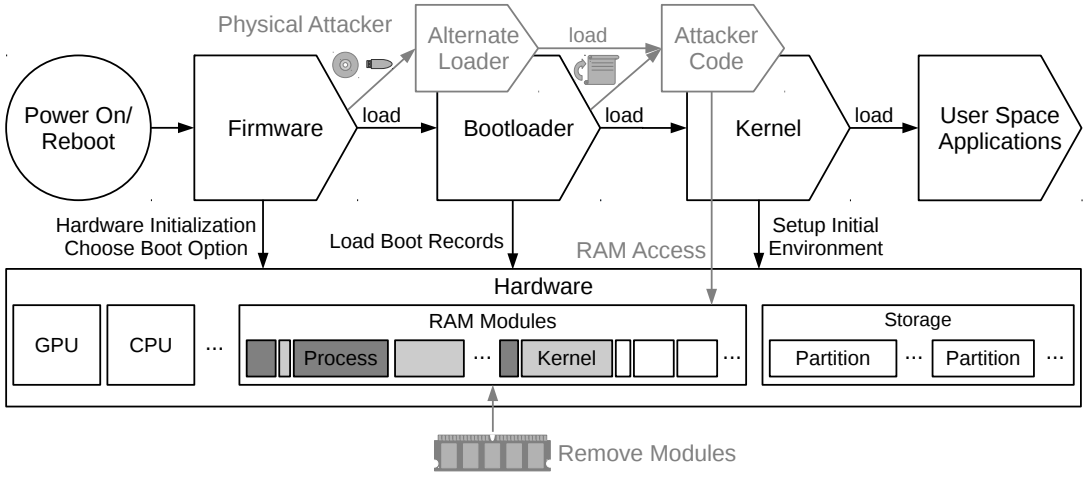


Figure 2.5: Boot procedure of a system and memory attacks via different cold boot paths.

devices and peripherals. These devices are, like the main application processor, connected with the system bus and in turn to the RAM modules. One attack path, illustrated on the top right hand side, is to attach to a physical interface that is DMA-capable, for instance, to the PCIe interface. When attached, some systems allow direct access to arbitrary main memory, such as kernel or process memory, instead of exclusively to DMA memory. This direct access is illustrated with gray-dashed arrows.

As mentioned, a DMA attack can also be realized by remote and local attackers. A remote attacker can, for instance, try to exploit the firmware of the device's Wi-Fi [Embb; Exo] or bluetooth peripheral [Mica] by sending rogue data. The attacker can in turn use the peripheral's DMA capability for direct access to main memory. A local attacker can try misuse or exploit DMA devices accessible within the privilege boundary, such as the GPU, to access arbitrary memory.

Figure 2.4 also illustrates the case where a physical attacker might attach to debug interfaces other than DMA peripherals. The JTAG debug interface is a feature that the system's CPU may have enabled and may be exploited when the interface is accessible from outside. This also provides direct access to main memory, but is not a DMA attack, as the CPU is involved. Another non-DMA means to get access to main memory is the Universal Asynchronous Receiver Transmitter (UART) debug interface. However, this interface needs to be configured by kernel, which is also involved in UART communication. Note that attackers can also launch bus monitoring attacks [EPN; Fut] to get access to main memory contents.

2.2.2.3 Scenario 3: Cold Boot Attack

Another path to directly access the main memory of a device is the cold boot attack. Compared to the other two scenarios, this scenario only applies for physical attackers. Like in the DMA attack case, the attacker does not need to acquire privileges on the running system. Figure 2.5 illustrates the cold boot attack paths with a view on a booting

device. On systems where RAM modules are not soldered, an attacker can remove the main memory module to read out all the module's contents on an attacker-controlled device, depicted on the bottom center part. Memory contents fade over a function of time and temperature while non-powered, resulting in loss of information upon removal [Gut01]. On systems with RAM modules soldered on-chip, removal is however not possible.

The boot-time perspective on a device in Figure 2.5 emphasizes that a cold boot attack can also be executed during the boot process of the system. When causing a reboot of the system, an attacker has a fair chance that the contents of main memory do not fade at all, as the memory may be continuously supplied with power. While only the bootloader and initial firmware, or ROM code, is executed, most of the main memory contents of the previously running system still exist and are likely not to be overwritten before the kernel and user space processes start. This is why the attacker who does not remove RAM modules typically tries to gain control over the system before the OS kernel is loaded. On some systems, the first point of user interaction is even before the firmware starts the boot loader. After basic hardware initialization, the initial firmware chooses the boot option in that case, depending on which devices are present. This gives the attacker the opportunity to boot from another device and to directly load attacker-controlled code. In case the firmware loads a fixed bootloader, the attacker can still try to influence the system to make the bootloader not load the common OS kernel, but instead attacker-controlled code. We show how this can be carried out on mobile devices in Chapter 4.

2.2.2.4 Outlook on Defenses

Having described the system model and attacker, we give a brief overview on state of the art mitigations for the presented attack scenarios and outline how we defend systems against the introduced attacker with our contributions.

Software Exploitation

A huge variety of defense mechanisms against software exploits exists in both practice and academia, implemented in different layers of the software stack, such as compiler-based or binary load-time approaches enforced by the OS. The enforcement of these mechanisms often depends on the platform the software stack is deployed onto and on the particular OS, such as stack canaries, ASLR, DEP enforcement, or control flow integrity mechanisms. The bottom line is that these mechanisms are often circumvented by attackers, which we discuss in more depth in Chapter 3. We assume that state of the art mechanisms are present in the systems we consider.

A possibility to increase the security of systems is to isolate attackers based on OS-level virtualization. OS-level virtualization allows to run multiple userland instances of an OS in parallel on a single, shared kernel and can thus be regarded as an OS kernel functionality. We pursue this approach in Chapter 3 in combination with system hardening to develop secure OS-level virtualization-based architectures. Figure 2.6 briefly sketches our approach and emphasizes the isolation of attackers with the bold elements. This makes it more difficult for local and remote attackers to exploit the hardened OS kernel, and in case the attacker acquires control over applications or over the middleware, the attacker still remains isolated in the respective userland instance without access to other instances that

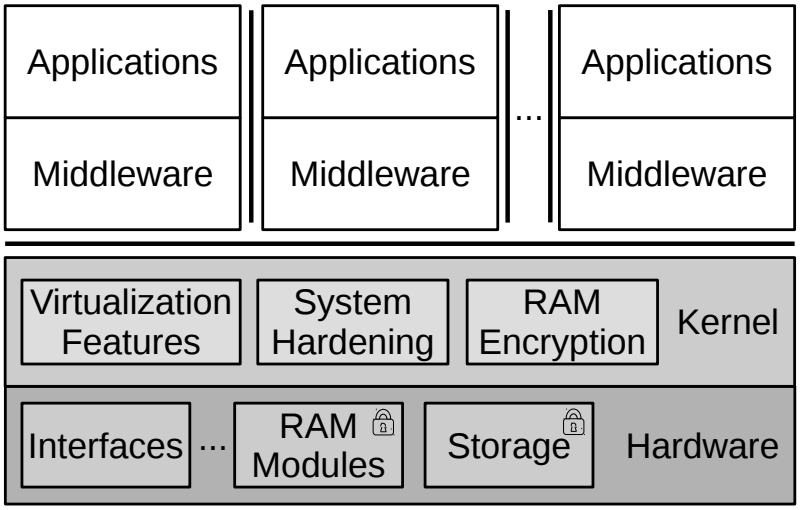


Figure 2.6: System architecture providing resilience against runtime attacks and direct attacks on memory.

may contain the targeted confidential data.

DMA and Cold Boot Attacks

An effective countermeasure against DMA attacks can be to attach an I/O Memory Management Unit (IOMMU), also known as system Memory Management Unit (MMU), to DMA-capable devices. The IOMMU represents an MMU for I/O devices. The CPU can configure the IOMMU upon system start to restrict a DMA device or interface to only access specific DMA memory ranges. However, it has been shown that devices have frequently been incorrectly configured or are missing IOMMU protection.

A defense to overcome cold boot attacks is on the one hand to solder RAM modules onto the CPU package, making them irremovable. Another way to mitigate the effect of cold boot attacks is, besides ensuring to load only authenticated software, to clear contents of main memory in the initial firmware, or in the bootloader. Despite these possibilities, modern systems are still vulnerable to the cold boot attack, which we show in Chapter 4.

Figure 2.6 also shows our approach to mitigate DMA and cold boot attacks, namely with RAM encryption on kernel-level, which we present in Chapter 5. Note that our architecture makes DMA attacks futile only in case of a *reading* memory attacker. A *writing* memory attacker interacting with a running system may be capable of modifying the OS and is especially hard to defend against. We address this point in our security discussions in our chapter about main memory encryption architectures. Even hardware architectures transparently encrypting main memory to prevent direct attacks on memory can only provide limited protection, which we emphasize in Chapter 6.

The presented attacker and system model lays the foundation for our security architectures and main memory extraction frameworks we explore in the following. We refine the attacker model according to the specific system setting and use case where necessary.

CHAPTER 3

OS-Level Virtualization Architectures for Secure Resource Isolation

This chapter covers our contributions for Challenges 1 and 2. These are, first, the design of architectures for the secure isolation of system resources and, second, the application of such architectures in productive environments. We first focus on Challenge 1, where we start with the design of an OS-level virtualization architecture for the secure isolation of system resources on mobile devices. The architecture protects the confidentiality of data by separating the system into different, isolated containers. This prevents possibly malicious containers from accessing data of other containers with sensitive contents. With the implementation of this architecture for containers running the Android OS on ARM-based mobile devices, this part reflects Contributions 1 and 2, which we describe in Section 3.3.

An implementation of prototypes for the virtualization architecture alone is not sufficient for its application in real-life environments. In particular, its applicability requires secure management of the whole device lifecycle, the topic reflected by Challenge 2. These environments require, for instance, concepts for the secure provisioning and enrollment of devices for end users, identity management, or the attachment to a backend for remote device management, especially for software updates. In Section 3.4, we therefore present Contribution 3. We develop a holistic security concept for the application of the secure virtualization architecture on mobile devices in productive end user environments. An example for a use case is a corporation providing employees smartphones with a container for business use and thus with sensitive contents, and a container for private use. For instance, this makes the possession of a separate business device with limited functionality, to keep business data secure, obsolete.

While Section 3.3 focuses on the design of the architecture for mobile devices where mechanisms for secure hardware device virtualization and user-based secure container switching are required, the isolating primitives of the architecture themselves are not tied to the mobile domain. The architecture design can also be transformed for use on traditional devices, or on embedded systems without end users, and be applied in respective other ecosystems, such as in industrial IoT ecosystems. We pursue this with Contribution 4. For that purpose, we transform the architecture design for use on environments without user interaction and implement it for different embedded hardware platforms in Section 3.5. Furthermore, we develop a security concept for managing the lifecycle of devices in IoT ecosystems running our virtualization architecture. This results in a *trust ecosystem* for the internet of things where devices are interconnected and form a distributed network for securely exchanging confidential data. This complements our elaborations from Section 3.4, which focus on a secure ecosystem with end users.

With these contributions, we increase the protection of systems from the local and remote attacker introduced in Chapter 2 and extend the therein introduced generic system architecture.

An overview on related work in Section 3.1, a brief background on Linux kernel mechanisms for OS-level virtualization in Section 3.2, and a subsequent summary in Section 3.6 complement our work on the resource-isolating security architectures in Section 3.3 to 3.5.

The following contributions can also be found in the publications [Bro18; Hub16a; Wes15]. Contributions 1 and 2 resulted from joint work with the co-authors Sascha Wessel, Michael Weiss, Julian Horsch, and Michael Velten [Hub16a; Wes15] and stem from previous work on OS-level virtualization [Wes13]. Contribution 3, part of the contents found in [Wes15], was co-developed with Sascha Wessel. In addition to the mentioned authors, Contribution 4 resulted from joint work with Mykolai Protsenko, Gerd Brost and Julian Schütte. All authors collaborated in the development of the concepts and implementations.

3.1 Related Work

The necessity for exploring novel approaches to protect data confidentiality results from the numerous attacks on today's interconnected devices, especially on widespread platforms like Android [Fel11; Pen14; Poe14; Zho12]. Another threat for confidentiality pose applications that actively leak sensitive data [Enc10]. Various approaches for security enhancements on Android have been presented, such as in [Alm14; Bac14; Chi11; Enc09; Ong09]. These approaches focus on the middleware layer to, for example, reach fine-grained control over the OS permission system [Bac14; Ong09], to restrict applications from OS resources [Chi11; Enc09], or to harden the OS [Alm14]. Introducing security mechanisms on the large middleware layer often results in a highly complex system, in a large TCB and in a highly OS-specific solution, not easily portable and dependent on specific versions of the OS. Virtualization techniques enable to create isolated environments and can help overcome these problems not only on mobile devices, but on many platforms. In the following, we discuss related work on different virtualization techniques, whereby we focus on OS-level virtualization.

Virtualization Techniques

We discuss three different approaches to virtualization, namely *user level isolation*, *system-* and *OS-level virtualization*. User-level isolation [Bug11; Bug13; Rus12] is an approach to create separate environments through isolating applications on the framework level. A successful attack on privileged processes still results in gaining full control over the system.

System virtualization [Dal14; Hwa08; Ros12] is a wide spread mechanism and provides full OS virtualization including the kernel, depicted on the right side of Figure 3.1. An advantage of system virtualization is that it can, depending on the use case, be realized with a small HV and thus result in a system with small TCB. However, the approach is strongly hardware dependent, because drivers have to be reimplemented for all hardware devices. Currently used open-source implementations for embedded systems are, for example, KVM [Dal14] or Xen [Bar03; Hwa08]. Even though current embedded CPUs more and more support hardware-assisted system virtualization, the performance overhead compared to native execution could be a crucial issue in embedded systems. Another

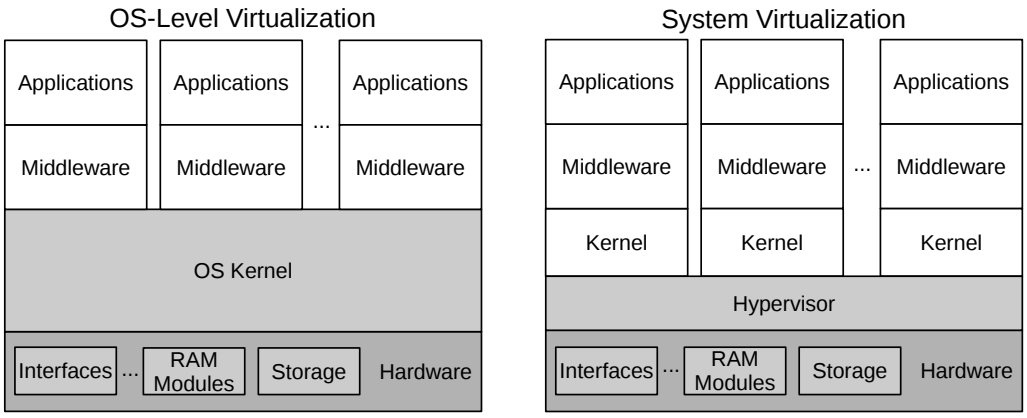


Figure 3.1: Overview on the software layers of a system with OS-level virtualization (left) and with system virtualization (right).

issue of full system virtualization is the overall system boot time, for instance, for IoT devices when frequently switching between power on/off states and booting the system from scratch.

OS-level virtualization, or container-based virtualization, aims to solve the shortcomings of user-level isolation and system virtualization. Depicted on the left side of Figure 3.1, OS-level virtualization separates userland OS instances running on a single modified kernel, which is responsible for resource virtualization. When properly virtualized, an attacker needs to compromise the kernel to break out of a container. Achieved with LXC, Jails [Kam00], Docker [Mer14], OpenVZ, or Linux-VServer, the technique is established on x86 and considered efficient [Res14; Xav13]. For Docker containers, Felter et al. [Fel15] showed that the performance overhead is negligible compared to native execution. Secondly, containers boot much faster than virtual machines [Seo14], since they do not have to run a whole new kernel for each virtualized environment. That is why we decided to design a container-based architecture instead of using full system virtualization. Due to the bootstrapping and performance advantage, container-based virtualization found wide application for server virtualization in cloud-based use cases [Ber14; Ger14; Pah15; Pei16]. Lately, Kaur et al. [Kau17] proposed an architecture called *Container-as-a-Service* which utilizes container-based virtualization as efficient approach for nano data centers.

In contrast to solutions like Docker, our virtualization approach specifically aims to reduce and modularize the TCB as well as the complexity of the overall virtualization infrastructure. In our architecture, we move critical functionality not necessarily required in the privileged virtualization layer to a separate container. Despite that this separate container is more privileged than other containers, the container is isolated from the virtualization layer and may only interact with it via a well-defined channel. Further, our proposed architecture provides a holistic security concept from software signing, build, deployment, rollout and operation, and leverages platform capabilities to build a secure platform as basis for the OS-level virtualization architecture [Fraa; Frab]. As we design our architecture for mobile devices and for embedded device use cases in the IoT, we now

discuss OS-level virtualization in these areas in more detail.

OS-Level Virtualization on Mobile Devices

Cells [And11] is an OS-level virtualization approach for Android mobile devices. The authors introduce *device namespaces* to provide a framework for device driver virtualization on kernel-level. Device namespaces multiplex hardware driver states on a per-container basis. With the concept of active device namespaces, drivers are made aware of the current *active namespace*, i.e., the foreground container. The work puts the main focus on realizing the functionality, but lacks the consideration of security aspects. No secure architecture is provided and data confidentiality is not discussed.

Based on Cells, Condroid [Che15] puts the focus on efficiency. Device virtualization is mostly realized in the Android framework. More OS resources are shared among the containers, such as their read only parts and OS services. For container management, the authors port LXC and run it in a single host Android in the root namespace. This makes the solution highly specific to a certain OS version and blends domain isolation with domain interaction, resulting in a weaker security model and a larger TCB.

AirBag [Wu14] leverages OS-level virtualization for a device model where probing and profiling of untrusted applications can be done on the respective device itself. The framework allows the user to install and execute new applications quarantined inside a second, untrusted container. In contrast to our goal, their objective is the preliminary analysis of Android applications before their execution in the trusted container.

The virtualization approach by Wessel et al. [Wes13] forms the starting point of our work. The authors leverage mechanisms, like (device) namespaces and control groups (cgroups) to realize the operation of different Android containers on mobile devices. The work lays the foundation for our secure architecture. We extend the work with a systematic approach for container isolation for achieving data confidentiality, and with mechanisms for secure device virtualization and secure container switching on mobile devices, aspects which we focus on in Section 3.3.

OS-Level Virtualization for IoT Devices

Mulfari et al. [Mul16] extended the container-based approach to embedded IoT devices, which they call *smart objects* in a message-oriented middleware for a cloud architecture named MOM4C [Faz13]. Their prototype for a smart object is a Raspberry Pi2 using Docker for containerization. The same prototype is evaluated more generally regarding container-based virtualization on IoT devices by Celesti et al. [Cel16].

In the context of IoT or embedded devices, nowadays further container-based solutions exist, such as Pulsar Linux [Ash16] or balenaOS [Bal]. Pulsar uses LXC, while balenaOS is based on Docker. However, those projects focus on the ease of deployment and developer convenience, but not secure isolation and data integrity. The latter is what we are going to focus on in Section 3.5.

3.2 Background on Linux Kernel Mechanisms for OS-Level Virtualization

In the following, we briefly introduce the Linux kernel mechanisms we leverage for our OS-level virtualization architecture.

Namespaces. Namespaces virtualize kernel resources for processes and are the foundation for containers. These virtualized resources are, for example, mount points, or process and user IDs. Groups of processes can be run in separate namespaces, making these groups individual containers. The kernel resources each container, i.e., process group, uses during runtime are then independent of processes in other namespaces, making containers unaware of others. All properties and changes to these resources are shared between the members of the same namespace, but inaccessible from outside the namespace. For the mobile device case, *device namespaces* were additionally introduced in [And11]. Device namespaces virtualize those hardware devices, for which their driver is fully located in the kernel, for example, the alarm device or input devices. This makes it possible to share hardware devices between containers.

Cgroups. The cgroups feature allows for flexible allocation and enforcement of policies on the basis of process groups, i.e., containers, to constrain their access to system resources. This makes it possible to form groups of processes and thus to enforce container-specific policies. The cgroups features is split into several subsystems representing the different policies, such as the CPU, memory, or devices subsystem. For instance, the CPU and memory subsystems allow restricting containers to a fixed share of CPU and main memory, preventing containers from exhausting other containers' resources. Another example is the cgroups devices subsystem which allows to define hardware device access rules for process groups, constraining containers from accessing certain devices.

Capabilities. Linux capabilities allow to restrict processes to fine-grained sets of permissions for bypassing kernel permission checks. The set of all capabilities forms the root user privileges set while a single capability represents a specific set of permissions. Each permission refers to one or more security-critical actions on kernel objects, such as the creation of special files or setting the system clock. Dropping capabilities allows to revoke permissions from processes or process groups. Especially, this allows to constrain privileged processes from bypassing the kernel permission checks. We use capabilities to further constrain containers and to restrict critical components from privileges that are not necessary for their operation.

Linux Security Modules. The Linux Security Modules (LSM) infrastructure of the Linux kernel provides a means to enforce custom policies on kernel objects by introducing hooks at critical points of execution, for instance, inside system calls. These hooks are callbacks, which are triggered at the respective point of execution. An LSM realization implements code at these hooks in order to enforce protection policies. This, for instance, enables us to implement an LSM aware of namespaces that distinguishes between containers and enforces different policies depending on the container. Existing LSM implementations, part of mainline kernels, are SELinux, Tomoyo, SMACK, or AppArmor, for example. AppArmor is shipped and enabled on modern Ubuntu versions and controls specific processes according to AppArmor policies explicitly written for the controlled processes [Ubu]. Android uses SELinux since version 4.3 for application sandboxing [And].

3.3 Design of the Secure Virtualization Architecture for Mobile Devices

In this part, we present our secure virtualization architecture for mobile devices and ensure data confidentiality at container boundaries. The protection of data is especially necessary for the mobile domain, because today's mobile devices are not only widely used, but also represent a fingerprint of their users. Essential corporate and private data is likely to be found on those devices, rising the necessity to carefully consider security aspects, such as data confidentiality protection. The prevalence of only few OSs, and the pace of their development make them prone to getting targeted by attackers [Bec11; Fei15]. The abundance of security issues made these devices vulnerable to a large number of attacks [Fel11; Pen14; Poe14; Zho12]. While efforts were made to mitigate the susceptibility towards common attack vectors [Alm14; Bac14; Chi11; Enc09; Ong09], none of the approaches features an overall secure architecture for data confidentiality. A promising way to approach data confidentiality is to provide multiple, virtualized containers with separated usage contexts on a single mobile device [Bra08].

We base our work on [Wes13], where Wessel et al. proposed a concept based on OS-level virtualization for Android systems. This allowed to operate multiple Android containers in parallel on a single mobile device. Not only their, but also other OS-level virtualization approaches [And11; Che15] lack a full-fledged secure architecture for the protection of the confidential data of containers. Our main objective is the confidentiality of sensitive user data at container boundaries in the presence of local and remote attackers. This means, we achieve data confidentiality when data inside a container remains inaccessible to other, possibly malicious containers at all times. To achieve this, we isolate containers by restricting them to a set of minimal, controlled functionality and confine the communication between components to only specific channels. We focus on the development of a solution including an SE for Linux driven OSs, easily portable between mobile devices, and suitable for real-life application.

We start with an overview on the components of our kernel-based secure virtualization architecture in Section 3.3.1. In Section 3.3.2 we present our concept for container isolation, where we confine containers to minimal, controlled functionality and to only specific communication channels. Based on that, we elaborate the refined secure device virtualization mechanisms in Section 3.3.3. We allow to dynamically assign hardware device functionalities on a per-container basis and classify devices into different categories. We develop a secure container switching mechanism with *security devices* in Section 3.3.4. In Section 3.3.5, we conduct a systematic security discussion of our architecture showing that data confidentiality is preserved even when large parts of the system are compromised. We describe a full implementation for the Samsung Galaxy S4 and Nexus 5 devices, available open source [Fraa], and show performance results for the Nexus 5 device in Section 3.3.6.

3.3.1 Architecture Overview

Figure 3.2 gives an overview on the components of our secure virtualization architecture and refines the system architecture from Chapter 2. The illustration depicts different containers C_0, C_1, ..., C_n sharing a single Linux kernel. We differentiate between components located in user and in kernel space. Another differentiation is between the components found on a stock Linux-based mobile device and the components we add. The latter ones are

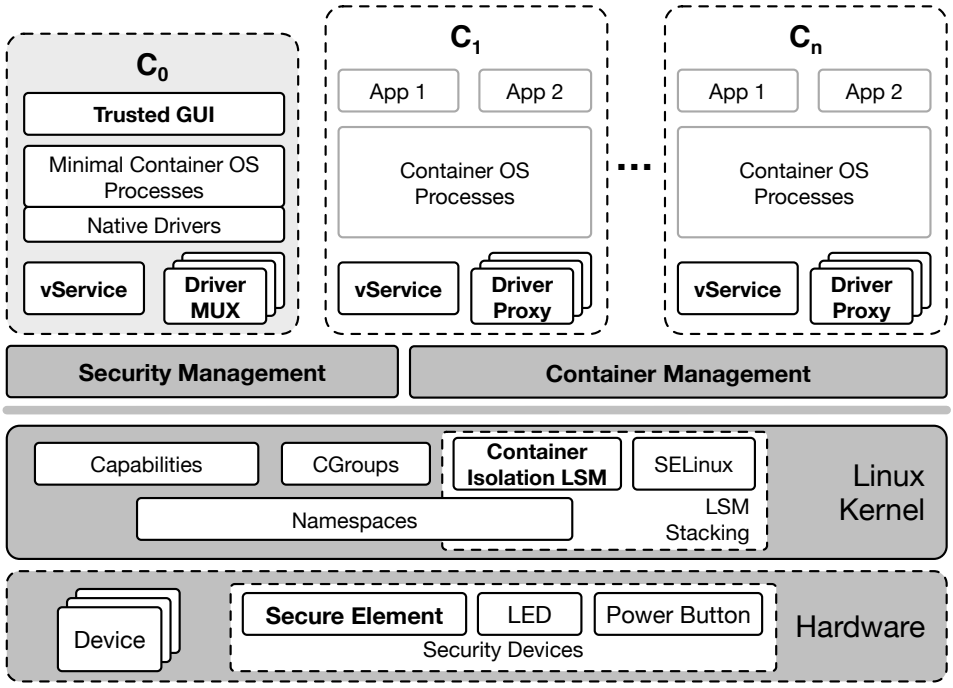


Figure 3.2: Overview on the components of the secure virtualization architecture.

highlighted in bold. The varying background grayscale colors visualize the separation of components into different privilege levels. The dark gray colored components are in the TCB. The mid gray C_0 is a *privileged container* in contrast to the *unprivileged containers* $C_{1..n}$. We explain these components in the following.

3.3.1.1 Hardware and Kernel Components

The hardware part consists of common hardware devices and *security devices*. These are non-virtualized hardware devices, because they serve a security critical purpose. We introduce the notion of security devices to provide the user devices that can be trusted even when a container is compromised and, for example, displaying bogus contents on the touchscreen. We define the SE, LED and power button as minimal set of required security devices. They are not accessible to $C_{0..n}$. The LED and power button are usually available on common mobile devices. In our architecture, the power button's purpose is to securely initiate a switch between containers, see Section 3.3.4. This allows a user to trust the switch from one container to the other is actually happening. The LED is a secure container indicator for the user, showing the unique color of the currently active container. $C_{0..n}$ are thus unable to disguise their identities to impersonate another container. We use the SE as secure storage for integrity and confidentiality protection, to be described later. The SE is a passphrase-protected device, for example, a smartcard connected via Near Field Communication (NFC). We securely virtualize the remaining devices in order to ensure a seamless user experience and the operability of the containers on the device

see Section 3.3.3. That includes, amongst others, device virtualization for graphics, input, the Radio Interface Layer (RIL), sensors, or the Wi-Fi functionality. In kernel space, we substitute the stock mobile device's kernel with our modified Linux kernel. We use the kernel's namespaces feature for containerization. For the isolation of containers and for controlling their access to system resources, we leverage the capabilities and cgroups kernel features, as well as LSM stacking functionality, using Security-Enhanced Linux (SELinux) and a custom LSM.

3.3.1.2 User Space Components

Containers $C_{0..n}$, and the entities Container Management (CM) and Security Management (SM) are located in user space. Only the SM and the CM are part of the TCB.

Security Management. The SM has the responsibility to securely and exclusively communicate with different SEs. The SM performs cryptographic operations for the CM, such as unwrapping keys for container storage encryption using the SE.

Container Management. The CM configures the kernel features and acts as mediator between the containers. It has exclusive access to the LED and power button. The CM is responsible for container operations, such as to start $C_{0..n}$ or to securely switch between containers, see Section 3.3.4. The CM is also responsible for setting up container storage encryption. We protect container storage with a symmetric container key. This key is wrapped with the public key belonging to the SE's private key. When a container starts, the CM asks the SM with a provided passphrase to unwrap the container storage key using the SE.

Container C_0. This is a special, privileged container, comparable to *dom0* in Xen [Bar03]. This container realizes functionalities required for the virtualization of the user containers. For this purpose, the container uses its privileges to interact with the kernel with less constraints - for instance for hardware device access, interaction which we partly prohibit other containers from. To further reduce the size of the trusted virtualization layer, we encapsulate such functionalities not necessarily required to run directly in the virtualization layer into C_0. Once an attacker manages to compromise functionality in C_0, e.g., through untrusted input channels, the attacker still lacks control of privileges to access other containers' data.

C_0 provides users with an interface for local container management with a Trusted GUI and provides functionalities for secure device virtualization, see Section 3.3.3. The Trusted GUI enables the user to securely enter the passphrase required for starting containers, to initiate a container switch and to make container-specific and device-global settings. We use the Driver MUX entities as device multiplexers for user space device virtualization over container boundaries. Device drivers, often proprietary binaries, are mostly running only within a userland OS, such as Android. We therefore require C_0 to run a minimal OS for hardware device driver access.

Container C_i. These components are the isolated and unprivileged containers. The CM encapsulates $C_{0..n}$ into their specific namespaces, maintained by the kernel. During start-up of a container, the CM, creates the namespaces for C_i and configures the

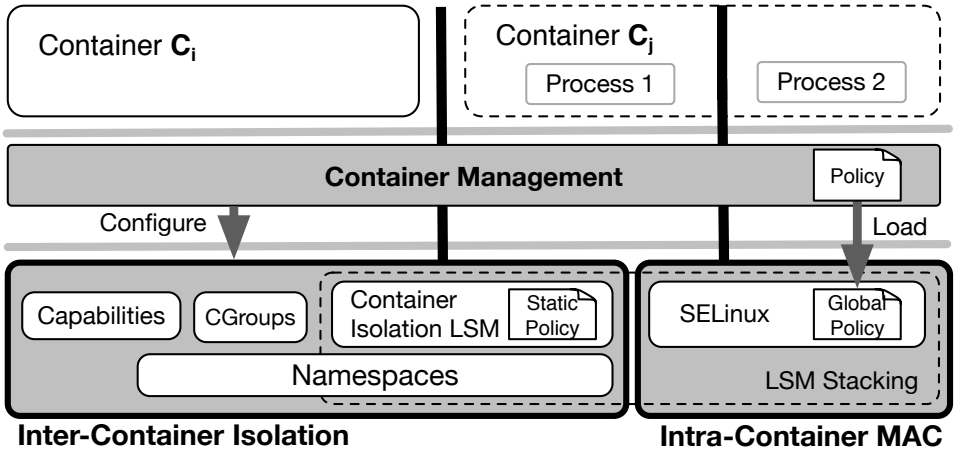


Figure 3.3: Overview on the kernel mechanisms used for container isolation.

security mechanisms. The vService in each container realizes an interface to the CM in the root namespace. We use this interface for sending commands to $C_{0..n}$, for example, to shutdown, suspend or resume. The Driver Proxies request device functionality from C_0's Driver MUXs, see Section 3.3.3. This enables C_i to make use of specific device functionalities without direct device access. For the channels we introduce between container boundaries, we introduce well-defined interfaces and allow no other communication by isolating them from unnecessary OS functionalities.

3.3.2 Container Isolation

In the following, we describe the isolation of the containers from each other and from the root namespace. In order to achieve strict isolation, we restrict $C_{0..n}$ to a minimal set of functionalities. We allow communication only over well-defined and protected communication channels. Figure 3.3 depicts a detailed view on the container isolation at the example of C_i and C_j with the kernel mechanisms we make use of. We isolate components on intra- and inter-container basis. We support and enforce the commonly deployed LSM implementation SELinux inside containers. This isolates processes inside containers to protect it from being compromised at first instance. The CM loads and enforces a global LSM policy for each container. Individual SELinux policies for containers are not applicable, because SELinux is not virtualized, i.e., not aware of namespaces. We also require LSM mechanisms for inter-container isolation. Therefore, we use the LSM stacking mechanism [Sch]. This mechanism allows to register multiple LSMs in the kernel. Multiple handlers are hence called on an LSM hook to perform access control. A hook is successfully passed only if each of the handlers grants access to the kernel resource.

3.3.2.1 Communication Channels

We specify secure and exclusive communication channels between the components over well-defined interfaces. This restricts the components to interfaces exclusively used for container

management and for secure device virtualization. First, we classify communication channels into three different layers of communication, as depicted in Figure 3.4.

Layer 1 Communication. Layer 1 communication is on system call level, i.e., calls like `open`, `write` or `ioctl`, which are executed in the kernel. Any communication between components results in layer 1 communication interacting with the kernel. On this layer, we prevent containers from unspecified device access using the cgroups devices subsystem based on device major minor numbers. We allow $C_{0..n}$ to directly access device drivers virtualized on kernel-level via device namespaces, see Section 3.3.3. To constrain components from critical system calls, we use Linux capabilities and our LSMs.

Layer 2 Communication. Layer 2 communication involves the communication between two or more processes. This layer represents all types of low-level IPC over OS resources, for instance, sockets, and results in system calls. We separate this layer between containers through namespaces isolation. With our custom LSM, we selectively allow access to defined kernel resources relevant for IPC. An example is the denial of accessing certain sockets. This makes it possible to explicitly grant or refuse the establishment of communication channels.

Layer 3 Communication. This layer uses a protocol for IPC between the components. We secure the communication by message filtering and by utilizing a secure protocol. Figure 3.4 illustrates the following layer 3 channels we allow.

CM with SM: The CM uses this channel in the root namespace to retrieve the results of the cryptographic operations that the SM executes.

CM with external components: For remote device management via a backend, the CM offers a protocol on an update and remote control interface.

CM with vService: To send commands to $C_{0..n}$ and to check their status, the CM communicates via the status interface with the vService inside $C_{0..n}$.

CM with Trusted GUI: The CM offers a control interface for local container management. The Trusted GUI in C_0 uses this control interface.

CM with Driver MUX: The Driver MUX utilizes this channel to notify the CM via the vDeviceRegister interface of the user space-virtualized device functionality the multiplexer offers.

CM with Driver Proxy: The Driver Proxy uses this channel to demand the CM via the vDevice interface for setting up the connection channel to the Driver MUX to obtain functionality of user space-virtualized devices. Only the CM may grant and set up this channel.

Driver MUX with Driver Proxy: This channel, set up by the CM, exists for user space-based device virtualization. C_0 accesses hardware devices on layer 1 on behalf of C_i and selectively provides the functionality to C_i.

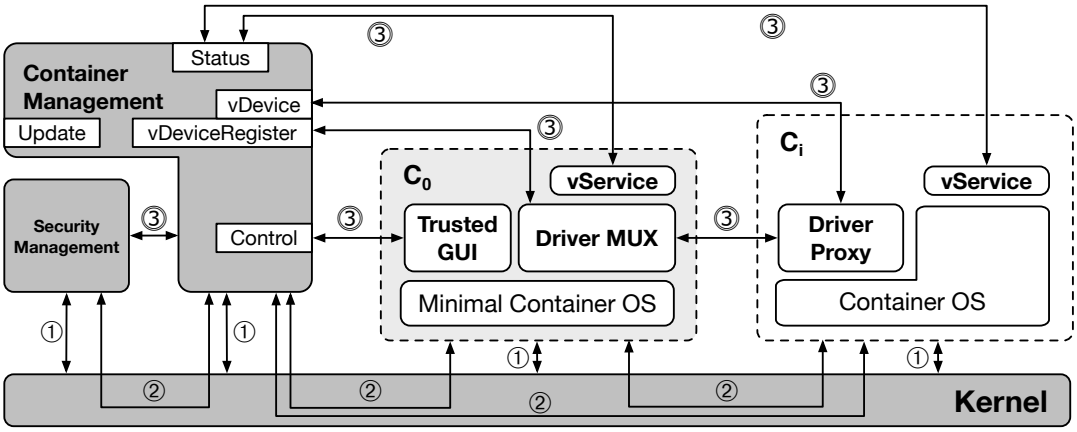


Figure 3.4: Communication channels between the entities of the secure architecture.

3.3.2.2 Identification and Isolation of OS Functionalities

To enforce data confidentiality across container boundaries, we prevent $C_{0..n}$ from crossing namespace boundaries through other than the specified channels. In order to do so, we confine the containers to minimal OS functionalities with kernel security mechanisms, as depicted in Figure 3.3. System calls represent the interface via which all components act and are thus critical for the isolation. In order to achieve a global view on these resources, we investigate all system calls and their usage. Based on the whole set of system calls, we try to identify and group OS functionalities. In the following, we list these functionalities and elaborate their protection using the aforementioned security mechanisms.

Mounting. We only allow containers to execute non-critical mount operations. First, we embed every container into its own mount namespace, which provides each container with isolated filesystem mount views. For managing the mount permissions of containers, we then introduce mount restrictions with our custom LSM. This prohibits mounting of non-required resources and specifies paths where a container can mount to. For example, $C_{0..n}$ are only allowed to mount `sysfs` to `/sys` and `procfs` to `/proc`. Containers are, for example, not allowed to mount cgroups, which prevents $C_{0..n}$ from overwriting cgroups configurations. Our custom LSM performs the mount permission checks based on a static mount whitelist in our LSM policy. The list specifies the device, mount point, filesystem type and mount flags. We furthermore drop the capability `CAP_SYS_ADMIN`, because it comprises various other critical functionality we prohibit, as described later. However, the mounting privileges are part of this capability. We therefore introduce a new capability `CAP_SYS_MOUNT`, which only allows a process to (un)mount and to create new mount namespaces. The new capability contains the minimal required subset of mount-related privileges former part of `CAP_SYS_ADMIN`.

Filesystem Access. For some of its mounted filesystems, a container should only have limited file access. To achieve this, we define protection rules with our custom

LSM. These rules restrict containers to access only specified locations in their own filesystems. We may assume fixed locations of objects in the filesystem due to the fixed mount points we defined. We utilize path-based whitelists to specify the access permissions for filesystem locations. We define *read-write*, *read-only* and *privileged container* whitelists in our LSM policy. The LSM traverses the whitelists when the system triggers the corresponding LSM hooks, for example, for the `open` or `ioctl` system call. An example is the access restriction to the `sysfs` filesystem. We allow a container to mount it in order to operate correctly, but we limit the access in this filesystem. For example, the LSM restricts an attempt from $C_{0..n}$ to set the LED color via the `sysfs` filesystem.

Device Access. Containers must be able to fulfill their usage purpose, which often requires virtualized device functionality from C_0, such as telephony or sensors, see Section 3.3.3. The goal is thus to enforce fine-granular control over device access permissions on a per-container basis. We grant or deny containers access to devices using the cgroups devices subsystem. This subsystem uses a whitelist configuration. The list specifies rules, which contain the device major minor numbers, its type, and the kind of operation allowed, for instance, `mknod`, `read`, `write`. The `/dev/random` pseudo device is an example for a device we allow a container to access. Since each container is in a different cgroup, we provide different per-container configurations. We adapt the configurations dynamically according to whether a container is in the fore- or background, see Section 3.3.4, as containers may have different requirements and different allowed device access behavior in either state. With the device namespaces for kernel-virtualized devices, we provide filtering mechanisms for fine-granular usage control of a device's functionality even when device node access is generally granted. Using the described mechanism allows us to permit the container to populate its own device directory. This results in less changes to containers and provides maximum compatibility. Therefore, we do not drop the capability `CAP_MKNOD` used for creating filesystem nodes. We enforce the security in using `mknod` via cgroups devices and LSM mount whitelisting.

IPC. To achieve container isolation, we generally restrict all kind of IPC between namespace boundaries. Solely for container management and secure device virtualization, we allow IPC functionality via protected and controlled communication channels, as described in Section 3.3.2.1. For inter-container isolation, IPC namespaces provide containers with dedicated resources for IPC inside containers and isolate them at container boundaries. With our custom LSM, we restrict unauthorized namespace-crossing IPC. The LSM considers the PID namespaces for file-based IPC via the mounting and filesystem access restrictions. An example are checks for socket functionality with LSM hooks responsible for controlling inter-container IPC. We drop the capabilities `CAP_IPC_(OWNER, LOCK)` and `CAP_SYS_ADMIN`. These capabilities include critical IPC privileges. For instance, `CAP_IPC_LOCK` allows a process to lock memory, for instance, to prevent the OS from swapping. Such locking goes beyond the scope of a container and can lock-up the whole system if used by a malicious process.

Networking. We allow containers to individually setup the network within their boundaries. Therefore, we keep the networking capabilities `CAP_NET_*`. We embed containers into their own network namespace. Thus, the scope of the capabilities is limited to only affect the container's own subnet. However, we preserve the privilege to control the network dataflow. The CM sets up the global network configuration of the containers. The CM provides virtual network interfaces (`veth`) for each container with individual IP addresses. We filter and control network packages on global level with netfilter components.

Signal Handling. With signals, a malicious container might adversely influence other components of the architecture. However, a container should be capable of sending signals inside its namespace. We thus restrict containers from sending signals over namespace boundaries. We secure this functionality through PID namespaces, which ensure that signals from processes remain only visible inside a container's namespace. Therefore, we are not required to drop the capability `CAP_KILL`.

Resource Consumption. We provide containers access to sufficient system resources for working conveniently, but not to excessively exhaust resources. Mount namespaces provide a container with its own fixed and limited filesystem. With the cgroups CPU subsystem we determine a maximum share of the CPU resource for a process group. With the memory subsystem, we ensure that a process group can only allocate a fixed maximum amount of memory. This prevents containers from excessively slowing down the system by blocking the CPU or exhausting memory. We do not need to drop the capability `CAP_SYS_NICE`, because even if a container changes process priorities, it cannot exceed its CPU usage limit.

Process Management. We grant a container to reduce the capabilities of its processes. We thus do not drop the capability `CAP_SETPCAP`. It allows processes to drop capabilities for child processes. The `init` process of each container has a reduced set of capabilities and is only allowed to further reduce this set. We prevent processes from process directory manipulation and accounting. For that purpose, we drop the capabilities `CAP_SYS_(PACCT, CHROOT, ADMIN)`.

Time Management. We allow only C_0 to set the system time. Consequently, we drop the capability `CAP_SYS_TIME` from C_i in order to prevent them from setting the system time via system calls. However, in Android the time setting functionality works via the `/dev/alarm` driver. This is unfortunately not covered by `CAP_SYS_TIME`. We thus prohibit the access to `/dev/alarm` driver functionality for C_i with LSM hooks in our custom LSM.

Power Management. In order to prevent containers from changing the global power state, such as from shutting down or waking the system, we drop the capabilities `CAP_SYS_BOOT` and `CAP_WAKE_ALARM`.

Kernel Module Loading. We prevent containers from loading kernel modules by dropping the capability `CAP_SYS_MODULE`.

Debugging. To prohibit containers from obtaining debugging control over other processes, we drop the capability `CAP_SYS_PTRACE`. We also enforce this with PID namespace

checks in the ptrace controlling LSM hooks.

Logging. In order to prevent containers from making changes to the kernel logging functionality, we drop the capabilities `CAP_AUDIT_(WRITE, CONTROL)` and `CAP_SYSLOG`.

3.3.3 Secure Device Virtualization

Our architecture allows to securely and dynamically assign device functionalities to $C_{0..n}$ on a per-container basis. We classify each device as either a *non-virtualized*, *security*, *user space-virtualized*, or *kernel-virtualized* device. The classification depends on whether a device is, e.g., used only by the virtualization layer, or whether device drivers to be virtualized are implemented only in kernel or also in user space. Depending on device type and container, we handle its access to device functionality. Figure 3.5 depicts the virtualization and access mechanisms for the device types.

3.3.3.1 Non-Virtualized and Security Devices

Security devices are part of the non-virtualized device category. We do not virtualize security devices, since they provide critical, security relevant functionality. In our architecture, these are the SE, LED and power button, as defined in Section 3.3.1. We prohibit $C_{0..n}$ access to these devices, as depicted by the crossed dashed lines in Figure 3.5. The hardware driver for accessing security devices is exclusively accessible to management components inside the TCB. As described in Section 3.3.2, we restrict access using the cgroups device access protection mechanism.

We allow access to other non-virtualized devices only to foreground containers. We enforce this device access rule during a container switch by dynamically adapting the cgroups devices whitelist, which we cover in more detail in Section 3.3.4. An example for such a device is the display, exclusively used by the foreground container.

3.3.3.2 Kernel-Virtualized Devices

We virtualize kernel-virtualized devices on kernel level using the device namespace mechanism [And11]. In Figure 3.5, C_i is in foreground as active device namespace, while C_j is in background. C_i is trying to access kernel-virtualized devices from userland. The device driver in the kernel is addressed via the container's `/dev` filesystem. Examples for kernel-level virtualized devices are the alarm and input device (except for the power button), handled via the `/dev` filesystem. The driver decides about access to the functionality it offers based on the information about the active namespace (provided by the device namespaces). This is represented by the device namespace filter component in Figure 3.5. This enables us to selectively grant or refuse functionalities to background containers running in inactive device namespaces. With the cgroups devices subsystem, we have an additional driver-independent and dynamic mechanism to deny containers access to a device, i.e., to deny access to the device driver.

3.3.3.3 User Space-Virtualized Devices

A lot of devices are accessed via proprietary user space drivers. In user space, we can re-use the existing drivers and achieve a portable solution. Thereby, we do not expand the TCB and avoid growing kernel complexity incurred by the virtualization. Since critical

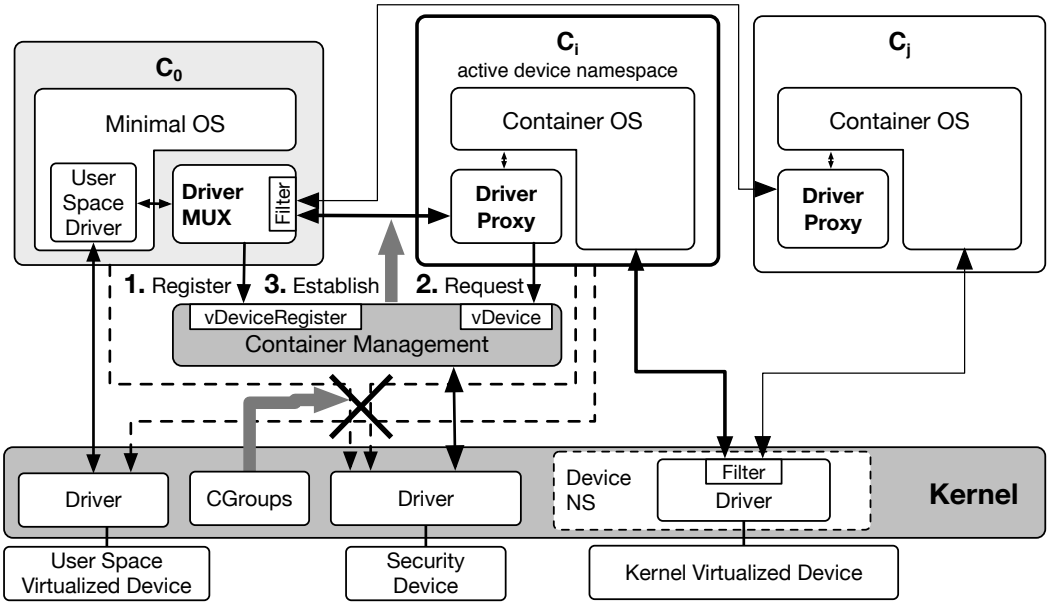


Figure 3.5: Overview on our different secure device virtualization mechanisms.

and exposed to untrusted input, we place the virtualization functionality inside C_0 and reuse its userland drivers. With the cgroups devices subsystem, we grant access to user space-virtualized devices exclusively to C_0, highlighted by the crossed dashed lines in Figure 3.5. Like C_i, C_0 is also allowed to use kernel-level virtualized devices, omitted in the illustration.

The Driver MUX in C_0 multiplexes the hardware device functionality for C_i. It utilizes the existing userland functionality and user space driver for hardware device access. The Driver MUX keeps track of the driver states and is aware of the different C_i. We forward device functionality from C_0 over a dedicated communication channel to C_i. User space components in C_i are not aware of the Driver Proxy redirection. The CM sets up the channel between C_0 and C_i. The Driver MUX registers the device it virtualizes at the CM via the vDeviceRegister interface. When C_i tries to make use of a user space-virtualized device's functionality, the Driver Proxy requests the CM for setting up a communication channel to the Driver MUX over the vDevice interface. Depending on whether we allow C_i access to the functionality of that device, the CM establishes the communication channel, as illustrated in Figure 3.5. This can, for example, be realized by creating a socket pair in the CM with the system call `socketpair`. The CM also informs C_0 of C_i requesting the device functionality. C_0 is thus aware of the specific container it is communicating with to securely provide C_i with different sets of functionalities for each hardware device. An example is the radio interface where C_i might be allowed to use the telephony and mobile data feature, while C_j might only be allowed to make use of the mobile data feature. The filter in the Driver MUX selectively handles device functionality access for C_i and filters non-protocol compliant data.

3.3.4 Secure Container Switch

When the user is in C_0, we use the Trusted GUI to trigger a switch to C_i. The user initiates a container switch from C_i back to C_0 by a long power button press, i.e., by using a security device. The CM handles the switch between containers. In the following, we describe the container switch procedure and its initiation inside C_i.

3.3.4.1 Container Switching Procedure

Figure 3.6 depicts the container switch procedure. The illustration shows the switch between a foreground container C_F and a background container C_B to be put to foreground. The CM requests the suspension of C_F via the status interface to the vService. The vService triggers the suspend routine of the container OS, for example, `PowerManager.goToSleep` on Android. The CM waits in a non-blocking mode for the OS to suspend. In the next step, the CM restricts C_F access to non-virtualized (and possibly kernel-level virtualized) devices, which are prohibited to background containers. We achieve this by dynamically reconfiguring the cgroups devices whitelist in the CM. With this mechanism, we separate device access decision making from device functionality filtering while accessing the device driver. A container could refuse or fail to suspend if certain processes do not release their resources. In that case, the CM kills those suspend blocking processes after a timeout. This forces the open devices to be closed and the container to suspension. In the next step, the CM grants C_B device access via dynamic cgroups device allocation. The following step is to switch the active device namespace to the new foreground container. The CM requests the resume of C_B via the container's vService, for instance, `PowerManager.wakeUp` on Android OS. To complete the container switch process, the CM sets the LED color according to the color of C_B via the kernel using the LED driver.

3.3.4.2 Switch Initiation in C_i

In order to securely switch to C_0 despite being in possibly malicious C_i, we use the power button. As a security device, it is exclusively accessible by the CM, meaning that power button events never arrive in $C_{0..n}$. We define the behavior, visualized in Figure 3.7, as follows: Pressing the power button in C_i for more than a fixed time interval ε, for example, 0.5 seconds, triggers a switch to C_0. Otherwise, the button triggers the suspend or resume functionality of C_F. We modify the kernel in order to forward power button events exclusively to the CM, i.e., to the root namespace. The power button driver notifies the kernel of a power button pressed event (`KEY_POWER`, 1). The kernel forwards this event to the CM, which starts a timer at time t_1. When the power button is released, the release event (`KEY_POWER`, 0) arrives at the CM at time t_2. The CM then decides about the action to be carried out according to the fixed time interval ε. Until now, none of the events has reached any of the containers. When the user is in C_i and t_2-$t_1 \geq \varepsilon$, the CM conducts the switch to C_0. Otherwise, the CM transparently forwards the power button press and release events to C_F, resulting in either a resume or suspend. For the injection of power button events into $C_{0..n}$, we add a custom event, `KEY_POWER_INJECT`, to the kernel. We modify the kernel to recognize this special event type and to forward it as a common `KEY_POWER` event type to C_F. The power button event now appears to C_F as a common input event resulting from an input device. In case the foreground container is C_0, the CM

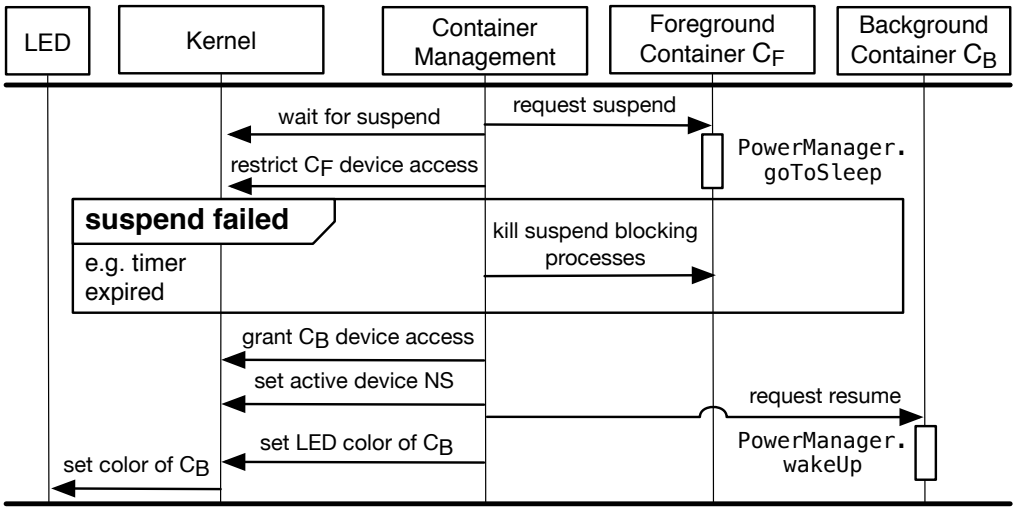


Figure 3.6: Secure switching procedure between a fore- and background container.

always injects the power button events unmodified.

A malicious container spoofing C_0 could try to trick the user into believing of having switched, while stuck in the malicious one. In the worst case, the user might enter critical information inside malicious C_i. As the LED is a security device, we use it in order to securely identify C_F. The CM sets the LED color to the container's specific color.

3.3.5 Security Discussion

In this part, we present a conceptual analysis of the security of our architecture regarding our main protection objective: the protection of data confidentiality. In accordance with Chapter 2, we assume local and remote attackers able to execute runtime attacks and furthermore to be capable of acting as a Man-In-The-Middle (MITM) between the device and backend according to the Dolev-Yao model [Dol81]. We also assume an attacker to obtain physical access to the device trying to manipulate it via common physical interfaces, for instance, via USB and the touchscreen, according to the definition of the local and remote attacker from Chapter 2. A typical attack scenario is given in Chapter 2 with scenario one. However, we do not consider the physical attacker from Chapter 2. This excludes side-channel attacks, especially cold boot, DMA, and JTAG, capabilities which the physical attacker from Chapter 2 has, see scenario two and three. Note that we address protection against the physical attacker in Chapter 5. Note that our physical attacker is unable to conduct covert channels and advanced physical lab attacks on the device or the SE, such as microprobing attacks. As a refinement to Chapter 2, we however consider the remote attacker to be able to fully compromise a remote management backend.

Through these attack vectors, we consider an adversary having the capability of compromising every component outside the TCB. This includes taking full control over the privileged C_0 and the unprivileged C_i. If a container is compromised, our isolation mechanisms ensure that the attacker with local root privileges cannot break out of the container's

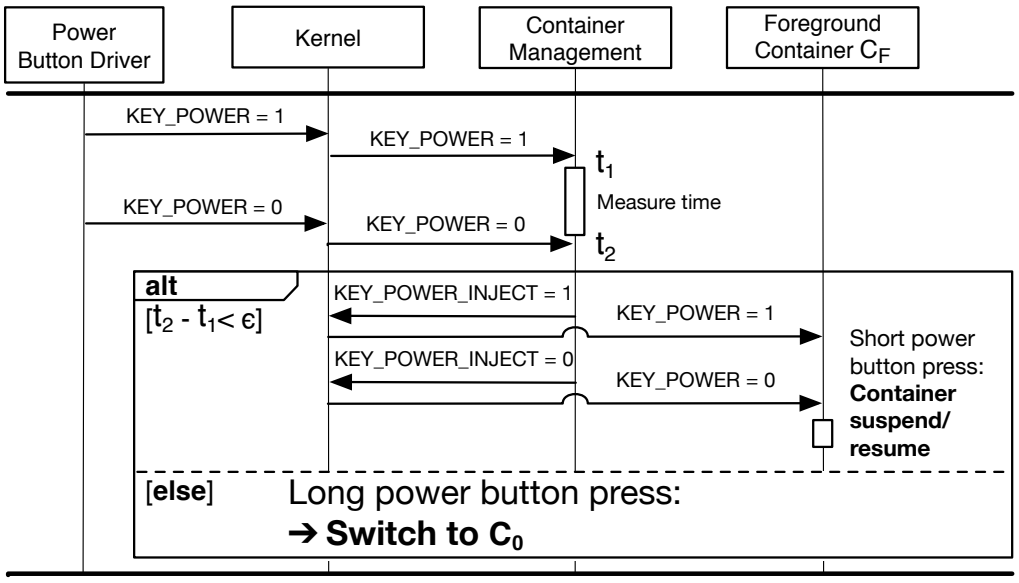


Figure 3.7: Power button event capturing and injection procedure.

boundary, unable to leverage global privileges. An attacker can affect other components only through the specified communication channels. In the following, we discuss different compromise scenarios and the implications on data confidentiality.

3.3.5.1 Compromise of C_i

C_i is exposed to common attacks on the OS. In order to harden a container from compromise, we propose to limit it to trusted applications and to functionality required for its special purpose only. The processes inside C_i are isolated and protected by SELinux. Full control over the container and its data is only exposed when the attacker manages to take control over a process and to circumvent SELinux protection. C_i cannot retrieve more device functionality via established user space virtualization channels than it is supposed to. C_0's Driver MUXs prevent this by making data routing decisions and input validation. C_i is also not capable of retrieving additional device functionality via the vDevice interface, as the CM handles setting up the connection between C_0 and C_i. The cgroups devices subsystem and device namespaces prevent C_i from prohibited device access to kernel-level virtualized devices. C_i can send fake status information or refuse commands from the CM via the status interface. However, C_i cannot deny container switching. Consequently, the overall system's behavior is not adversely affected by the compromise of C_i. Data confidentiality is retained beyond container boundaries, meaning that sensitive user data stored in other containers remains protected.

3.3.5.2 Compromise of C_0

We tailored C_0 to the minimal amount of required functionality. SELinux policies further raise its security level. However, proprietary code of the container's drivers cannot be

completely controlled. In case of the compromise of C_0, the attacker has access to the local control interface to the CM. The adversary can misuse device management functionality and intercept the user's passphrase entry for the SE. In this case, the attacker can start containers without user interaction when the SE is present. The attacker is also able to change settings, create and shutdown containers. C_0 has full access to user space-virtualized devices and to already established Driver MUX channels. The attacker can hence drop, eavesdrop and transmit forged data from, resp., to those devices. Sensitive data transmitted over these communication channels must be encrypted to protected from C_0. The adversary controls the registering of device functionalities via the vDeviceRegister channel, but cannot set up new channels. The same consequences as for a compromised C_i hold regarding the status interface. Kernel-level virtualized devices and security devices cannot be impaired. The adversary also cannot take advantage of the update functionality of the CM, since the update interface is not accessible for C_0. In sum, the attacker controls many functionalities. However, the data in other containers remains confidential.

3.3.5.3 Compromise of the TCB

The TCB exposes full access and control over all functionalities, communication channels and data on the device. In contrast to running containers, non-running ones are still encrypted and hence remain opaque to the attacker. Only if the passphrase of the SE was intercepted and the SE is present, as well as unlocked, the adversary is able to retrieve to the containers' data. If the passphrase was not intercepted, the attacker cannot brute-force a present SE, because it locks itself after a certain amount of retries. In order to obtain control over the SE, physical access to the SE is required. The attacker has control over the backend communication channel between the device and backend. This exposes the capability to download updates and encrypted backups.

3.3.5.4 Backend and Communication Channel

The network communication between the device and backend is exposed to attacks according to the Dolev-Yao model. The channel is protected by TLS encryption using certificates, which prevents gaining control over this channel. In case the backend is compromised, the adversary can access the CM's control interface. The attacker is furthermore capable of carrying out denial of service attacks towards the device. The device verifies software updates through signature verification. The adversary cannot sign updates of the device's software entities, since a valid assumption is that the software-signing key and functionality are separated from the backend. Data confidentiality is hence preserved.

3.3.5.5 Physical Device Access

If the device is switched off and an attacker manages to extract all data, data confidentiality is not impaired. The storage cannot be decrypted, since the SE and its passphrase are both required for decrypting the containers. We also lock the device to prevent attackers from overwriting partitions, for example, in firmware upgrade mode. We provide our own tailored recovery image featuring only uncritical functionality. Connecting to the USB port, an attacker has no access to the device's data and functionality. We remove functionalities, such as ADB, or mounting the device storage.

3.3.6 Implementation and Performance Evaluation

We fully implemented the proposed architecture for the Samsung Galaxy S4 and Nexus 5 smartphones, where we used Android 5.1.1, resp. Android 4.4.4, as container OSs. The implementation is open source, accessible via [Fraa]. We verified the easy portability of the architecture by a proof-of-concept prototype for the Nexus 7 tablet.

Linux kernel. We enabled the support of namespaces, capabilities and cgroups features on the device's kernel (AOSP kernel 3.4 [Gooc]). We extended the kernel with our new capability described in Section 3.3.2 and with power key event capturing, as described in Section 3.3.4.2, as well as with the LSM stacking feature proposed in Section 3.3.2. For the latter, we included the LSM stacking patch, SELinux and our custom LSM implementation. We used device namespaces [And11] for kernel-level virtualization, for example, for the alarm, audio, binder and input devices.

CM. We implemented the CM as a non-blocking callback-based daemon in C using the epoll, inotify and timer kernel features. In contrast to LXC, the CM is a specifically tailored, minimalist implementation. It consists of less than 10,000 lines of code. We realized the update, control, vDevice and status interface's protocol layer with protobuf [Gooe]. Protobuf serializes structured data transmitted over the different components and validates input. The CM processes incoming messages with callbacks. For the internal status, control and vDevice interfaces, we used UNIX domain sockets. We realized the remote control and update interfaces for the backend with TLS protected internet sockets. The CM establishes communication channels into C_i during its startup procedure. For that purpose, the CM creates a new Unix domain socket and inherits the corresponding file descriptor to the newly created root process of C_i. The root container process, still under control of the CM, binds the socket to a specific location in the container's filesystem. This location inside C_i can be accessed by specific processes supporting the virtualization, for instance, the vService or the Trusted GUI. The CM listens on the shared file descriptor and is hence able to accept connections from these processes over container boundaries. The CM sets up the cgroups subsystems, drops capabilities, loads the SELinux policies and revokes custom LSM privileges for C_i before delivering control to the container's `init` process. For dropping privileges, our LSM provides a special file in the `securityfs` pseudo filesystem. As soon as a process opens this file, its namespace and nested namespaces lose their LSM privileges, which is a one-way operation. We drop the set of capabilities in C_i to a minimal amount, represented by bold letters the following list:

CAP_IPC_(**LOCK**, **OWNER**)

CAP_MAC_(**OVERRIDE**, **ADMIN**)

CAP_AUDIT_(**WRITE**, **CONTROL**)

CAP_DAC_(**READ_SEARCH**, OVERRIDE)

CAP_NET_(RAW, BROADCAST, BIND_SERVICE, ADMIN)

CAP_SYS_(**MODULE**, **CHROOT**, **PACCT**, **BOOT**, **ADMIN**, **TIME**, **TTY_CONFIG**, **PTRACE**, NICE, RAWIO, RESOURCE, MOUNT)

CAP_(**LEASE**, **SETFCAP**, **LINUX_IMMUTABLE**, **WAKE_ALARM**, **SYSLOG**, **FSETID**, CHOWN, KILL, FOWNER, SETGID, SETUID, SETPCAP, MKNOD)

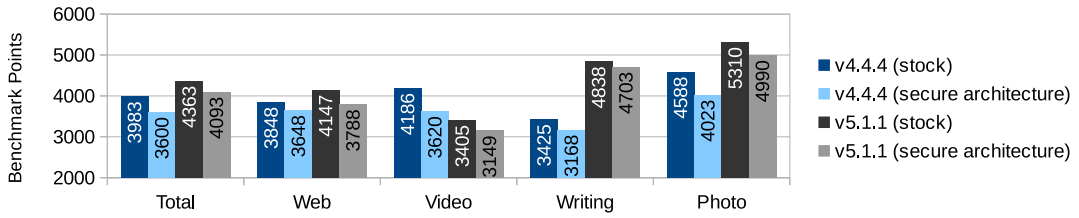
In C_0 we drop the same capabilities, but retain CAP_SYS_TIME for C_0's time setting functionality.

SM. We also implemented this component in C as a non-blocking callback-based daemon using the epoll, inotify and timer kernel features. The SM includes the OpenSSL library for cryptographic operations. In our implementation, we replaced the SE by a PKCS12 softtoken. With the token's private key, the CM wraps the symmetric key for container en-/decryption using `dm-crypt`. We protect the container images with a hash in a signed container configuration, including mount points, and user data images. The SM implementation comprises less than 1,500 lines of code.

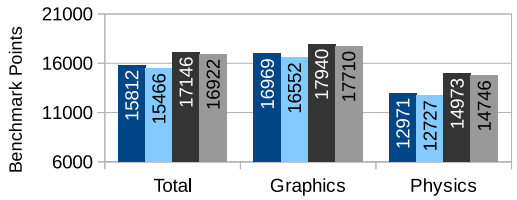
C_0. This container runs a minimal Android. We kept only basic functionality and the native user space drivers and modules, such as the daemon `rild` for accessing the radio hardware. We implemented the vService as an Android Service and the Trusted GUI as an Android system application. We realized the Driver MUXs as daemons that utilize the native drivers for user space-virtualized devices, such as RIL, Wi-Fi and sensors, including GPS. The interfaces for user-space virtualization are also realized via UNIX domain sockets.

C_i. We modified the `init` process of Android to prevent `init` from firmware loading. The firmware is loaded only once into the system by C_0. We also prohibit the OS from loading the SELinux policy. We modified the Android framework to comply with a dropped resource set. For example, the stock Zygote process checks capabilities and would prevent the OS from booting.

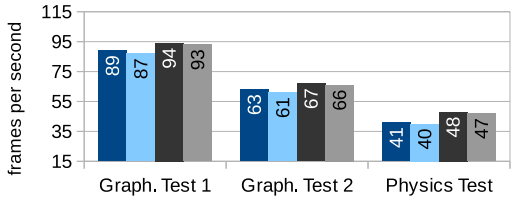
To evaluate the performance impact of our implementation, we ran the benchmark tools PCMark [UL b] and 3DMark [UL a] on the Nexus 5 with stock Android and with our secure architecture on Android 5.1.1, resp. Android 4.4.4. PCMark measures performance and battery drain based on regular user behavior, such as when using a web browser, a video and photo editor, or data and document processing while 3DMark specifically measures CPU and GPU performance. For our test, we deployed $C_{0\text{-}2}$ onto the secure architecture and ran the containers simultaneously. We executed the benchmarks in foreground C_2. Figure 3.8 summarizes the performance results, which are average values over more than 30 test runs. The total number of points achieved with our secure architecture in 3DMark is close to the stock device results, because 3DMark is rather stressing the graphics hardware. 3DMark determines this figure based on the graphics and physics test results in Figure 3.8b, deducted from the results in Figure 3.8c. The performance impact of our architecture in PCMark is no more than than 6.5% compared to stock Android 5.1.1, resp. 10% with Android 4.4.4. PCMark obtains the total amount of points by aggregating over the subtest



(a) Evaluation results for the PCMark workbench test



(b) 3DMark results (Ice Storm Unlimited)



(c) 3DMark subtest results

Figure 3.8: Performance comparison for the Nexus 5 device between stock Android and our secure architecture using the benchmarks PCMark and 3DMark on Android 4.4.4 and 5.1.1

results in Figure 3.8a. In general, the user experience with the secure architecture exposed no recognizable performance impact.

We measured the container switching time, $C_{0\text{-}2}$ running, from C_0 to C_i and vice versa when C_F is not suspended. The switching procedure consumes about 330 ms to switch from C_i to C_0 and 300 ms from C_0 to C_i. High load in C_1 and C_2, such as running HD videos, caused only negligible overhead. Most time is allocated for suspending C_F. We measured the switching time in case C_F is already suspended to consume only about 60 ms. Thereby, most time is spent in resuming former C_B and only very little time in the CM.

3.4 Application of the Secure Architecture in Corporate Environments

In this part, we develop our holistic security concept for the application of the secure virtualization architecture for mobile devices in productive end user environments, i.e., Contribution 3 addressing Challenge 2. We design an ecosystem in which the devices are utilized, and describe its entities with their relationships. Together with a description of a secure device lifecycle, this makes the architecture suitable for use in real-world scenarios. The overview on the ecosystem is part of Section 3.4.1. Following that, we describe the PKI to establish trust between all the entities in Section 3.4.2. We elaborate the concepts for secure device provisioning and enrollment in Section 3.4.3.

3.4.1 Ecosystem Overview

In the following, we introduce the entities of the ecosystem and describe their interaction based on a system-centric and device-centric view.

3.4.1.1 System-Centric View

We distinguish between the mobile device entity and global entities. As depicted in Figure 3.9 (right-hand side), the device entity and its software interacts with the global

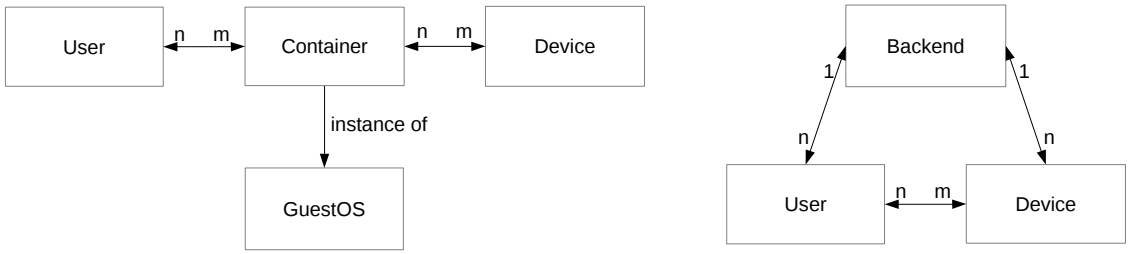


Figure 3.9: Entities on the mobile devices (left) and global entities (right)

entities user and backend. These entities represent our overall ecosystem structure and are detailed in the following.

Device. The device entity represents a mobile device, for example, a smartphone or a tablet. Each device is bound to a device certificate as described in Section 3.4.2.2.

User. The user entity represents a user identity/account. A person may have multiple identities for, for instance, private and business usage. Each identity of a user is bound to a certificate as described in Section 3.4.2.4.

Backend. Devices and users are managed by the Mobile Device Management (MDM) system. The MDM, for instance, provides updates for both containers as well as for the components of the TCB.

The 1:n relation in Figure 3.9 between the MDM and the device defines that each device is managed by one MDM, which is in turn able to maintain multiple devices. Similarly, there is a 1:n relation between the MDM and the user entity. Finally, we have an n:m relation between the user and device entity. This means that a device may be used by multiple users and each user identity may be associated with multiple devices in our ecosystem.

3.4.1.2 Device-Centric View

Figure 3.9 (left-hand side) points out the main logical entities on the mobile device. These are the container, the guestOS, as well as the device and user entities. These entities are detailed in the following sections.

To support multiple users on one physical device, our device architecture allows multiple containers on a physical device. Figure 3.9 (left-hand side) shows with the n:m relation between the user and container entity that containers on a mobile device can be used and shared by multiple users and that a user can have multiple containers. Hereby, a container is not exclusively tied to a certain mobile device. It can be present on different physical devices, and may be moved by the MDM. This is realized by the fact that the encryption of container data is linked to access tokens, for example, an NFC-based smartcard. Tokens can be moved between different devices. This is expressed by the n:m relationship between the container and device entities. Finally, the container entity is an instance of the guestOS entity, which we introduce in the following.

Device Entity. This entity represents the device-global characteristics, such as a device-wide configuration. On the one hand, the entity comprises device-wide policies. One of these policies is, for instance, the capability to receive phone calls, usually given exclusively to one of the containers. On the other hand, the device entity defines the remote link to the MDM in order to associate the device with a certain backend, i.e., with a group of managed devices.

Container Entity. To the common user, a container is one of the multiple operating system instances that runs on the device. The container entity is characterized by a configuration with different components that allow the personalized instantiation of a generic OS type, as described below. For example, each container has a user-defined container name, which is shown in the user interface and an internal container identification based on a unique identifier. It is also possible to define policies for a container, for example, to define the size of its data partition. The container configuration and the container's data are protected with a symmetric key. As there is sensible user data contained and because important container properties are defined in the container configuration, the confidentiality and integrity of the data must be protected from unauthorized third parties. The symmetric key is hence used for cryptographic operations on the container configuration and the container's corresponding data and cache partition's image files. Cryptographic operations with this key are performed on the application processor. This key is wrapped, i.e., encrypted with a private key in the SE, and can only be unwrapped when the user unlocks the SE with its passphrase. The wrapping of the symmetric key for container encryption with user-specific tokens allows to have multiple users for one container and is described in Section 3.4.2.

GuestOS Entity. The guestOS entity represents the type of a container. On a mobile device, multiple guest OSs, such as Android, Firefox OS or Ubuntu may be available and can be instantiated by a user in form of a container. The guestOS entity is represented by a configuration with general OS components and is signed with a software signing key to guarantee its integrity. This signature is verified when a guest OS is booted in order to secure the boot process, see Section 3.4.2.3. A guestOS may implement one or more features. An example is the radio feature indicating the ability to make phone calls. OS-specific features can however be revoked with a device-global configuration policy of the device entity.

3.4.2 Trust Management with a Public Key Infrastructure

This section gives an overview of the PKI and certificates used in our ecosystem. These are shown in Figure 3.10. The goal is to protect the device from unauthorized third parties trying to obtain any sensitive data. The purpose of the certificates is furthermore to harden the system against tampering to improve data integrity, e.g., it is crucial to sustain the integrity of the CM to sustain data confidentiality. The certificates enable to authenticate the backend, the device and users against each other to establish trust relationships. Our concept also includes the protection of containers and software updates against tampering. For this purpose, there are four different types of certificates involved

in the PKI: the backend certificate, device certificate, software signing certificate and user certificate. The certificates are issued by their respective Certificate Authorities (CAs) as shown in Figure 3.10. Note that we describe the certificate lifecycle on the devices in the provisioning and enrollment part in Section 3.4.3.

3.4.2.1 Backend Certificate

To allow the device to communicate with the backend, i.e. the MDM, both communication partners have to be capable of proving their identity to each other before starting to exchange data. This is realized by certificates associated to both sides. The backend certificate's function proves the trustworthiness of the MDM by transmitting it to the device, which can in turn verify the certificate's integrity by verifying the signature of the whole certificate chain up to the trusted root certificate. The device must know the trusted CA's public key to verify the validity of the certificate's signature. In case the verification is successful, the device knows that the certificate corresponds to its associated backend. This is because the device trusts the CA and because only the backend holds the private key corresponding to its certificate. The latter makes the backend uniquely able to decrypt messages from the device encrypted with the backend public key. This means that attackers cannot eavesdrop unencrypted information transmitted between the device and backend and are not able to forge malicious valid messages.

3.4.2.2 Device Certificate

For the purpose of authenticating the device on the MDM, the device certificate of the mobile device is utilized. The MDM is able to verify the device certificate in the certificate chain. The device is not able to transact with the MDM without a valid certificate, which can also be revoked, for example, in case the device gets lost or stolen. When the device is revoked within the PKI, the validation of the signature fails and the backend stops trusting the device. Revocation can either be realized using Certificate Revocation Lists (CRLs) or protocols like Online Certificate Status Protocol (OCSP)

3.4.2.3 Software Signing Certificates

Besides the device-backend communication, the software transferred to the device must be protected against malicious modifications. Therefore, we introduce software signing certificates to the PKI. These certificates are used to sign data such as software and configuration files. This especially includes the guest OSs. Every software update or change to a static configuration must be signed by the developer. On the device, the incoming update including a signature is verified with the software signing certificate it has to come along with. The certificate itself is then verified in the secure environment with the CA's public key. This way, manipulations to the guest OS and its configuration on the device will be detected. The private software signing key does not reside on the backend but is kept in a secure location which provides the backend with signed updates.

3.4.2.4 User Certificates

Our system depicted in Figure 3.9 allows multiple user identities on a single mobile device, as one or more containers on a device can be used by multiple users. Furthermore, a container's sensitive data, i.e. configuration and filesystem, is encrypted with a symmetric

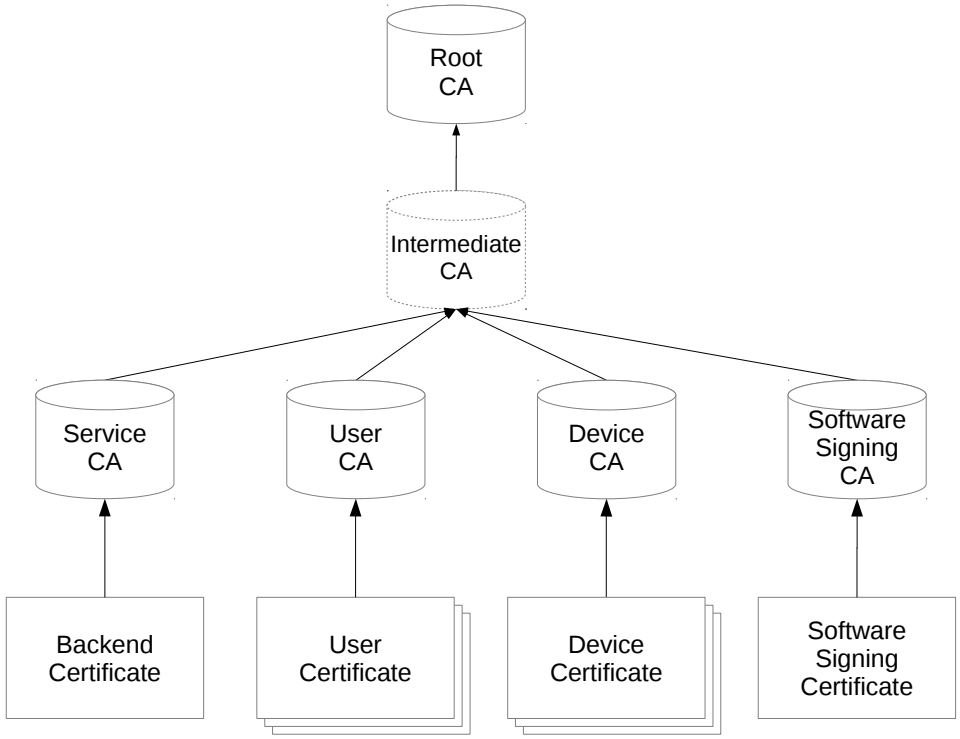


Figure 3.10: Overview on the PKI and certificates in our ecosystem for secure mobile device provisioning, enrollment and management.

key and this key has in turn to be protected. This is achieved by encrypting it with the public key of each user identity which is associated with the container. This public key forms the user certificate, as depicted in Figure 3.10, which must be signed by the user CA. The respective private key is typically securely stored on the SE and never leaves it. This key is used by the trustworthy environment when decrypting the symmetric container key on the SE. With the PKI, authorized users are thus enabled to decrypt the symmetric container key and hence the container by using their passphrase-protected SE and conducting a Multi-Factor Authentication (MFA) to unlock it. This approach enables only authorized users, associated to a container, to attain information on containers. Attackers are prevented from decrypting any of the container's information, which also improves data integrity. Note that even the MDM or an administrator can not decrypt such data. The certificates for authorized users on the device must be securely stored and transferred to the device in a provisioning step, which is explained in Section 3.4.3.

An advantage that comes with the PKI appliance is the capability of an administrator to withdraw or add accepted user certificates, for example, via the MDM. In addition, the validity of the certificates can be regulated in time and users can be revoked when they are no longer in the corporation or trusted. This principle also enables to differentiate between privileged and common users, resulting in the realization of, for instance, different policy settings for a device. Note that in a typical PKI, a user identity may have more than one

certificate, for example, different certificates for decryption, authentication and signing.

3.4.3 Device Provisioning and Enrollment Process

This section gives an overview of our concept for device provisioning and enrollment. To be able to utilize the certificates presented in the previous section, the device must be provisioned with a CA certificate, i.e., the verification root public key. To enroll the device for usage, a device certificate has to be generated for each device.

3.4.3.1 Device Provisioning

When a mobile device is released for a customer, it needs to be provisioned with the base system in a secure environment. Therefore, the public key pair of the CA and the address of the MDM must be available and also transferred. This step is necessary in order to equip the device with the certificate required to verify the user, backend and software signing certificates, see Section 3.4.2. Hence, the device is able to reliably verify identities. The CA certificate can hereby consist of further certificates in a certificate chain, which however does not affect any of the principles mentioned.

3.4.3.2 Enrollment

A provisioned device, which is supposed to be initially activated, must be securely enrolled via a channel to the certificate signer. The goal is to obtain a device certificate to be able to communicate with the backend. A random public key pair is generated on the device in hardware-protected environment, like a Hardware Security Module (HSM). As a standard Certificate Signing Request (CSR), the device transfers the device-specific CSR data within a secure environment to the CA. The CA generates the certificate by signing it with its CA private key. Note that this step includes the insertion of manufacturer, model and device serial number into the certificate. A trusted entity then transfers the certificate back to the device. The result is the existence of a valid device certificate with a corresponding private key. After this step, the device is ready for communication with the backend and hence for personalization. The backend only accepts devices with properly signed certificates and is thus protected against device spoofing.

A further possibility to craft the certificate is to generate the device certificate in a trust center. Hereby, the trust center generates the (passphrase protected) software-based cryptographic token containing the public key pair and gets it signed by the CA. As a next step a privileged user transfers the software token back to the device, where it is unlocked. The problem of this solution is the rather insecure transfer of the cryptographic token including the private key compared to generating it directly on the device with a HSM. However, for devices without HSM this solution is preferable, as the key generation is rather insecure on a field device without HSM.

3.4.3.3 Personalization with the Secure Element

Having the device equipped with the device and CA certificates, as well as with the device certificate's private key, the next step is to add new users to devices. The purpose of this is to associate users with containers in later steps, such that users are able to decrypt or create new containers. The trust center sets up a new user and creates a user certificate (signed by the CA) including its private key. This step also involves either transferring the

generated private key to an SE, for example, the user's smartcard and to securely transfer it to the user, or to directly generate the key in the SE. Next, the user certificate is synced from the MDM to the device. For existing users in the system, syncing their certificate to the device is the only step.

3.4.3.4 Container Generation with the Secure Element

A user registered on an enrolled device is capable of generating disk-encrypted containers. In the first step, the required strong symmetric key for the new container is created on the SE and encrypted with the user's public key. The encrypted version of the symmetric container key is stored on the device. The plaintext key is only present in main memory when the SE is unlocked and used to unwrap the encrypted version with its private key. Hence, the decryption key for the symmetric container key is solely stored in the SE of the corresponding user. In the second step, the device then creates a new container image and encrypts it and the container configuration file with the symmetric container key. After this process, the container is created, and the device optionally sends the encrypted container key and container data to the MDM as a backup. Since the MDM is not in possession of private keys of users, it can never decrypt this information.

3.4.3.5 Linking Users to Containers

Having user certificates and containers on the device, further users can be associated with containers and privileged users can create containers on the device. A privileged user having created a container beforehand can add another user to a container. This privileged user is the only instance to do this, as only this user holds the private key for decrypting the container's symmetric key. The privileged user creates the newly encrypted container key from the decrypted symmetric container key with the new user's verified certificate public key. This new key is added to the decrypted containers' keys on the device. Note that this process can be compared with adding keys derived with a key derivation function to the key slots in Linux Unified Key Setup (LUKS) headers. After that, the new user is able to decrypt the symmetric container key, and hence the container image and configuration, with his own private key.

3.5 Design and Application of the Secure Architecture for IoT Scenarios

In Section 3.3, we presented the secure virtualization architecture for mobile devices, and embedded the architecture into an ecosystem with end users in Section 3.4. In this section, we develop an industrial *trust ecosystem* for the industrial internet of things, and transform our virtualization architecture and device lifecycle to hardware platforms representative for such ecosystems, see Contribution 4. With this, we show that the virtualization architecture is not tied to the mobile device domain, but can also be leveraged for different fields of application addressing Challenge 1 and Challenge 2. While the key challenges, such as isolation of critical components to protect data confidentiality, or provisioning, enrollment and secure operation, remain the same between the two application fields, new solutions are required for solving these challenges within Industrial IoT (IIoT) scenarios. In the remainder of this paragraph, we motivate the industrial application field and a concrete instantiation of our trust ecosystem, the so-called Industrial Data Space (IDS) to emphasize

the differences from the mobile device domain with end users. In the end, we provide an outline for the section.

The modern IIoT computing landscape is shaped by emerging use cases that rely on interconnecting networks and components that used to be isolated [Ope; Ott16; Pló16]. This means that valuable corporate data is exchanged over the interconnecting networks, which are oftentimes characterized by the presence of a large number of interconnected devices without end user interaction. Use cases in the IIoT domain are often associated with the term *Industry 4.0* or CPS where sensor nodes or automation actuators form a substantial part of the IIoT use case, gathering data and actuating based on the gathered or exchanged data. The data produced and exchanged in such scenarios constitutes an important part of the underlying business processes and is therefore a valuable core asset by itself. With the demand for this data to flow across different trust domains, new security challenges arise.

One of the emerging use cases in the IIoT domain is the IDS, a data exchange platform composed of interconnected devices, which gather, process, and exchange data [Ott16; Ott17]. The goal of the IDS is to go beyond traditional scenarios, where few partnering organizations knowing each other beforehand created a common trust domain and exchanged data in a closed system. The IDS creates a common data space that spans multiple sectors with different needs and standards, bridging these sectors and allowing for new use cases, workflows and business models. The software that realizes the functionality required for the IDS forms the concept of the *IDS connector*. IDS connectors can be deployed on different interconnected devices part of the IDS ecosystem. Each connector runs different *services*, possibly from third parties, used for the exchange and processing of sensitive data in the distributed network of IDS connectors. To create trust in complex and decentralized ecosystems like the IDS, it is mandatory to design a solid yet flexible security architecture for the ecosystem and its components.

We design a holistic security architecture for these kinds of ecosystems and connectors. Our architecture allows for secure and controlled exchange of the sensitive data between the interacting connectors in a decentralized, untrusted environment. We keep the design of the trust ecosystem generic and thus abstract from the IDS use case. We call the interconnected devices of the trust ecosystem the *trusted connectors*. The abstraction makes our architecture suitable for general use in IIoT scenarios. We start with defining and assessing security requirements for the trust ecosystem with its data flow and entities. Based on these requirements, we establish our overall security architecture for the ecosystem. We provide a flexible identity management concept for the whole ecosystem including fine-grained data access control, remote integrity verification of connectors, and the ability to enforce data usage policies over the data lifecycle. As the central part of the ecosystem, we transpose our OS-level virtualization-based security architecture from Section 3.3 for use within the trusted connector, The virtualization architecture isolates services with their confidential data from each other. We realize a full open-source implementation of the architecture [Fraa; Ind].

This section is organized as follows. Section 3.5.1 provides a high-level overview of the IDS ecosystem and its entities. In Section 3.5.2, we define the main security requirements for our trust ecosystem. We describe the security architecture of the trust ecosystem

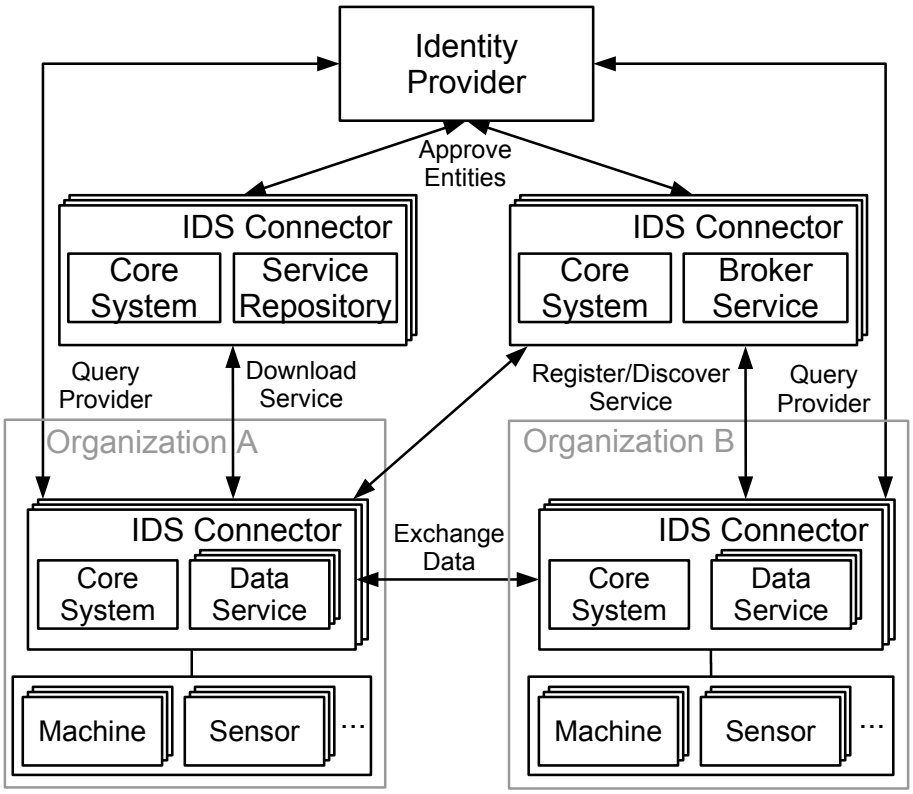


Figure 3.11: Functional roles of the essential entities in the IDS.

in Section 3.5.3. The architecture of the trusted connector is provided in Section 3.5.4. We present a secure communication concept for trusted connectors in Section 3.5.5 and focus on data usage control in Section 3.5.6. Section 3.5.7 covers the trusted connector implementation. For a security discussion of the architecture to evaluate it against the local and remote attacker from Chapter 2, we refer to Section 3.3.5.

3.5.1 Industrial Data Space Overview

In the IDS ecosystem, participating organizations buy connectors from approved device vendors to operate their own connectors. This is coupled with a *service repository* concept, where IDS repositories offer services for download to connectors. Figure 3.11 depicts the functional roles of the most essential entities taking an active part in the communication within the IDS with the example of organization A and B exchanging data. We describe these entities in the following.

IDS Connector. The IDS connector operates the services, which produce or consume data, or which provide an ecosystem management functionality. An example are connectors deployed on *edge devices* in a manufacturing unit or in a cloud center. Each connector has a core system, which is a service integrating the connector to the IDS ecosystem and to manage other services. The data services in case of organization A and B gather, process, and exchange data, such as manufacturing data from connected

hardware or sensor arrays. In addition, services on the core system can represent infrastructure services for the management of the IDS ecosystem, such as for service downloads or service discovery. As the infrastructure functionalities of the IDS are deployed in the form of services on IDS connectors [Ott16], they are the defining element and the central part of the ecosystem.

Broker. The broker service running as part of a connector, see *Broker Service* in Figure 3.11, serves as a directory for all services deployed onto connectors. For this purpose, the broker stores the metadata of known services, for example, service descriptions and policies. In order to make itself known in the network, every service on a connector must register itself at a broker service. The service on a connector can then be discovered by other connectors querying the broker, for example, enabling organization A to discover a data service of organization B.

Service Repository. This entity serves as a download repository for services. Service repositories are services themselves, see *Service Repository* in Figure 3.11, and are locatable by connectors using the broker service. Organization A can use an appropriate service downloaded from a service repository to retrieve data from a corresponding service of organization B. Broker services and service repositories themselves form a decentralized network, in turn.

Identity Provider. The identity provider manages all identities of the involved entities part of the ecosystem. Services can query the identity provider to gain information whether a service they connect with is a legitimate entity in the ecosystem. Note that an identity provider can also be represented by a service on a connector, for example, hosting a service offering revocation statuses of connectors.

In addition to the described entities, the IDS ecosystem is based on several administrative units, which do not participate in data exchange between trusted connectors. Due to the heterogeneous structure of such a loosely coupled network like the IDS, strong security requirements exist for the connectors, their interaction and the sensitive data they exchange. This requires a concept for the root of trust and a clear definition of trust boundaries, as well as the definition of secure gateways, the IDS connectors, between those trust boundaries.

3.5.2 Definition of Security Requirements

In this part, we define the main security requirements for the design of our trust ecosystem and trusted connectors, categorized according to the upcoming design sections 3.5.3 to 3.5.6. We first summarize the motivated specifics of IIoT scenarios to substantiate the security requirements to be defined for the trust ecosystem. In the following, we refer with the term trusted connector to both its core and service software functionality and to its underlying platform as an integral unit.

Compared to the previous ecosystem designed for smartphones, ecosystems considered here are no longer centered around end users, but rather comprise of spatially dispersed, embedded devices without direct user interaction. Data valuable for corporations is processed on the devices and heavily shared, oftentimes with other corporations. The

perspective shifts from privacy of user data to high-value corporate data with usage control requirements, and as a result to controlling all software running on the devices which ensures that none of the data may be used illegitimately. A further result is that the valuable data transferred between devices needs special protection and mutual trust between devices and their software configurations. With these requirements, the operator of the ecosystem plays an important role, responsible for controlling the software running on the devices, for the data they store and that flows in the ecosystem.

Based on the presented specifics and considering our attacker from Chapter 2, we thus define requirements on the operation of the trust ecosystem, trusted connectors, their communication and data usage. In the IIoT ecosystem, our attacker may have physical access to trusted connectors, is able to compromise services running on the connectors and may interact on all communication channels in the ecosystem and spoof parties.

Trust Ecosystem, exposed to untrusted third parties and software, which the operator needs to be able to regulate:

TE-I The operator must be the exclusive authority in the ecosystem to approve and reject parties, such as service developers, assigning them fixed roles.

TE-II Services may only originate from approved sources in service repositories. Prevents untrusted parties from introducing software and repositories, which might, for instance, leak data.

TE-III Security analysis and quality assessment for services to be offered in service repositories, reduces the risk of exploitable vulnerabilities and backdoors.

TE-IV Licensing/certificates for service instances installed on connectors, allowing only approved service instances the interaction with other services.

Trusted Connectors, running possibly insecure or untrustworthy third-party services, which can be leveraged or exploited by attackers:

TC-I Enforcement of service integrity before download and start, only verified services may run.

TC-II Strict isolation for all running services, prevents compromised services from unintended data access.

TC-III Resource limitation for services, such as CPU usage. This prevents a service from exhausting other services' resources.

TC-IV Fine-grained data access control for services, allows to define data usage policies that ensure services access only intended data.

TC-V Confidentiality for data stored by services, prevents attackers with physical access from obtaining data.

Trusted Connector Communication, possibly eavesdropped, altered, or spoofed by attackers on the communication channels:

CC-I Verifiability of the integrity of the software stack of connectors before allowing a connection. As confidential data is transferred through connectors, this requirements ensures that only verified connectors may exchange data.

CC-II Authenticity, confidentiality, and integrity. This prevents an attacker on the communication channel from gathering and modifying data.

Data Usage. Exchanged data can possibly be disclosed by services on other connectors:

DU-I Attachment of usage policies to exchanged data.

DU-II Enforcement of usage policies defined by data owners.

DU-III Logging and accounting of all relevant data use.

At the end of each of the following design sections, we refer back to the corresponding security requirements category summarizing how we achieve the requirements.

3.5.3 Trust Ecosystem Architecture

This section presents the architecture of the trust ecosystem in response to the defined requirements. We first focus on the entities and then on our PKI for managing their digital identities.

3.5.3.1 Ecosystem Entities

Figure 3.12 depicts the different entities of the generic trust ecosystem and their relationships, described in the following. These entities can, for instance be mapped to the IDS, but also to other scenarios.

Ecosystem Operator

The operator has the responsibility to approve new parties to become part of the trust ecosystem. The operator determines the role of each new party, such as representing a service repository, device vendor, service developer, or service authority. Only if granted by the ecosystem operator, these entities may introduce components to the ecosystem, for example, new services, or devices. To enable the operator to establish trust for protecting the parties of the ecosystem and to ensure that they comply with their roles, the operator manages a PKI, see Section 3.5.3.2. The operator enforces PKI functionality in the form of the identity provider, see Section 3.5.1, for instance, offering CSR signing functionality, OCSP services, or CRLs.

The operator may, for instance, be an association, or an operating company, which can be a single unit, but also a consortium. For the sake of simplicity, we refer with the term *operator* to a centralized, singular unit in the following. However, the operator may also represent a decentralized instance in practice.

Device Vendor

The device vendor retails or produces trusted connectors and is responsible for their initial provisioning with a software system and trust anchors. This allows connectors to automatically enter the ecosystem when deployed by an organization. After the initial provisioning, the vendor ships the trusted connector in a completely enrolled and preconfigured state.

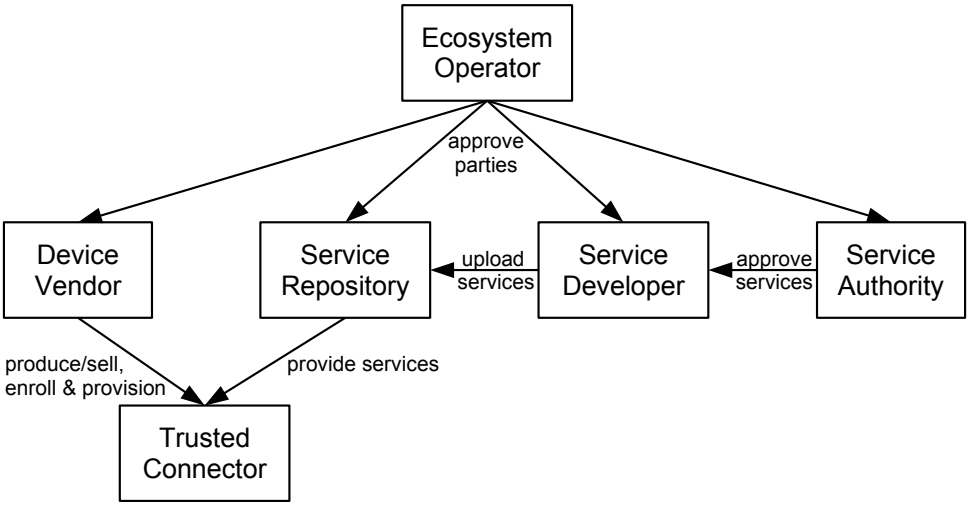


Figure 3.12: Entities and their relationships in the trust ecosystem.

Trusted Connector
The trusted connector is a hardware device that is running the software stack allowing to securely interact inside the ecosystem. We allow a connector to communicate with other connectors only if it origins from an approved device vendor, operates approved services only, and if it is capable of proving the integrity of its software stack.

Service Repository
Service repositories provide trusted connectors with approved services. Connectors only accept services from repositories approved by the ecosystem operator. Service repositories are also responsible for managing licenses of the services. We only allow services with a valid license to run on the connectors. The service repository also provides fingerprints of legitimate service versions and service licenses for mutual verification prior to connector communication, see Section 3.5.5.

Service Developer
Only approved service developers may develop services for connectors. After implementing a service, developers must sign their service. Then, developers must inquire the service authority for service approval. The service developer can only upload services to the service repository when approved by the service authority.

Service Authority
The service authority ensures the quality and security of services as criteria for approval. Every service is subject to analysis and evaluation performed by service authority before approval or rejection.

3.5.3.2 Public Key Infrastructure

The ecosystem operator establishes mutual trust between the parties based on a PKI and issues or revokes certificates for the entities. Figure 3.13 depicts the PKI structure for

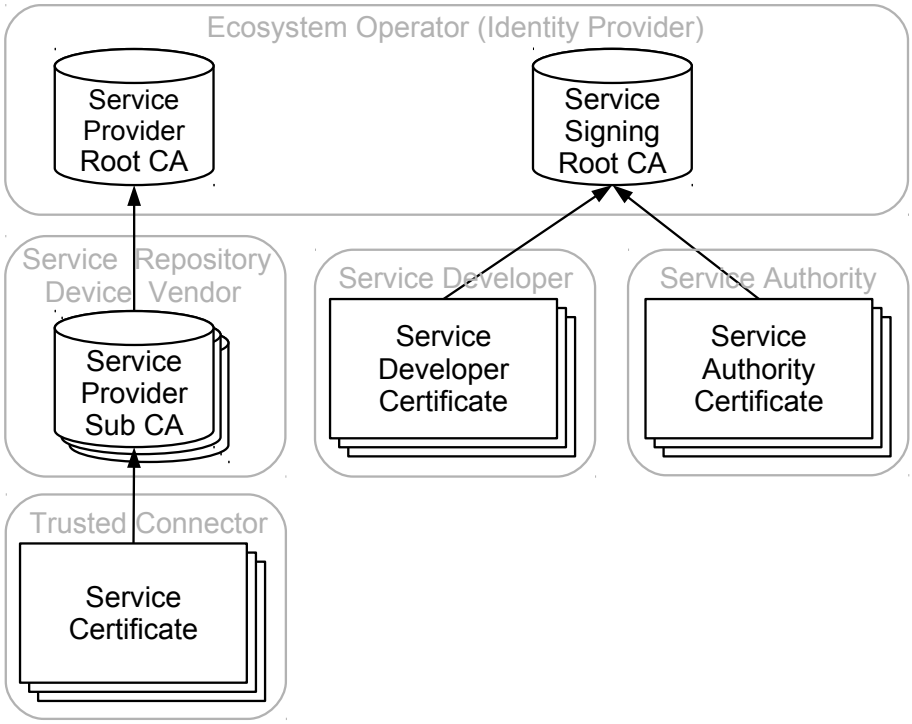


Figure 3.13: The PKI hierarchies of the trust ecosystem.

the trust ecosystem, which reflects the control relationships between the entities. Our structure has two separate PKI hierarchies managed by the operator, the service provider and service signing CAs.

Service Signing

The service signing hierarchy is for managing the service developers and authorities, which use their certificates to sign and hence approve services prior to distribution. Service developers and authorities receive their certificate after approval by the operator. The role and properties of each entity are provided in the certificates' attribute fields. Services are only valid with two signatures, each from one of the both entities.

Service Provider

The service provider hierarchy is for the device vendor and service repository. The device vendor and service repository both can leverage or operate a sub CA in the hierarchy of the operator's root CA. The device vendor uses the sub CA to sign particular instances of services to for initial deployment on a device. Similarly, the service repository uses the sub CA to sign the particular service provided to a particular device. Instances of services are thus represented by their service certificate, serving as a license. Trusted connectors verify whether downloaded services have valid service licenses before deployment and refuse connections with connectors with expired or revoked licenses. The connector can refresh expired licenses using the service repository.

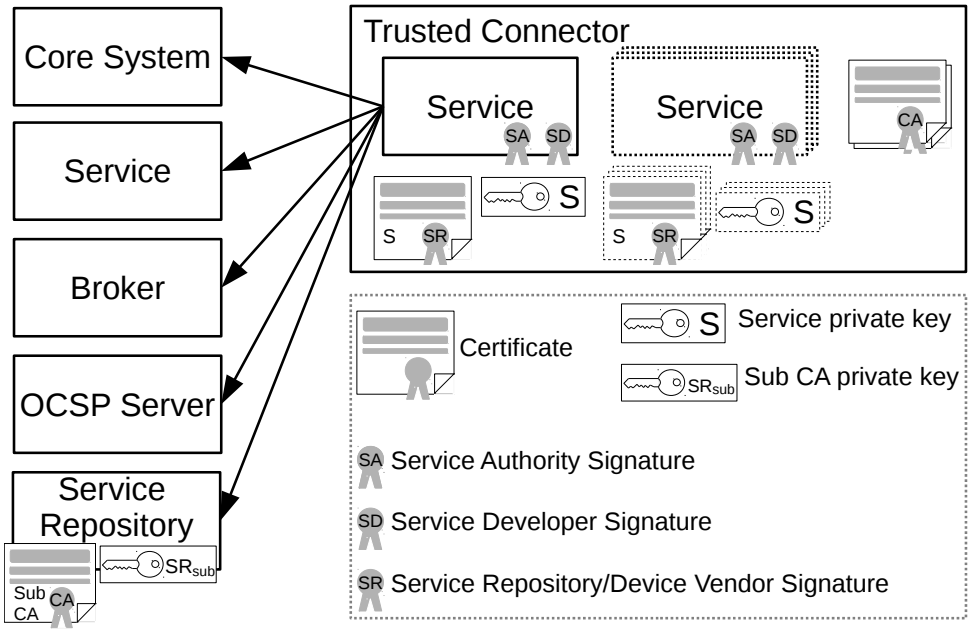


Figure 3.14: Service types with certificates and trust anchors on trusted connectors.

3.5.3.3 Service Types and Distribution

Figure 3.14 provides an overview of the service types, certificates and trust anchors deployed on the trusted connectors. The illustration emphasizes that each service has a digital signature from the service developer and from the service authority. Additionally, each service instance has a service certificate signed by the service repository or device vendor.

Depending on the different types of deployed services, a trusted connector may represent a special role in the ecosystem. This means that a service repository is a connector running a special service type, same as a broker or an OCSP server. The service repository type has its sub CA certificate and key, used for creating service certificates. This allows to provide services with a valid license. The core system is a service type present exactly once on every connector. The core system manages the connector, its services and the connections with other connectors. The core system can verify service certificates using the CA certificate as trust anchor, initially deployed during provisioning by the device vendor.

The provisioning of trusted connectors includes the installation and configuration of the core system, and issuing the initial services' certificates. The customer can still use the pre-installed core system to exchange it with another core system, or replace trust anchors, remove and install further services. Figure 3.15 illustrates the steps for setting up and starting a newly developed service on a trusted connector.

1. The service developer uses its private key corresponding to the service developer certificate to sign newly implemented services. The developer sends the signed service to a trusted service authority for approval.
2. The service authority verifies the developer's signature of the submitted service and performs its approval process with regard to the software quality and security. In

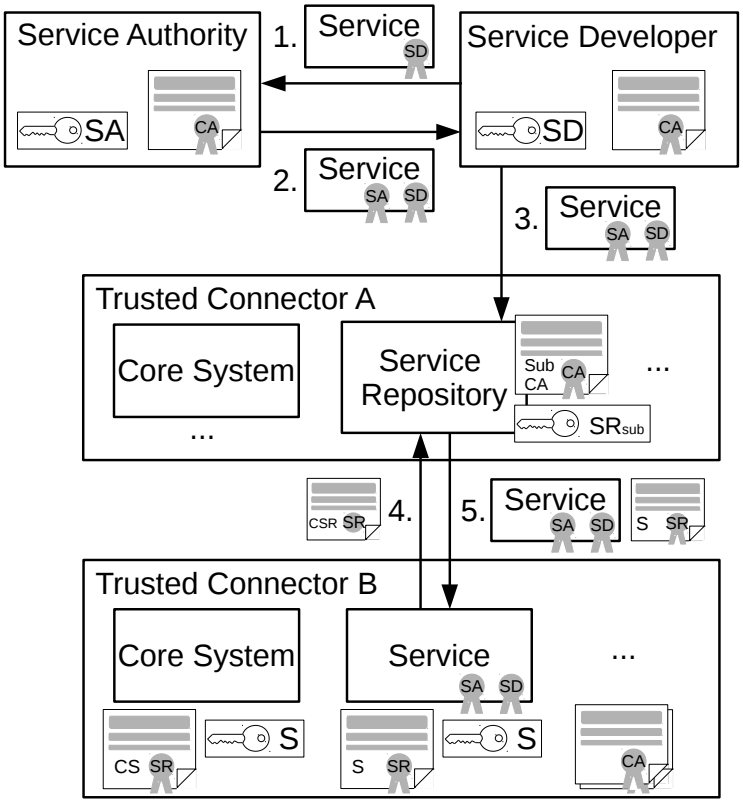


Figure 3.15: Service lifecycle from development to deployment on trusted connectors.

case of successful evaluation, the authority signs the service and sends it back to the developer.

3. The developer uploads the service to a service repository running on trusted connector A. The service repository accepts the service with the two valid signatures and publishes it.
4. Trusted connector B can download the new service from the service repository. To issue a service download request, connector B creates a service key-pair and a CSR and sends the CSR to the service repository. The service repository signs the CSR and issues a service certificate representing the service instance and license for connector B.
5. Trusted connector B then downloads the new service and certificate. The connector checks the service developer and authority signatures, as well as its own certificate for the received service using its deployed trust anchor. The connector is only able to use the service for remote connections, with a valid certificate. Otherwise, interacting connectors will refuse to connect, see Section 3.5.5.

Summary

The ecosystem operator is in full charge of the PKI infrastructure, making it possible to approve and revoke every party from the ecosystem and to assign them fixed roles (TE-I).

Each service in the service repository carries digital signatures by an approved service developer and service authority, which guarantees that services originate from trusted sources only (TE-II). The security and quality analysis by the service authority ensures all services in the service repository are assessed (TE-III). Additionally, each service installed on the trusted connector receives a certificate from the service repository. The certificate's validity period defines the service license and ensures that only licensed services may exchange data (TE-IV).

When enforcing the integrity of the software loaded on the trusted connectors such as OS kernel and services as referred to by TC-I, the trusted ecosystem may form a *closed ecosystem*, for instance comparably to the system architecture of Apple's iOS. All software intended for running on the trusted connectors has to be approved by the operator, and the trusted connector enforces from early boot to service start time that only such approved software is loaded (and sandboxed when achieving TC-II). The ecosystem operator is thus the only party to approve services and service developers in the ecosystem. With the introduction of service authorities, however, the ecosystem operator may delegate service approval. This allows different service authorities to introduce services - but not privileged software components such as an OS kernel - into the ecosystem. On the trusted connector side, this further allows to enforce that services originating only from a specific authority may be loaded. It depends on the enforcement of the operator in an instantiation of the trust ecosystem whether the system is considered closed or open. The IDS is designed to be an open ecosystem. For IIoT use cases where the software running on connectors plays a crucial role for data usage and thus confidentiality, at least certain control over service origin and quality must be enforced.

3.5.4 Trusted Connector Architecture

Figure 3.16 depicts the security architecture of the trusted connector. The design of our architecture enables trusted connectors to run their services in containers similar to Section 3.3. The illustration shows the OS-level virtualization-based architecture. Like in Section 3.3, we only allow the containers to communicate over strictly defined channels and assign them minimal privileges only. Despite that our trusted connector architecture generally follows Section 3.3, the overall design changes for IoT devices. In contrast to the usage model on mobile devices, where the end user plays a crucial role, functionalities and interfaces for end users are not an integral part of our architecture, for instance. This section briefly describes the main components of our trusted connector architecture bottom-up starting with the hardware layer.

3.5.4.1 Hardware Layer

Compared to a mobile device use case, where full Android containers share the various hardware devices, such as the radio interface, the Wi-Fi chip or bluetooth interface, the services in our ecosystem are mainly designed to access and process sensor data. The hardware devices on the trusted connector used by services are thus interfaces for the connection of sensors and actuators to the connector, such as the network interface. To provide a root of trust for the software stack and for the attestation between connectors, we integrate a TPM into our architecture. To verify the integrity of software loaded at

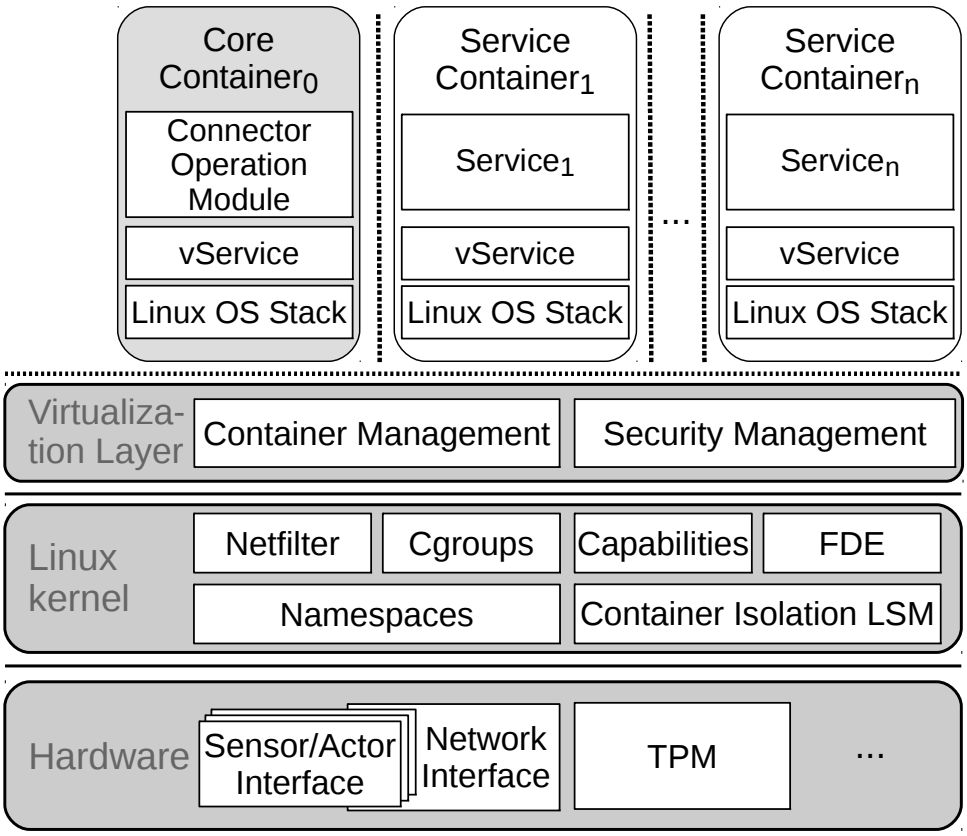


Figure 3.16: Trusted connector architecture for service isolation based on OS-level virtualization.

boot time, trusted connectors must provide a root of trust for realizing a chain of trust implementation, such as UEFI secure boot.

3.5.4.2 Kernel Layer

The features to realize the separation with OS-level virtualization form the Linux kernel's namespaces feature and parts of the cgroups feature. The other building blocks are responsible for the isolation of the containers. We briefly explain how we use these blocks in the following.

Namespaces

Each container runs in a separate namespace like in the virtualization architecture for mobile devices. For instance for regulating network device access between containers, we use the network namespaces. We can re-use the device namespaces feature to virtualize the access of containers to further sensor and actuator interfaces connected other than via the networking interface.

Cgroups

Like on the mobile device architecture, we use the CPU and memory subsystem to restrict containers to a fixed share preventing containers from exhausting these resources. We also make use of the cgroups devices subsystem. We restrict containers to the access to hardware devices, the sensor and actuator interfaces, depending on the specific service they host. This makes it possible to restrict containers from accessing any device or interface except for those a service is designed to use, for example, to gather data from a network-connected temperature sensor.

Container-Isolation LSM

With our container-isolation LSM from Section 3.3.2, we restricted containers from making uncontrolled system calls, for instance, and to protect objects relevant for container isolation. We use this LSM and tailor it to our platform.

Capabilities

We re-use the capabilities feature from Section 3.3.2 to restrict containers to the minimal set of their necessary permissions. Depending on the purpose and privileges of the containers, we drop not necessarily relevant capabilities.

FDE

The mobile device architecture used the Linux kernel's FDE infrastructure to transparently encrypt all the Android containers' persistent service data. The container storage protection key was bound to an SE, such as a smartcard, to prevent brute-force attacks on the encryption key. The user had to provide the SE's PIN via the user interface to unlock the SE. User interaction and user authentication are not part of the model on our devices. Therefore, we make use of the TPM as secure key storage. We use the sealing functionality to bind the storage protection key for containers to the TPM and ensure that it is unsealed on container start only in a known-good system state.

Netfilter

In contrast to the use case for mobile devices where containers must access the internet unimpeded, our architecture confines the network traffic of containers to specific network addresses. We make use of the netfilter Linux firewall to ensure that containers use the network interface only to communicate within the boundary of our ecosystem. A data service may, for instance, be restricted to only gather data from a sensor in the local network via a specific route.

3.5.4.3 Virtualization Layer

The virtualization layer in user space is part of the TCB and hence a privileged component. Its integrity is verified at boot time by a chain of trust implementation. This layer splits up into the entities CM and SM.

Container Management

Like in the prior mobile device case, the CM is responsible for starting and stopping of containers. Before starting a container, the CM uses the SM to verify the signatures of the container to be started Section 3.4. Then, the CM configures and activates the kernel's

security and virtualization mechanisms associated with the container. An example is to enforce the cgroups devices subsystem for a container to be started, such that access is granted to only the specific hardware devices the contained service is allowed to use. When starting a container, the CM mounts its partitions and embeds the container into its own namespace. The CM is also capable of updating containers and the virtualization layer. The CM applies new updates for containers and the virtualization layer only when the signatures match against the CA certificate. The core container is responsible for providing the CM with the updates.

Security Management
The SM performs the cryptographic operations for the container's FDE obtaining the FDE key from the TPM prior to container start. Furthermore, the SM performs the signature checks on all read-only container images, which are signed prior to distribution, before allowing the CM to start a container. Another task of the SM is to make use of the TPM's remote attestation functionality for establishing communication channels with other connectors before exchanging confidential data, see Section 3.5.5.

3.5.4.4 Container Layer

The software layer on container level consists of a number of untrusted service containers and one higher privileged core container.

Service Container
Each service container runs an isolated service or a service bundle in a separate execution environment based on a generic Linux OS stack. The architecture confines the service to only communicate with the CM and with other services of remote connectors after the core container established the communication and set up the channel for the container. The vService entity, also available in the core container, is used by the CM to communicate with containers. For example, the vService sends a message to the CM after startup to indicate the successful completion of the boot process. The CM also uses the vService to initiate a container shutdown.

Core Container
In the mobile device architecture, a responsibility of the core container was to securely virtualize those hardware devices, which could not be virtualized in kernel space using the device namespaces. Devices, such as the Wi-Fi or radio interface, have parts of their driver base realized in user space, for instance, in the form of background daemons. We re-use this principle and existing infrastructure to virtualize the access for such sensor-actuator interfaces. We replaced the Android OS userland stack with a minimal GNU OS user space stack keeping the attack surface of the privileged connector small. In the trust ecosystem, the privileged core container bundles the management functionality for service containers and provides the integration into the ecosystem. This is the responsibility of the core system service, called Connector Operation Module (COM), described in the following.

3.5.4.5 Connector Operation Module

We designed several software components on a generic Java stack representing the functional blocks of the COM. For instance, the *Trust Ecosystem Secure Communication Protocol (TESCP) endpoint* functionality represents the connector-to-connector protocol integration point for the trust ecosystem. We designed the TESCP to realize remote attestation, metadata exchange and policy handshaking prior to data exchange between containers, see Section 3.5.5. Another example is the *service manager*, which uses the CM interface to manage service lifecycles, i.e., to start, stop, deploy, or remove containers. The service manager also provides runtime information to the administration interface component. The *routing manager* controls data routes between deployed service containers and internal and external devices, like sensors or external connector instances and services. The COM sets the routes of the containers' network to grant or prevent services from network access to sensor and actuators. Services are not reachable from the outside until a valid route has been declared by the routing manager making it the central configuration point for connectivity. This is possible, because the COM, running in the privileged container, shares the network namespace with the CM. The *dataflow control* component allows for creation and monitoring of data flow policies that enable usage control to be performed by the routing manager.

Further details on the COM's design and implementation based on the Open Service Gateway Initiative (OSGi) framework [OSG] can be found in [Bro18]. All the components are bundles running in OSGi, which we can dynamically bind or replace whenever there is a need for change of functionality. Our COM implementation is part of the *Trusted IoT Connector* open-source platform [Ind].

Summary

We achieved the service integrity (TC-I) with a chain of trust implementation and by verifying and only loading signed container images. This reduces the risk of uncontrolled drain of confidential data. The container isolation with the Linux kernel security mechanisms achieves the strict service isolation between containers (TC-II). The kernel security layer also achieves resource limitation (TC-III) and access constraints (TC-IV). In particular, the management of critical resources and accesses is configured by the core container. The confidentiality of the data of services (TC-V) towards attackers with physical access is achieved with the storage protection.

3.5.5 Connector-to-Connector Communication

In this section, we describe the model for secure connector-to-connector communication, the TESCP for our trust ecosystem. The COM is responsible to enforce this model for every connector communication attempt on behalf of the isolated services. A successful access control decision is the precondition for data exchange between services. Figure 3.17 depicts the process for connector A establishing an active channel with connector B, as described in the following.

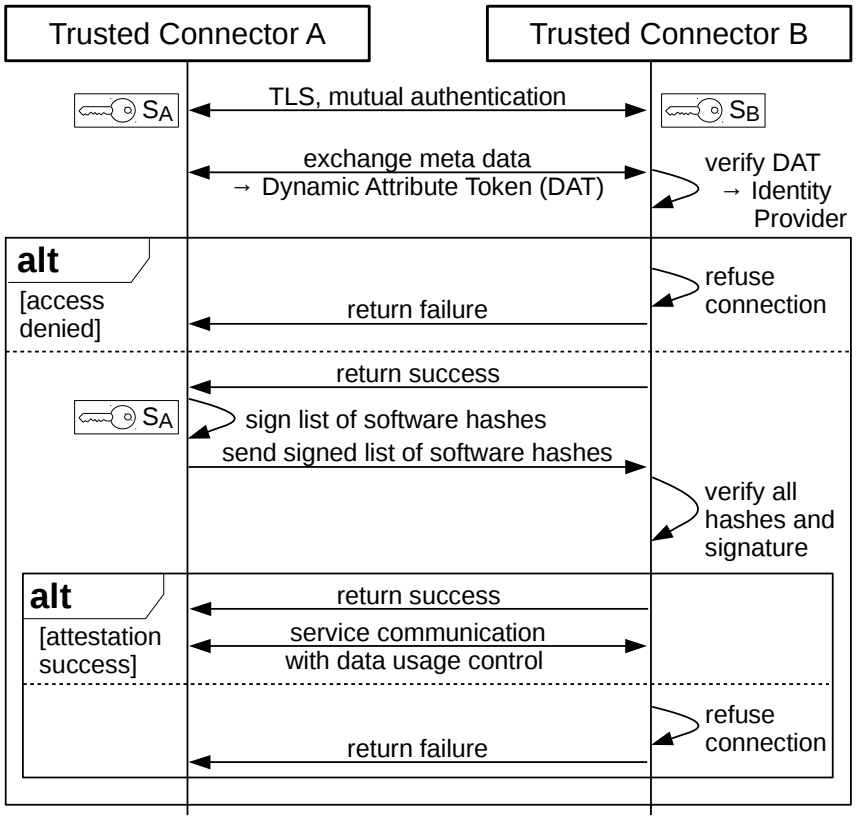


Figure 3.17: Simplified secure communication protocol between trusted connectors.

Secure Channel

Both connectors first establish an authenticated TLS tunnel with mutual authentication. For that purpose, the COM, as a service, uses its service certificate. The certificate must be signed by an approved entity, neither expired nor revoked, see Section 3.5.3.1. The connectors contact the identity provider to query the revocation status of the other connector. Thereby, the identity provider, for example, operating an OCSP server, returns whether the certificate has been revoked or is still valid. In scenarios where OCSP services are offline, certificates must be checked against a local, updated CRL. After successful tunnel creation, the connectors exchange their metadata in form of dynamic attributes. These are represented by so-called Dynamic Attribute Tokens (DATs) signed by a service operated by the identity provider. The token format could be, for instance, an OAuth or JSON web token format. In Figure 3.17, connector A presents a DAT to connector B, which verifies the validity of the DAT using the identity provider. The DAT contains current identity attributes, for example, describing a specific domain or business area, or the connector's additional security features, for instance, a secure server room. Connector B can then allow or deny access based on the contained attributes. We decided to use this dynamic attribute approach over static attributes tied to service certificates, because the attributes can change over time. For instance, if the connector is moved from the secure

server room to a public location, it looses that attribute. In case of static attributes, the COM's service certificate must be invalidated and a re-approval performed.

Remote Attestation

In the next step, both connectors must provide proof of the integrity and validity of their software stacks. The attestation considers the core system, as well as all installed services along with their certificates. This step is referred to as remote attestation, which we based on the TPM 2.0. Note that Figure 3.17 simplifies the procedure by showing only attestation of connector A.

We base the attestation of connectors on the measured boot process proposed by the Trusted Computing Group (TCG). With the Core Root of Trust for Measurement (CRTM), we verify the boot firmware, for example, BIOS, and extend the verification chain to the OS. We then ensure the integrity of the virtualization layer and the core container. To be able to cope with software updates, we use the custom policies specified by the TPM 2.0 delegating the decision about the legitimacy of particular Platform Configuration Register (PCR) values to third parties. In the trust ecosystem, these are the service repositories, to which connectors provide the relevant quote of the PCR values signed by the TPM's private key. We simplified this step in Figure 3.17, omitting connector B contacting the service repository.

The service repositories not only provide the knowledge about the fingerprints of the core container and virtualization layer, but also about approved and rejected fingerprints of all other services. The verification of the measurements of services by the service repository requires the connectors to provide a measurement log in addition to the quote of the PCR values. This log contains fingerprints of installed services along with their metadata. Connectors query the service repository to reconstruct the sequence of service installations and PCR updates using the measure log. The service repository then verifies the correctness of the PCR value. The service repositories relieve connectors from maintaining a whitelist of allowed software by themselves and preserve connectors' privacy, because connectors only have to be provided with service fingerprints. Additionally, the connectors contact the identity provider to validate each service's validity regarding its license represented by the service certificate. We also omitted this step from the illustration. Only when a connector meets all conditions, we allow the interaction between services.

Summary

The communication requires mutual remote attestation between connectors based on the TPM 2.0 to verify the integrity of their software stacks before exchanging data (CC-I). Communication between trusted connectors is performed through TLS tunnels with mutual authentication based on service certificates issued by the ecosystem operator (CC-II).

3.5.6 Data Usage Control

One key aspect of the IDS use case is the promotion of *data sovereignty*. On the one hand, this means that the owners of data can be ensured that data kept locally is protected on their premise with solid security measures, while allowing them control over data usage. On the other hand, this also means freedom of choice to define usage and access rules for data in transit at the owner's discretion. The protection of confidential data is thus not only a

fundamental aspect when it comes to protect devices or communication channels, but also the further usage of data by other connectors. Data usage control addresses this aspect. In the context of the trust ecosystem, the ability to control fine-grained message-based data usage is important to avoid drain of sensitive data by trust connectors. Services may accept input data records, process them and generate a new set of data records to be output. The data services output may be sent across trust domain boundaries, i.e., to other corporations. We attach data usage policies to all data connectors exchange. With this, we restrict the flow and usage of data. For example, this allows to anonymize data before sharing with other connectors.

For defining data usage policies, we assign the data and the services *predicates*, which serve as labels and descriptors for service properties. The predicates may be simple tags, such as data being labelled *personal*, or represent data usage conditions, such as a time-to-live before the data record must be destroyed at the recipient. Data gets first labelled at its origin, depending on the service collecting the data record. Services with special properties, such as data anonymization or filtering of contents, then remove the specific labelled information in data records before their transmission.

We define data flow policies to model the possible data flows in the ecosystem. We base data transmission on predefined routes, based on which we model data flows and enforce flow policies. To realize the data usage control for the trusted connector, we designed a policy framework that enforces label-based usage control. This framework builds on the work in [Sch16]. The key aspects are the following:

1. Definition of a policy language that we translate into a first-order logic representation.
2. Introduction of a component responsible for the runtime evaluation of policies based on their first-order logic representation.
3. Static model checking of message routes against policies.

The framework allows to create policies, which are easy to administer and supports attaching usage policies to connectors to restrict data flow and usage. Our container-based isolation architecture enforces the local compliance with the policies. The attestation capabilities in the trust ecosystem allow the verification of deployed policies on other trusted connectors. Further details on data usage control for the trust ecosystem and the implementation on trusted connectors can be found in [Bro18].

Summary

Each communication message is appended a label by the COM with a set of predicates, which specify a data usage policy associated with the transmitted data (DU-I). The COM of the respective connectors enforces the data usage policy (DU-II). The COM provides the functionality for data audit logging and log signing (DU-III).

3.5.7 Implementation

We implemented a prototype for the trusted connector based on the implementation of the virtualization architecture in Section 3.3.6, both to be found in the open-source project trustm3 [Fraa]. Instead of an Android build chain, we based our build chain on

the Yocto project [Sal14] to ensure high portability of the connector's software layers to a variety of hardware architectures and application scenarios. The corresponding build environment includes Linux kernel configuration as well as bitbake recipes for the software components. Furthermore, several build configurations are provided, producing the system images for the different containers and the base system. With this build setup, we successfully executed and tested the prototypes on various x86, ARM, and PowerPC architectures. Following Figure 3.16, we describe the most important aspects for software layer components bottom-up. We then focus on the communication between connectors.

3.5.7.1 Kernel Layer

On kernel layer, we ported the necessary features from the 3.10 Android kernel from Section 3.3.6 to kernel version 4.4 and built it for the x86 architecture. The kernel starts the CM and SM as privileged processes after kernel boot completion.

On our current connector prototype, we could abstain from device namespaces. First, the foreground-background usage model does not exist. Instead, the containers all run in parallel and do not require simultaneous access to hardware devices, such as a touchscreen. Second, the sensor/actuator interfaces were all connected via the network interface, which we could virtualize with `veth` interfaces and container-specific routing. When other interfaces or hardware devices for accessing sensors and actuators are used, the multiplexing can be enabled using device namespaces. To prevent access to other hardware devices than the network interface, we restricted the access using the cgroups devices subsystem.

3.5.7.2 Container and Security Management

We implemented the CM as a Linux daemon close to Section 3.3.6. The CM is responsible for the basic container network setup by creating a `veth` pair in the core and service container when a new container starts. The core container and CM share the same network namespace. The network namespace sharing and set-up of `veth` pairs allows the core container to set the network routes for the service containers using ipconfig and the netfilter firewall. The CM sends the core container a message via the vService to trigger that functionality when a new container starts.

We split up the SM into two separate daemons, the security helper daemon and the `tpm2d`. The security helper daemon handles the containers' FDE and signature verifications using the OpenSSL library in combination with the TPM engine. The responsibility of the `tpm2d` is to implement an interface between the TPM and its users, the core container and the security helper daemon. The tpm2-daemon's code base includes IBM's TPM 2.0 TCG Software Stack (TSS) [Gol]. We used the tool `tpm2simulator` based on [Wag] to emulate a TPM and test the functionality of our daemon. We adapted the TSS implementation to directly communicate over a UNIX domain socket with the simulator.

The labelled, shaded rectangles in Figure 3.18 depict the interfaces between the virtualization layer and the containers, for which we use UNIX domain sockets. The tpm2 socket offers the core container TPM functionality, whereas the core container uses the control socket for service container management with collaboration of the CM. The service socket is used for the vService communication between each container and the CM. The unlabelled, shaded rectangles depict the internal socket communication between the entities

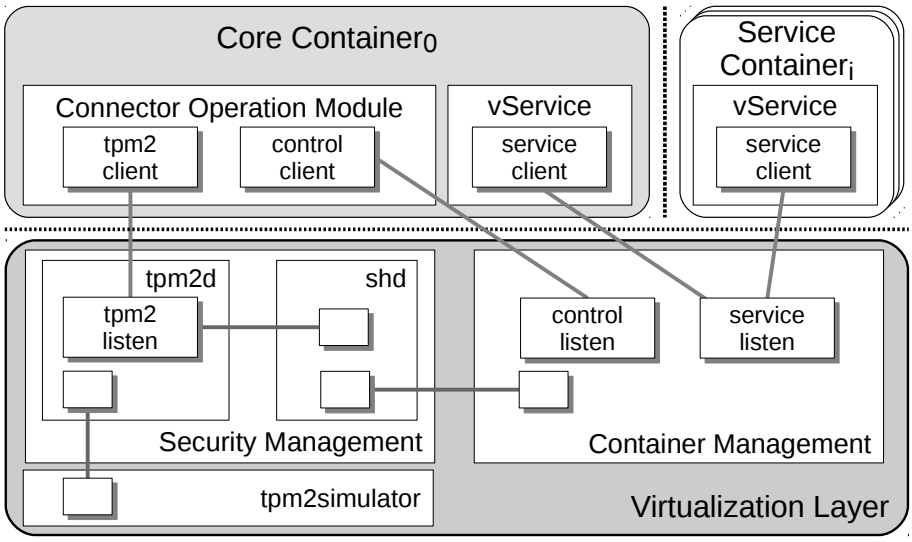


Figure 3.18: Communication channels between container and virtualization layer.

of the virtualization layer. To provide a secure implementation, we used protobuf [Gooe] as communication protocol over all sockets except for the tpm2simulator, which implements the corresponding protocol of the TSS.

3.6 Summary

We developed a secure architecture for OS-level virtualization on mobile devices in Section 3.3. Including an SE, the main objective of the secure architecture is data confidentiality at container boundaries. To fulfill this goal, we systematically isolated the different, simultaneously running containers from each other. Therefore, we restricted the different containers to a minimal set of controlled functionality. This made it possible to confine communication of the architecture's components to only well-defined channels for container management and device virtualization. In order to realize the strict isolation, we devised a stacked LSM concept using SELinux and a specially tailored, custom LSM. We furthermore leveraged Linux capabilities and the cgroups devices subsystem. Based on that, we developed mechanisms for secure device virtualization and secure container switching sustaining a seamless user experience. Thereby, we classified devices into different categories and provided containers with distinct hardware functionalities on a per-container basis. To demonstrate the feasibility of our approach, we realized the secure architecture with a fully-functional implementation on the Samsung Galaxy S4 and the Nexus 5 devices. The performance evaluation shows that the system performs well and that it is suitable for real-life application. In our security discussion, we concluded that the architecture provides data confidentiality even when large parts of the system are compromised.

In Section 3.4, we focused on the applicability of the secure architecture in real-world scenarios. Our approach was driven by the application of mobile devices in corporations or governmental institutions. In such scenarios, users are likely to work with sensitive data while using the devices for miscellaneous other tasks. We introduced a secure provisioning,

enrollment and update process to cover the whole lifecycle of devices. To associate users with containers and devices, we introduced an SE. We also introduced PKI hierarchies to provide trust between container software, users, devices, and the management backend.

In Section 3.5, we designed the secure virtualization architecture for ecosystems where heavily interconnected devices in distributed networks exchange, gather and process sensitive data. As the interconnectivity of devices and system progresses, like in IIoT scenarios, and because more and more sensitive data is exchanged, our proposed concepts represent an important building block for securing these heterogeneous systems. One of the characteristic scenarios we referred to was the IDS, a platform to exchange data across the boundaries of organizations. We started with identifying security requirements representative for such ecosystems. We then developed a holistic security architecture of a trust ecosystem in which we sought to address the identified requirements. We covered identity and trust management, a trust ecosystem's data-, application- and device-lifecycle, as well as secure device-to-device communication. On device-level, we transformed the previous secure virtualization architecture for mobile devices with end users to the embedded domain without end users. We called the embedded devices trusted connectors, on which we isolated the containers, securing their sensitive services and data from malicious third parties and other possibly malicious containers. We implemented a full-fledged prototype of our secure architecture on device-level including a communication protocol to establish trust between devices. Compared to smartphone use cases, the exchange of data has higher requirements on achieving data confidentiality. Therefore, we extended our embedded architecture with secure and measured boot capabilities for boot-time integrity verification and measurements. This allows to establish trust between connectors through remote attestation before exchanging data. Future design of ecosystems for connected devices can thus benefit from our work building on our architectural design and our open-source implementation [Fraa] and documentation [Frab].

With the design and implementation of our virtualization architectures and with the design of ecosystems for their application in practice, we made Contributions 1 through 4 and tackled Challenges 1 and 2.

CHAPTER 4

Main Memory Extraction based on the Cold Boot Attack

This chapter addresses Challenge 3, the main memory extraction from conventional computing platforms. For this purpose, we present Contribution 5, a forensic framework for main memory extraction based on the cold boot attack, for the domain of mobile devices. With this framework, we not only develop a method for systematic memory extraction, but also urge the need for mechanisms protecting against memory extraction by physical attackers, a topic we cover in Chapter 5. This domain is of special interest because our today's mobile devices store vast amounts of sensitive data about their owners in both volatile and non-volatile memory [Nta14; Pet07; Tan12]. This includes contact details, user credentials, personal and business emails, pictures, location history, or perhaps even the user's health states. Especially in sensitive corporate or governmental domains, the reliable protection of valuable and possibly classified data is an important topic. While modern mobile device OSs, like Android and iOS, increase the confidentiality of data by encrypting the file system, main memory remains unencrypted, making confidential data prone to getting extracted with a cold boot attack.

Cold boot attacks exploit the remanence effect of DRAM, which causes data to fade gradually in main memory [Gru13; Gut01]. The longer the timespan between the power cut-off and restarting the device, the more bits degrade in main memory. Cooling down the physical memory mitigates the amount of data loss by the remanence effect. While on desktop devices, RAM modules can be removed and plugged in to a host device for memory analysis, RAM modules of mobile devices are soldered on the System on a Chip (SoC) and thus not removable. A cold boot attack can be initiated on mobile devices by either shortly removing the battery from the device or by triggering a hardware reset functionality. This makes it possible to reset the system and to deploy forensics tools in early boot to conduct the data acquisition, instead of booting back into the OS [Mül13].

The first part of this chapter describes the design of our novel framework for main memory extraction on mobile devices based on the cold boot attack. Our framework does not require any runtime privileges on the targeted device's OS. Our framework almost completely sustains the device's previous memory contents and requires overwriting only a minimal amount of bytes in main memory while not initializing device memory. Keys usually have known patterns or kernel structures, which makes them easy to identify. Examples are the kernel structures for FDE keys [Hew; Pet07], or the patterns of AES and RSA keys in memory, such as key schedules [Pri]. The more memory contents are sustained, the higher the chance to precisely reconstruct the device's previous state and its secrets. The amount of unimpaired bytes in memory after the restart thus reflects the quality of

an attack. This not only depends on time and temperature, but also on the impact of the framework on the device's memory. Not allowing the forensics module to overwrite important content during the acquisition process is essential and an often neglected factor in the data acquisition process. The existing Forensic Recovery Of Scrambled Telephones (FROST) framework [Mül13], for instance, overwrites kernel memory when booting the forensics tool, deployed on the recovery partition, on the device. In particular, this includes the system state, for example, the list of running processes and their mapping in physical memory. Instead of utilizing a full-fledged Linux kernel, like in the FROST framework, we boot the mobile device with a minimalistic and easily portable application. Our framework thereby sustains all the data relevant for the analysis of the previously running system by overwriting no more than three kilobytes of constant data in the kernel code section. For the practical demonstration of the feasibility of our framework, we implement it for the Samsung Galaxy S4 mobile device and port it to the Nexus 5 device.

Based on the memory-preserving property of our framework, we present a method for the systematic acquisition and analysis of the device's memory contents in the second part of this chapter. As the memory still reflects the previous system state, such as the kernel structures and page tables of the MMU, we make use of this unaltered state for our systematic analysis of the data. This allows us to efficiently analyze the memory contents with existing memory forensics tools. For that purpose, our application provides a communication interface to a host system via the UART serial interface. The communication interface makes it possible to request memory dynamically and offload the analysis to the host system. Data acquisition tools utilizing this interface can thus be leveraged for the forensic analysis on the host system instead of running them on the device [Aga15; Hoo11]. We extend the memory forensics tool Volatility with an implementation of the communication interface for data acquisition. We conduct an extensive evaluation of our proposed framework and compare the cold boot-based analysis with traditional memory dump analysis. We also show the potential of our framework by acquiring sensitive user data in a concrete use case. In addition, we extract the container storage encryption keys from mobile devices running our virtualization platform from Chapter 3.

The remainder of this chapter is organized as follows. In Section 4.1, we present related work in the fields of forensics and cold boot attacks. In Section 4.2, we provide background information about the interpretation problem of raw data obtained from memory dumps. We elaborate the design of our framework in Section 4.3. In Section 4.4, we describe the implementation of our framework. We explain the device-specific realization for the Samsung Galaxy S4 device and describe the framework's portability in Section 4.5. We then evaluate our forensic framework in Section 4.6 and discuss important aspects in Section 4.7 before we summarize this chapter in Section 4.8.

Contribution 5 is based on publications [Hub16b; Tau15]. The contribution results from joint work with Benjamin Taubmann, Sascha Wessel and Lukas Heim. Both Manuel Huber and Benjamin Taubmann collaborated in the conception, implementation and evaluation. Benjamin Taubmann formulated the idea and defined the architecture of using a cold boot attack for the forensic data acquisition process on mobile devices. He also introduced the concept of the bare-metal application. Lukas Heim worked as part of a student project on the implementation while Sascha Wessel provided input to the concept work.

4.1 Related Work

In this section, we present related work that addresses cold boot attacks and alternative memory extraction methods, especially in the context of mobile devices. We also provide a brief overview regarding forensic data analysis on Android devices and on countermeasures to cold boot attacks.

Cold Boot Attacks

A preliminary approach to cold boot attacks for acquiring memory of a previously running system was to force a reboot where memory is fully preserved [Cha08]. The preserved memory was leveraged to circumvent OS authentication mechanisms and allowed to even recover the state of the previously running system. The first cold boot attack was published by Halderman et al. in [Hal09], where the authors showed that it is possible to extract data from main memory on the x86 architecture after a short interruption of the power supply. This also works when a DRAM module is moved to another host computer. They showed that the rate of the degradation of volatile memory can be drastically reduced by cooling down the RAM module. For data acquisition, the tool *bios memimage* boots directly from a USB flash drive or via Preboot eXecution Environment (PXE) boot [Pri]. The tool transmits the content of the DRAM modules via network to another investigation host or stores it on a USB flash drive. Additionally, the tool provides features to find and fix corrupted RSA and AES keys in a memory dump.

Based on the approach for desktop computers, the cold boot attack found its adoption on the ARM architecture. With FROST [Mül13], Müller et al. showed that cold boot attacks are feasible on Android phones. On those devices, it is not possible to boot from external sources, like USB sticks. In addition, the RAM module is non-removable because it is integrated into the SoC. The approach with FROST is to force a restart of a running device by interrupting the connection to the battery. Afterwards, the already installed FROST framework is booted from the recovery partition by pressing the corresponding buttons on the device that trigger the recovery mode. The FROST framework loads an entire Linux kernel and features a kernel module that searches for AES keys in main memory. The FROST boot image is flashed onto the recovery partition before the analysis. This circumvents the restrictions in booting external sources. To flash the boot image, the bootloader has to be unlocked. This usually triggers a routine that formats the user data partition. A drawback of the FROST framework is that the heap of the previously running kernel gets overwritten, because the framework boots a full-fledged Linux kernel. The overwritten contents include information like structures of the MMU, the list of running processes and the memory mappings of processes to physical locations. Additionally, the kernel likely reinitializes I/O devices, which resets the corresponding device memory, possibly in the interest of a forensic investigator.

Alternatives to the Cold Boot Attack

Cold boot attacks do not allow for live forensics, as the system halts for a short moment rebooting the device. For live forensics, it is possible to directly access memory on a running device. The two most obvious ways are to either read directly from `/dev/mem` or to use tools like the Linux Memory Extractor (LiME) kernel module [Syl12a]. One problem

is that both approaches require root permissions [Ros]. It is only possible to bypass that problem by exploiting security flaws in processes that have root access. The other problem is when using kernel modules, the running kernel has to be capable of loading custom kernel modules.

Another option to access main memory is the use of devices that offer DMA. For the x86 architecture, this was shown for the PCIe [Dev09], Firewire [Bec05] and Thunderbolt [Maa12; Sev13] interface. Those interfaces are in general not available on mobile devices. However, mobile devices often have a JTAG interface for debugging purposes. This provides full access to main memory at runtime. In [Wei12], the author uses the JTAG interface to exploit the baseband of a smartphone. The RIFF Box [RIF] is a device that makes it easy to retrieve a memory dump or even to read or write the memory on the internal flash drive via the JTAG interface.

Forensic Data Analysis

Digital forensics goes back to approaches on early computers decades ago [Aga15] and nowadays finds its adoption on Android devices. In [Thi10], the authors use the process trace system call to stop and resume processes and to create memory dumps of their address spaces. This is useful when data remains in memory only for a very short period. This happens, for example, when it is loaded and erased in only one routine.

In [Apo13; Nta14], Apostolopoulos et al. search for authentication credentials in the process memory of applications. They use the Dalvik Debugging Monitor Server (DDMS) tool [Apo13] and the LiME kernel module [Syl12a]. The authors execute both analyses on running mobile phones with root privileges. Hilger et al. show a memory forensics application that uses the memory of cold booted devices in [Hil14]. They create tools based on the FROST framework to analyze the heap of the Dalvik Virtual Machine. With this approach, they are able to obtain critical data, for example, the phone call history, the last user input, and passwords [Syl12b]. Gruhn et al provide a detailed analysis on the susceptibility of DDR1 and DDR2 RAM to the cold boot attack [Gru13]. The work in [Lin15] shows how to run cold boot attacks against DDR2 and DDR3 RAM while [Yit17] investigates the feasibility of the cold boot attack on DDR4 modules. Carbone et al analyze the applicability and the effects of memory decay in detail in [Car11].

Countermeasures against Cold Boot Attacks

Research on mitigating cold boot attacks mainly focuses on protecting the FDE key against the attack [Göt13; Mül11; Mül12; Ski13]. In [Göt13], the authors relocate the disk encryption key from main memory to the CPU registers of the ARM microprocessor. This goes back to [Mül11], where the authors develop the approach for keeping the disk encryption key in registers for the x86 architecture. Note that these approaches specifically protect against cold boot attacks, but do not defend against DMA attacks where an attacker is capable of writing memory [Bla12]. The work in [Ski13] describes a software-based approach to protect the key while being in a private mode that allows using basic device functionality. In [Col15], the approach is to encrypt user data in main memory when the device switches to the screen-locked state. The utilized key is stored on the ARM SoC rather than in DRAM. There is however only little research on the application of the cold

boot attack for sophisticated forensic analysis or for the inspection and improvement of security features. Anti-forensics techniques, for example, [Zha15] for the x86 architecture, aim to defeat memory acquisition modules by manipulating the physical address space layout. Data scrambling in DRAM interfaces is another countermeasure, which however has been shown to be still susceptible to the cold boot attack [Bau16; Lin15; Yit17]. Another approach against cold boot and memory attacks is main memory encryption, which we focus on in Chapter 5.

4.2 Background on Data Interpretation

In order to interpret the raw data in main memory, we require meta information. The problem of data interpretation is the *semantic gap* in the fields of Virtual Machine Introspection (VMI) [Che01]. The required meta information depends on the OS, the kernel version, and its configuration. This information is necessary to determine the location of kernel structures and which components those structures include.

The kernel data structures are important for a full analysis, for instance, to reconstruct the list of running processes and their memory mappings. In FROST, the full Linux kernel is booted and overwrites the data of the former running kernel. This causes a high amount of crucial information loss and results in non-reconstructible data.

In our framework, we utilize the tool Volatility for data interpretation [Theb]. Volatility is an easily extendable open source application for memory forensics. Volatility consists of a collection of tools for acquiring memory, for bridging the semantic gap and for the extraction of relevant information. Volatility provides a decent amount of plug-ins that obtain detailed information on the target system. For example, the plug-in *linux_pslist* retrieves a list of running processes. The application supports different operating systems (Microsoft Windows, Mac OS X, Linux and Android) and architectures (x86 and ARM). Volatility requires supplementary data about the target system. *Volatility profiles* reflect this supplementary data. Profiles contain the metadata about the structures and debug symbols of the kernel running on the target system. Volatility comes with a set of standard profiles, such as for Windows systems. Profile information can be automatically extracted from the kernel source code, such as with [Thec] for Linux-based systems.

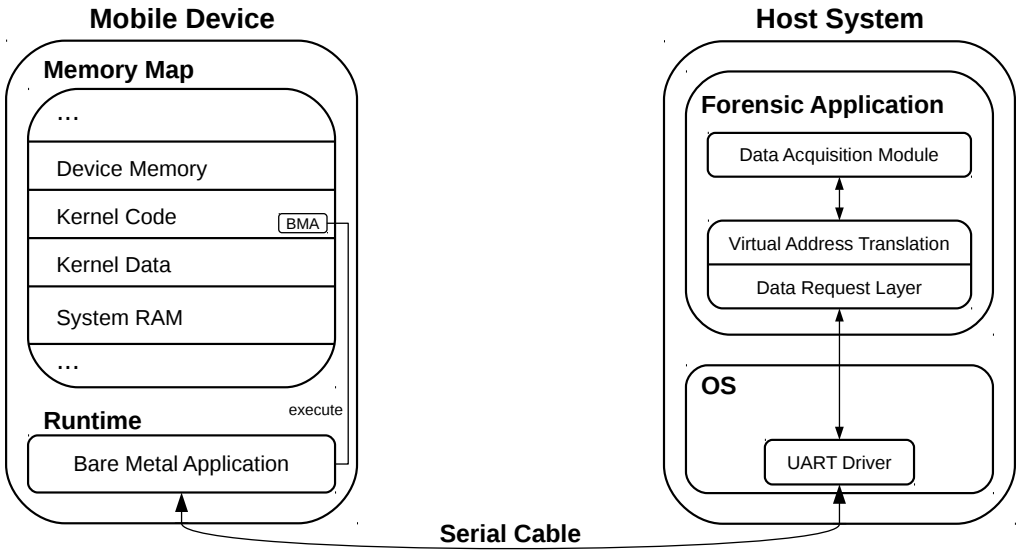


Figure 4.1: Schematic overview of our main memory extraction framework showing the BMA on a mobile device communicate with a forensics application on the host.

4.3 Design of a Cold Boot-based Memory Extraction Framework

The central design primitive for our memory extraction framework is to preserve as much memory as possible on the target device. We construct a minimalistic module that runs on the device during memory extraction. This module is an application which does not require a Linux kernel or any libraries at runtime. The sole purpose of this module is to read the device's main memory and transfer it to a host device, which can systematically analyze and interpret the device's main memory contents. This module, once deployed on the target device, occupies only a smallest amount of main memory. Amongst other important memory regions, we preserve the structures of the formerly running Android kernel when booting our module. We call this module the Bare-Metal Application (BMA).

In order to transfer the data between the target device and the host device, the BMA uses the target's serial interface. Mobile devices usually expose a serial interface to the outside. The corresponding driver to initialize and access the serial interface is part of the BMA. The driver can be implemented very efficiently in terms of memory, which keeps our BMA small. Drivers for other hardware peripherals that can be used to transfer data from the mobile device, such as the USB interface, are significantly more complex. In particular, our driver only comprises functionality for reading and writing from a dedicated register of the UART interface.

Figure 4.1 depicts an overview of our framework. The illustration shows the two main elements of our framework: the minimalistic BMA on the mobile device and the forensics application on a host device. The design choice of using a minimal BMA and serial interface has the advantage of the framework being optimized for the memory footprint on the target device. However, due to the slow transmission rate of the serial interface, this choice comes at the cost of performance when transferring the memory contents to the host device. In our framework, we thus aim to only request *relevant* data from the target device, see

Section 4.4.2. The relevant data comprises of the data intended to be extracted, such as key material or other valuable secrets, and the data that is necessary to efficiently locate the secret, such as kernel structures pointing to areas where secrets are likely to be found. Because the location of secrets in main memory is usually not known beforehand, the latter data helps to significantly speed up the extraction of the desired data, compared to the naive extraction of main memory contents. The utilized forensics application on the host device contains the decision logic for acquiring the relevant data from main memory.

The left side of Figure 4.1 shows that we map the BMA to the code segment of the previously running kernel. We assume that (at least the small portion of occupied memory in) the kernel's code segment does not contain any relevant data. The BMA boots directly on the target system without requiring any other dependencies. The BMA implements a simple protocol on top of the serial driver in order to receive and process commands from the forensics host. For our framework, we only require one command. This command includes a physical start address and the amount of bytes the BMA has to read from main memory and return.

The forensics application for data acquisition and analysis on the host system is depicted on the right side of Figure 4.1. The application directly requests data from the serial interface, connected with the target device via a serial cable. The forensics application uses *data acquisition modules*, for example, to retrieve a list of the former processes running on the system. This list is represented by data structures of the former running kernel in memory. Based on the list of former running processes, sensitive data of processes can be extracted in a targeted way. A forensic investigator can choose a specific process and retrieve the kernel structures, which indicate in which regions the memory of a targeted process can be found. However, these memory regions are expressed as virtual addresses, because of the kernel's virtual memory management for processes.

Forensics applications usually use a memory dump for the analysis, instead of directly analyzing memory contents with the BMA. Based on a traditional memory dump, two steps are required to locate a the virtual address of a process in the dump:

1. Translation of virtual to physical addresses in main memory. This is hardware dependent and requires the information stored in the page tables of the MMU.
2. Translation of the physical address to an offset inside the dump. This depends on the storage format and requires the meta information for memory segment mapping inside the dump.

As shown in Figure 4.1, the virtual address translation layer in our proposed framework determines the virtual to physical address mapping. The data request layer usually translates the physical address to an offset in an acquired dump. Normally, this layer reads a dump file at the determined offset. In our framework, this layer makes use of a UART driver to directly request the memory starting from the physical address from the BMA on the mobile device. We explain the details of the data acquisition process in more detail in Section 4.4, where we use Volatility as forensics application.

On the one hand, despite a comparably slow throughput using the serial interface, the design of our framework allows to gather full, genuine memory dumps from devices for later

analysis. On the other hand, the design allows to directly conduct the analysis of main memory on the powered device. In both ways, we are able to reliably identify confidential data, ranging from FDE keys to vast amounts of confidential user data. The flexibility of the framework allows for customization, depending on the goals of an investigator. Volatility is our choice of forensics application on the host, but other tools for systematic memory analysis and extraction can be leveraged. However, utilizing tools or plug-ins for exhaustive search results due to the serial interface in poor performance. Since the kernel code segment is way larger than the size of our BMA, it is possible to extend the BMA and its communication protocol. One such possibility is to establish algorithms as part of the BMA that search for known patterns or structures, such as for FDE keys [Hew; Pet07], and other AES and RSA key structures [Pri]. The advantage of this approach is that memory can be rapidly accessed from within the BMA, and that only the identified key material needs to be transmitted to the host system. Another extension of the BMA can be to introduce a simple compression mechanism to the protocol. This increases the performance of the data requests, for example, when an application requests large amounts of data which mostly consist of zerobyte chunks. However, the current functionality of our BMA fulfills our design goal of only overwriting a minimal amount of memory on the target device.

4.4 Implementation

In the following, we first describe the implementation of the BMA. Afterwards, we elaborate on the extension of the memory forensics application Volatility.

4.4.1 Bare-Metal Application

As already discussed, there are two ways to execute a cold boot attack. The first way is to move the memory module to another host device with an installed forensics application. On mobile devices, this is not feasible because the memory is integrated into the SoC and cannot be removed. The second way applies for mobile devices, where it is possible to halt the system and to abruptly restart the device in order to directly boot a forensics application on the device itself, which is the BMA in our case.

The main tasks of our BMA is to initialize the UART interface and to process incoming data requests. We utilized a large portion of the code for initializing the UART interface directly from the Qualcomm Linux kernel [Cya]. The BMA, comparable to a daemon with an infinite loop, waits for incoming requests on the UART port. For this purpose, the initial code of the BMA sets up a stack with a size of 1,024 bytes. In the next step, the BMA reads and parses these requests. We implemented the parsing routine for incoming commands with a minimal encoding, which saves memory to be allocated. After parsing the command, the BMA iteratively reads the corresponding data from main memory and writes it to the UART interface. The BMA allows the forensics application to request arbitrary memory chunks up to a full image of main memory.

In contrast to the Linux kernel, our BMA does not initialize any other peripherals, but only configures the UART interface. We thus also preserve device memory, which otherwise might be altered by the set-up routine of a device driver of the Linux kernel.

We discussed that the UART interface provides only a low data transfer rate, which

makes the extraction of a full dump of main memory inconvenient. As an extension, we implemented an FDE key search functionality based on the kernel structures wrapping this key, close to [Hew]. This functionality comes with a simple heuristic that identifies keys even when bit flips occur. We trigger this functionality with a further command in the protocol to make the BMA search and return identified keys.

4.4.2 Extension of Volatility

We execute the forensic analysis of the contents of main memory with the memory forensics application Volatility. Volatility commonly requests data from memory dumps, for example, acquired directly on running devices using the LiME kernel module. Volatility provides the possibility to translate virtual addresses to physical addresses with its concept of Address Spaces (ASs). Depending on the use case, Volatility allows to combine and stack ASs. We used the *ARM address space* for virtual to physical address translation on the ARM-based mobile devices in our framework [Syl12b].

According to our framework design, we need to extended Volatility with the implementation of a custom AS to realize the data request layer, as depicted on Figure 4.1. This AS, which we call *serial address space*, requests data over the serial port, instead of requesting data from a memory dump file. Furthermore, we require a *Volatility profile* for the specific device under analysis [Hil14]. We describe the implementation of the serial address space and how to create a Volatility profile in the following.

4.4.2.1 The Serial Address Space

Our implementation of the serial address space resembles the existing *file address space*. The latter requests the contents from a local memory dump file. Instead, the serial AS implements the protocol required to request data from the BMA via the UART interface. The serial AS opens the serial port on the host device upon AS initialization. Consequently, when a plug-in requests data from an address, the serial AS directly passes the request to the BMA on the target device. For this purpose, the serial AS writes the dump command with the start and end address of the request to the UART interface. When the BMA returns the requested data via the serial interface, the serial AS receives the requested data and returns it to overlying ASs. Figure 4.1 depicts the different layers used by a standard Volatility plug-in, such as *linux_pslist*. In the example of *linux_pslist*, the plug-in retrieves the list of running processes by following the linked list of task structures, called `task_struct`, in the Linux kernel.

As some address requests are repeatedly made during forensic analysis, such as for reading the page tables, we equipped the serial AS with a cache that stores the data of former requests. In case a request occurs repeatedly, the serial AS immediately returns the cached data instead of repeatedly requesting the chunk from the device. This likely decreases the duration of the analysis many times over.

4.4.2.2 The Volatility Profile

Volatility stores the meta information required to interpret the data extracted from main memory in a Volatility profile. The profile bridges the semantic gap, see Section 4.2. For example, it provides a map of the Linux kernel's data structures in memory. A profile

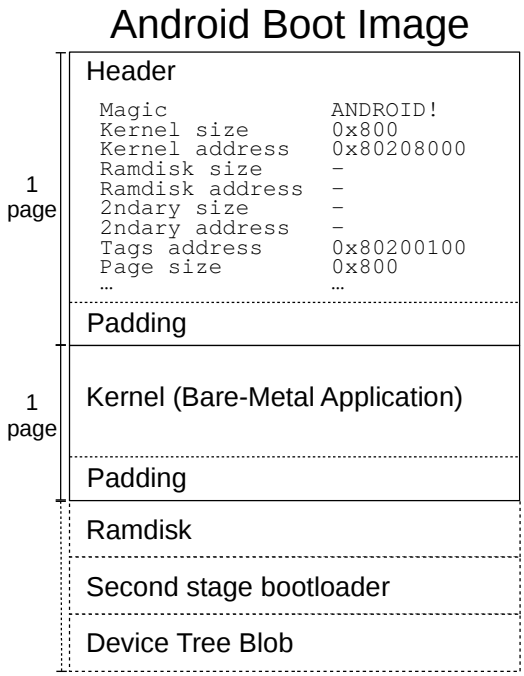


Figure 4.2: An Android boot image wrapping the BMA.

strongly depends on the OS type and the corresponding kernel version on the target device. In order to create a Linux profile, the source code of the deployed kernel is required, see Section 4.5. The source code for mobile devices is, at least for Android-based devices, typically available open source for the different devices on the market. Despite that hardware vendors might make non-public changes to the kernel, its data structures required for forensic analysis remain usually unmodified.

4.5 Device-Specific Realization

We selected the Samsung Galaxy S4 GT-I9505 device for our specific use case to realize our framework. The Galaxy S4 device is a commonly used mobile phone and provides a serial port. We deployed the CyanogenMod Android 4.4.4 distribution (in version `11-20141008-SNAPSHOT-M11-jflte`) on our target device. To create the Volatility profile, we utilized the corresponding CyanogenMod kernel source code (in version `3.4.104-cyanogenmod-g42b4b50-dirty`) [Cya].

We first describe the deployment of the BMA onto the device's recovery partition. For flashing the BMA onto the recovery partition, we wrap the BMA into an Android boot image. Then, we describe the boot procedure in order to launch the BMA. To be able to connect the BMA with a host system, we also describe our hardware setup. Finally, we show that our solution is easy to port to other devices that offer a serial interface by porting the framework to the Nexus 5 device.

Listing 4.1: Truncated output of `/proc/iomem` on a Samsung Galaxy S4 device.

```
...
2a03f664-2a03f6a4 : pc-cntr
2a03f720-2a04071f : tz_log.0
80200000-87dfffff : System RAM
80208000-80f8e523 : Kernel Code
8111a000-817a6da3 : Kernel Data
89000000-8d9fffff : System RAM
8ec00000-8fdfffff : System RAM
8ff00000-9fdfffff : System RAM
a6700000-fe1fffff : System RAM
fff00000-ffffefff : System RAM
```

4.5.1 Wrapping and Deployment of the Bare-Metal Application

In order to be able to deploy the BMA on the recovery partition of the mobile device, we wrapped the BMA into an Android boot image. This makes the device's bootloader capable of loading the BMA as a regular Android boot image. Our generated boot image contains the addresses the bootloader reads to map the BMA into main memory when loading it from the recovery flash partition.

Figure 4.2 illustrates the structure of the boot image. An Android boot image has a flash page-sized header. The header always carries an eight byte magic start value (`ANDROID!`). The header contains the necessary fields of different size to provide the bootloader with information about the image's contained data elements. In the kernel address field, we store the address the BMA gets mapped to in main memory (`0x8020800`). The flash page size of the Samsung Galaxy S4 device is set to 2048 byte (`0x800`). The size of the kernel, here the BMA, is exactly one page. The BMA is located in the boot image right after the header, where the compressed kernel binary can normally be found. Our boot image neither contains a ramdisk, a second stage bootloader, nor a Device Tree Blob (DTB). The remaining fields in the header are optional and can remain empty. The exact layout depends on the device's bootloader, see Section 4.7.

Our configuration in the boot image header maps the BMA to the address `0x80208000`. According to the `iomem` output for the Galaxy S4 device, this address is where the code segment of the formerly running Linux kernel is located, see Listing 4.1. The initial code of the BMA sets its 1,024 byte stack adjacent to its own code segment, i.e., also within the kernel code section. In doing so, we overwrite no more than 3 KB of memory completely located in the kernel code segment.

In order to launch the BMA, we flash the generated Android boot image to the recovery partition of the device. We execute this step after the short power cut-off, respectively, the hardware reset of the device during the cold boot attack.

4.5.2 Boot of the Bare-Metal Application

To deploy and boot the BMA on the target device, we need to consider the particular device's boot procedure. Figure 4.3 illustrates a schematic representation of a typical device boot process. The illustration describes the different boot stages between pressing

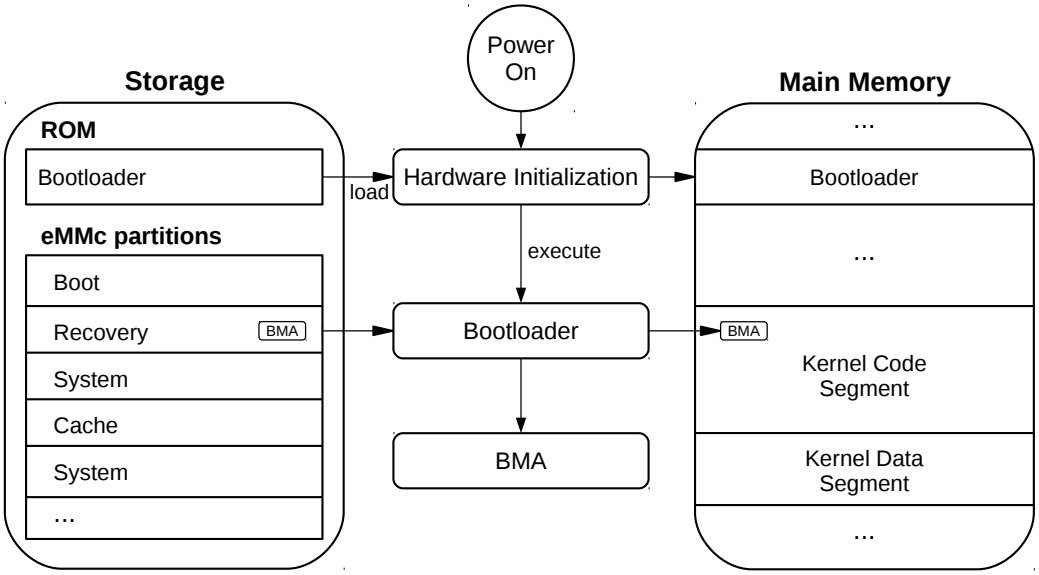


Figure 4.3: Boot procedure when starting the BMA from the recovery partition on a cold booted device.

the power button and booting the recovery image. The bootloader boots our pre-deployed boot image from the recovery partition and places the BMA inside the former kernel code segment. In order to do so, the bootloader reads out the values inside our boot image header and also checks the header's magic value.

The common boot procedure requires the following steps. When turning on a device, the system initializes the hardware and executes the routines stored in the boot ROM. These routines load the bootloader. Upon its execution, a routine in the bootloader checks the status of the hardware buttons to identify which boot mode to trigger. In general, most Android devices can boot in three different modes: *normal*, *recovery* and *download*. The normal mode is the default boot mode. This mode starts the Android OS when switching on a device the common way. In order to do so, the bootloader starts the kernel stored on the boot partition of the embedded Multi-Media Controller (eMMC).

Figure 4.3 describes the scenario where a startup into recovery mode takes place. The usual purpose of the recovery mode is to update, install or repair an Android system. In order to trigger the recovery mode, a special key combination must be pressed when switching on the device. On the Samsung Galaxy S4, this is the combination of the *Volume Up* and *Power* buttons.

The download mode allows to directly write data to partitions via USB, for example, to the recovery partition. In this mode, we use the tool `heimdall` [Gla] to write the BMA via the USB interface to the device's recovery partition. To trigger the download mode on the Galaxy S4, the *Volume Down* and *Power* buttons must both be kept pressed.

4.5.3 Hardware Setup

We realized the connection between the Galaxy S4 and the host system, a common PC, via the Samsung Anyway Jig [XDA]. The Samsung Anyway Jig serves as a universal

maintenance tool for various devices produced by Samsung. It is equipped with a D-Sub DB-25 connector and can be connected to the supported device via a proper adapter cable. To connect the jig to the test device, we used a custom Micro-USB to D-Sub DB-25 cable. We established the connection to the host PC via the RS-232 interface. The Samsung Anyway Jig Adapter connects the GND and ID-line of the Micro-USB of the device port with a resistor. This configures the Micro-USB port such that it acts as a serial interface port. We configured the dimension of the resistor with the DIP-switches of the Samsung Anyway Jig. After connecting to the UART port, the whole boot process of a device can be monitored because the bootloader writes debug information to the serial interface. Depending on the kernel command line parameters, the kernel can also output its debug information over the serial interface.

4.5.4 Portability of the Framework

We verified the easy portability of the framework by realizing and deploying the BMA for the Nexus 5 device. We did not need to modify our Volatility extensions, as the only requirement is a new profile based on the Nexus 5 kernel code. For the BMA, only three changes to our created boot image were necessary. First, the BMA had to be remapped in memory to a different location. According to the `iomem` layout of the Nexus 5 device, the first *System RAM* segment starts at address zero and the kernel code segment is located at `0x8000`. Second, the base offset value of the UART registers in the BMA had to be adjusted. This is because the UART core is mapped to a different location in memory, compared to the Samsung Galaxy S4 device. We could completely reuse the serial driver of the BMA for the device. Third, the bootloader on the Nexus 5 device expects a DTB. Otherwise, the bootloader refuses to boot up. The bootloader locates the DTB in the boot image based on an offset determined by a pointer at an offset in the boot image's kernel image. We modified the BMA at that offset to have a pointer, which points to the straight after the BMA. This allowed us to append a minimal self-crafted DTB to the BMA, which the bootloader accepts. The bootloader relocates the DTB to the *tags* address. We do not overwrite crucial data with the relocation, because at that location, the constant-valued DTBs of the former running kernel can be found. Like on the Galaxy S4, the bootloader required no ramdisk or second stage bootloader. To flash the boot image to the device, we used the tool *fastboot* [Gooa]. On the Nexus 5 device, the physical UART interface is integrated into the headphone socket. Therefore, we crafted a 3.3V USB to serial cable that we soldered onto a headphone jack.

Table 4.1: Number of successfully retrieved bytes from a 10,000-byte array in main memory with different cold boot attacks.

Attack Type	Min	Max	Average
Fast Cold Boot	9,983	9,998	9,991
Slow Cold Boot	26	9,521	3,379
Reset Cold Boot	9,998	10,000	9,999

4.6 Evaluation

In this section, we evaluate our proposed framework. We first measure the amount of data that degrades due to the cold boot attack in Section 4.6.1. This quantifies the feasibility of our approach. In Section 4.6.2, we demonstrate the application of our framework on the Samsung Galaxy S4 device using Volatility. We compare traditional Volatility analysis on a LiME memory dump with the cold boot-based analysis using our BMA. In the last step, we demonstrate the potential of our framework by showing how to extract sensitive user data from a cold-booted device in a concrete use case.

4.6.1 Loss of Information

We evaluate the loss of information with our framework considering three aspects: the decay of memory through the device restart, during the analysis takes place, and trough the memory the BMA occupies.

4.6.1.1 Decay based on the Cold Boot Attack

We executed two different types of the cold boot attack to evaluate the amount of data that decays during the power cut-off:

1. Momentary removal of the battery and restart of the device once the battery is re-inserted.
2. Power button press for a few seconds while the phone is still running, causing a hardware-based reset.

Note that we executed both attacks at a temperature of approximately 20° Celsius. The results improve when cooling down the phone and its memory modules, as described in [Hal09; Lin15; Mül13].

Before rebooting the device, we wrote an array of 10,000 known bytes to main memory with a kernel module, which prints the physical start address of the array. Afterwards, we read the contents of the array's physical address and counted the unimpaired bytes. We executed the analysis 25 times for three different cold boot attack scenarios. Table 4.1 depicts the corresponding results. In case of the *fast* cold boot attack, we re-inserted the battery as fast as possible. In case of the *slow* attack, we re-inserted the battery after approximately 1 second. In case of the device reset, we pressed the power buttons as described to reset the device.

The depicted results mainly show that a non reset-based cold boot attack does not always return reliable results, as it depends on multiple factors like temperature and the

speed of battery re-insertion. In case of the fast cold boot attack, we retrieved 9,991 of the 10,000 known bytes on average. Despite the only weak degradation, the application of Volatility plug-ins was less frequently successful. With more bytes fading in memory, the analysis based on the possibly broken pointers of kernel structures is no longer reliable. When conducting a slow cold boot attack, the degradation proceeded quickly. In this case, the successful application of Volatility plug-ins became infeasible.

However, the reset-based attack provided promising results during our tests. This scenario is more reliable as it does not depend on how fast the battery can be re-inserted. In most cases, we retrieved all of the bytes successfully and had occasional bit flips only in very few test cases. This shows that even in case of the reset attack, where we did not remove the battery at all, the memory occasionally decays. In normal cases, where we retrieved all of the bytes correctly, the application of Volatility plug-ins was always successful.

In our test set-up, it was also possible to successfully extract data with our framework from the device when it was restarted twice. A second restart is required on the Samsung Galaxy S4 device to reboot into recovery after flashing the BMA onto recovery partition before the analysis. On the Nexus 5 device, the bootloader allows to immediately boot from a partition after flashing. This makes a second reboot obsolete.

4.6.1.2 Decay during the Analysis

As we progressively acquire data from the device's main memory, we rely on the data to remain intact on the target device during the whole analysis process. To demonstrate that this requirement is given, we extracted data multiple times 15 minutes after the device has booted the BMA. As expected, we always retrieved exactly the same unaltered data between the test requests. Furthermore, we inspected memory dumps on Nexus 5 devices running the virtualization architecture from Section 3.3, which comes along with FDE using dm-crypt. Gathering the dump of the 2 GB main memory with the BMA required about 42 hours. We compared this dump to a LiME dump created shortly before, finding that there was no decay during the analysis. As an example, we were able to recover the FDE keys of the Android containers in the LiME dump, as well as in the BMA's dump. In addition, we were able to quickly identify the keys using the BMA's FDE key search functionality. For this reason, we may assume that data does not decay any further when the BMA executes, because memory is constantly supplied with power.

4.6.1.3 Information Loss based on the Size of the BMA

Another important source of data loss is the amount of data that the BMA occupies in main memory. The boot image containing our BMA has a size of 4 KB and the size of the stack that the BMA sets up is 1,024 bytes. The bootloader loads the image header of size 2,048 bytes to a fixed location in memory. Based on the information in the header, the bootloader maps the BMA of the same size to the kernel code section. Thus, we overwrite no more than 3 KB of memory in the kernel code section. This is way less than the previously running kernel's size. The size of the CyanogenMod boot image for the Galaxy S4 device is about 5.9 MB. The kernel is compressed and extracted before it gets started. Because the kernel code segment forms a set of constant and known bytes, the segment

Table 4.2: Comparison of the outputs and runtimes of different Volatility plug-ins when using a dump file and the BMA for data acquisition.

Plug-in	Results Dump	Results BMA	Time Dump	Time BMA
pslist	230 entries	225 entries	03.54 s	24.23 s
iomem	138 entries	138 entries	02.18 s	08.97 s
proc_maps (init)	9 entries	9 entries	02.00 s	06.03 s
dump_maps (init, heap)	340.0 KB	340.0 KB	01.98 s	35.84 s
dump_maps (init, stack)	139.3 KB	139.3 KB	01.94 s	07.02 s
proc_maps (rild)	156 entries	156 entries	05.38 s	18.55 s
dump_maps (rild, heap)	380.9 KB	380.9 KB	02.05 s	41.48 s
dump_maps (rild, stack)	139.3 KB	139.3 KB	02.06 s	08.29 s

does not change between different runs of the system. This means that the overwritten bytes do not overwrite relevant data as the code segment of the formerly running kernel is mapped to the BMA's address range, see Section 4.4.1.

4.6.2 Forensic Memory Analysis

In the first part, we compare the application of multiple Volatility plug-ins using the BMA with a traditional LiME dump-based analysis. With this comparison, we show that we are able to similarly analyze the data retrieved from the BMA, compared to previously recorded memory dumps on running systems. Thereby, we also focus on the performance impact and on the caching effect of our serial AS during the BMA analysis. In second part, we demonstrate the potential of our framework by retrieving sensitive user data through an analysis with various Volatility plug-ins.

4.6.2.1 Comparison with LiME Dump Analysis

We created a memory dump on the running phone with the LiME kernel module as a reference right before we executed the analysis with our framework. Then, we reset the device to execute the Volatility plug-ins with our framework using the BMA running on the phone. Afterwards, we restarted the device, pulled the LiME dump and executed the same set of plug-ins on the dump file. Table 4.2 lists the results and runtimes of the plug-in applications on the LiME dump and on our BMA in combination with the reset attack.

The plug-in *linux_pslist* extracts a list of running processes. The resulting entries of our measurements differed between the dump file (230 entries) and the cold boot analysis (225 entries) by solely five more threads. This comes from the LiME kernel module creating these threads during the acquisition process. The analysis with the BMA took 24.23 seconds, whereas the LiME dump analysis took 3.54 seconds.

The plug-in *linux_iomem* extracts the map of the system's memory for physical devices. The results received from the cold boot-based analysis were equal to the memory dump analysis. As in the scenario before, the runtime of the plug-in was longer in case of the BMA application. Compared to the LiME dump analysis with a duration of 2.18 seconds,

the analysis with the BMA took 8.97 seconds. According to Table 4.2, the application of other plug-ins shows comparable runtime differences between the BMA and the LiME dump analysis.

The plug-in *linux_proc_maps* returns the memory mappings of a single process. This renders results similar to contents in `/proc/<pid>/maps`. For our measurements, we requested the mappings of the `init` and the `rild` process. The latter is responsible for the radio functionality of an Android phone. In both cases, the measurements returned exactly the same results: 9 entries in case of the `init` process and 156 entries in case of the `rild` process.

We finally requested the stack and heap memory segments of the `rild` and the `init` process with the plug-in *linux_dump_maps*. The amount of data in bytes was for both processes the same for the stack and heap. The data of the stack of the `init` process turned out to be consistent between the two acquisition methods. The same holds for the `rild` process.

In every test case, the time required for executing a plug-in which operates on the memory using the BMA was significantly higher. This emerged as a result of the low transfer rate of the UART interface. The average transfer rate we measured with our hardware was at about 11.25 KB/s when requesting large chunks of data. This speed reduces when plug-ins make lots of small data requests during the analysis due to the BMA's protocol overhead. For our purposes, the low transfer rate was acceptable, since the plug-ins terminated within less than 45 seconds.

The speed strongly increases due to the caching functionality in our serial AS, which buffers previous requests. Data once requested from the device is thereby stored in the cache. Caching was particularly useful for plug-ins that accessed the same sets of addresses frequently, such as *linux_lsof*. Furthermore, all the plug-ins frequently requested only small amounts of bytes at a time from the memory during the analysis. For example, the plug-in *linux_pslist* requested about 93 KB of data in total, 4 bytes in average, and due to caching we reduced this amount to about 12 KB. The plug-in *linux_iomem* requested about 58 KB of data in total, 9 bytes in average, and due to caching we reduced this amount to about 30 KB. The application of other plug-ins yields comparable results. Considering the time required for dump file creation in other approaches, our framework can even provide faster results.

4.6.2.2 Acquisition of Sensitive User Data

We show the potential of our framework for data acquisition at the example of launching a cold boot attacker after a real user session. Since we demonstrated that we are able to retrieve FDE keys with the BMA, see Section 4.6.1, we focus on other sensitive assets in the following. There is way more confidential information to detect, which is possibly never persisted and can only be found in main memory. Therefore, we created a potential usage scenario where the user enters confidential data on the phone, which is masked in the following. After the scenario, we reset the device, flashed and booted the BMA. Then, we started an investigation using various Volatility plug-ins.

The scenario starts by booting the phone, using it for approximately 15 minutes and ends after leaving it idle for about one minute. During that time the user carries out the

Listing 4.2: Truncated output of the Volatility plug-in *linux_pslist* using a cold boot attack for data acquisition.

```
Offset     Name            Pid  Uid   Gid   ...
0xc000e000 init              1 0     0
0xde9da400 keystore         227 1017  1017
0xdd25ac00 d.process.acore  798 10003 10003
0xdcea3000 m.android.phone  828 1001  1001
0xdbf8e000 m.android.email 1480 10032 10032
0xdbeb7c00 droid.gallery3d 1505 10035 10035
0xdc1a8400 ndroid.exchange 1522 10033 10033
0xdcd0d000 ndroid.contacts 1689 10003 10003
0xdbb99c00 mod.filemanager 2028 10022 10022
0xdca5bc00 android.browser 2364 10020 10020
```

Listing 4.3: Truncated output of the Volatility plug-in *linux_lsof* using a cold boot attack for data acquisition.

```
Pid FD Path
828  0 /dev/null
828 70 pipe:[12680]
828 71 /data/data/com.android.providers.telephony/databases/mmssms.db
828 88 anon_inode:[4225]
828 89 /data/data/com.android.providers.telephony/databases/telephony.db
```

following activity:

- Create a new contact *Secret Contact* with phone number *017** in the contacts application.
- Synchronize a previously set up exchange account within the mail application.
- Create a draft short message *Top Secret Short Message Draft* to *Secret Contact* using the messenger application.
- After a while, edit the stored short message draft to *Top Secret Message* and send the message.
- Visit webpages with the browser, use search engines. Login to pages with a user account and enter confidential data, such as *Top Secret Information*.
- With the filemanager, create a new file */data/secret.txt* and edit the file with the content *Top Secret Text*.

As a first step of the analysis, the investigator with physical access to the device retrieves the process list with *linux_pslist*, see Listing 4.2. The full list has 239 entries in total. The amount of processes that an investigator suspects confidential data to be contained is way smaller. Inspecting the open file handles of the phone process `com.android.phone` with the plug-in *linux_lsof* reveals the potential sensitive file *mmssms.db*. The truncated list is depicted in Listing 4.3. In total, the plug-in finds 90 open files, but most of them can be left out of consideration.

Listing 4.4: Truncated output of a file acquired with the Volatility plug-in *linux_find_file* using a cold boot attack for data acquisition.

```
%004917*
h13Top Secret Short Message Draft
Top Secret Message
004917*
Top Secret Short Message Draft
```

Using the plug-in *linux_find_file*, we searched the corresponding `inode` and retrieved the cached file contents of about 103 KB. By dumping the strings of the read file, we obtained about 105 strings. This helped us to quickly recognize the recipient, the initial draft and the edited message, see the truncated output in Listing 4.4. The plug-in *linux_lsof* is especially useful for determining open files of processes, such as logs.

As a next step of the analysis, we retrieved the memory segments of the process `com.android.exchange` using the plug-in *linux_proc_maps*. We suspected relevant data of the process to be located in the processes' heap segment. Listing 4.5 shows the output cut to the lines containing the keyword heap.

Using the plug-in *linux_dump_maps*, we retrieved the heap segments. The strings in the dalvik-heap segment quickly revealed the mail account's username and password, separated by a semicolon *firstname.lastname@*.de:**. Even though the dalvik-heap segment seems to be large, the request for the segment was quickly handled, because Volatility recognizes that the segment is sparsely allocated.

We conducted the same steps for the process `android.process.acore`, which serves Android's contact provider, for `com.cyanogenmod.filemanager` and for `com.android.browser`. Inside anonymous memory segments, we were able to find the contact *Secret Contact* with phone number *017**. The browser's memory segments contain vast amounts of loaded websites, user account names, search queries and text entered in webmail and social media

Listing 4.5: Snippet of the output of the Volatility plug-in *linux_proc_maps* for the Android exchange process, using a cold boot attack for data acquisition.

```
PID  Start      End        Flags Pgoff Major Minor Inode Path
1522 0x41a22000 0x41a2a000 rw-   0x0   0     0     0     [heap]
1522 0x41e82000 0x61a2a000 rw-   0x0   0     4     8872  /dev/ashmem/dalvik-heap
```

Listing 4.6: Snippet of the output of the plug-in *linux_route_cache* using a cold boot attack for data acquisition.

```
Interface Destination      Gateway
--------- ---------------- -------
wlan0     131.159.0.91     10.144.207.1
wlan0     173.194.112.136  10.144.207.1
wlan0     131.159.0.91     10.144.207.1
lo        10.144.207.39    10.144.207.39
```

pages. This made it possible to recover entered data, such as *Top Secret Information*. The dalvik-heap of the filemanager exposes the filename */data/secret.txt* and its content *Top Secret Text*.

In a further step of the investigation, we inspected the data in the routing table cache with the plug-in *linux_route_cache*. We recovered the hosts we recently connected to during our browsing session, such as our webmail page, see Listing 4.6.

In order to successfully and efficiently carry out an analysis, the investigator has to be aware of where Android processes store their relevant data. Open file handles and the dalvik-heap are the most probable locations to expose such data. We were in knowledge of the data we were searching for in our scenario. However, relevant processes and data can be relatively quickly identified and filtered from memory dumps.

4.7 Discussion

As we read memory from a cold booted device, we need to be aware of the decay of data. In case of corrupted data, this could lead to the situation where the pointers in the structure `task_struct` of the Linux kernel cannot be correctly dereferenced, for example. This causes the forensic analysis to fail at some point, because previously running tasks cannot be detected. In order to treat these cases, we propose to extend forensics applications to work in combination with corrupted data acquired by a cold boot attack. Heuristics can help fixing invalid pointers, or to at least ignore them. However, our data remained in almost all of our test cases intact so that we did not have to deal with this problem. This is due to the reset attack where the battery is not removed. The feasibility of the reset attack depends on whether the specific device offers the hardware reset functionality or not.

Another important aspect is that the target device must expose a UART port for allowing a simplistic BMA implementation. A lot of devices have it even though it is not visible at first glance. The UART port is often integrated into the Micro-USB port or the headphone socket.

Care has to be taken considering the bootloader. Using the Samsung Galaxy S4 device, the bootloader accepted a simply crafted boot image, but the requirements changed for the Nexus 5 bootloader. To figure out what the bootloader requires is not always obvious, but open source bootloader code helps to recognize such requirements [Lit]. Fortunately, most of the mobile device bootloaders work similarly. Nevertheless, it is possible that bootloaders are capable of overwriting volatile data. This would represent an inevitable problem and be a possible mitigation on mobile devices.

The deployment of the BMA onto the device either requires direct write access to the recovery partition at runtime or at bootloader that can be unlocked for flashing the BMA. However, write access is only possible with root privileges. In case the bootloader is locked, it has to be unlocked before flashing partitions. Unlocking the bootloader normally leads to erasing all user data on persistent storage. Persistent memory can then no more be recovered. But with our method, volatile memory remains unimpaired when unlocking the bootloader and we do not require root privileges on the phone. This means that we are still able to recognize crucial contents of the previous session in main memory, which were possibly never made persistent.

We expect that our framework can be used for further scenarios, because our implementation is easy to extend and can be easily ported to other devices. The framework can be used, for example, to further evaluate whether it is possible to access application memory running in the secure world of the TrustZone [ARM09].

4.8 Summary

In this chapter, we tackled Challenge 3 by introducing a novel forensic framework for mobile devices based on the cold boot attack. In contrast to other state-of-the-art techniques, we do not boot a full-fledged Linux kernel on the target device. Instead, we boot our easily portable, minimal BMA, which occupies no more than three kilobytes in main memory. The BMA preserves the data structures of the previously running kernel and does not initialize device memory. As we only overwrite constant data in the kernel code section, this ensures that all of the important kernel data remains available for analysis. The BMA provides a serial communication interface. This interface allows to dynamically request parts of the main memory. Forensic analysis can thus be conducted on the host system. For this purpose, we extended Volatility with a serial communication module for the systematic analysis of the target device's memory. The framework with the implementation of the BMA and with the host-based model for the systematic analysis represented our Contribution 5.

We realized the framework for the Samsung Galaxy S4 and ported it to the Nexus 5 device in order to demonstrate the feasibility of our approach. In our evaluation, we compared our cold boot-based analysis with traditional memory dump analysis using Volatility, providing proper results. We have shown that our BMA allows to request full, genuine memory dumps and have demonstrated how to efficiently gather vital information based on the sustained kernel structures, such as FDE keys and further confidential data. We have also shown that we were able to extract the FDE keys of containers running on our secure virtualization platform from Chapter 3. This motivates the urge for techniques to mitigate the effect of physical attacks, such as main memory encryption for devices susceptible to cold boot attacks. In the next chapter, we provide several architectures for main memory encryption for different types of devices.

CHAPTER 5

Architectures for Main Memory Encryption

In this chapter, we pursue Challenge 4, the design of architectures for main memory encryption, for which we present with Contributions 6 to 9 different architectural solutions to defend against the physical attacker from Chapter 2.

We first present two suspension-based memory encryption architectures where the system, or parts of the system, suspend before the memory encryption. Contribution 6 focuses on traditional computing devices like laptops or desktop PCs with en- and decryption tied to full system suspension and resumption. The architecture for Contribution 7 allows for the suspension and encryption of arbitrary process groups, and hence containers. The flexible encryption of only parts of the system adds complexity to the architectural design. Some process groups might remain unencrypted while others are in turn encrypted. This, and the group formations may dynamically change. This is why the encryption procedure must consider memory shared between arbitrarily formable process groups and ensure that the process space of non-suspended processes remains unencrypted. We describe the concepts and implementation of this architecture in more detail in Section 5.3, and the architecture for full system suspension in Section 5.2.

We present Contribution 8 by combining the main memory encryption architecture for containers with the secure virtualization architecture for mobile devices from Section 3.3. This enables main memory encryption for suspended Android containers. In combination, these architectures defend both against local and remote attacks during runtime, and against physical memory attacks. This results in a container-based platform achieving a high level of data confidentiality against the attacker types defined in Chapter 2.

In contrast to Contribution 6 and Contribution 7, we present a runtime memory encryption architecture with Contribution 9. This architecture does not rely on suspension, but transparently encrypts all main memory of a system. We base this architecture on a minimal HV, at the example of ARM mobile devices with the ARM TrustZone.

This chapter is organized as follows. We first describe related work in the field of main memory encryption in Section 5.1. Then, we present our suspension-based architectures with the integration in real-world scenarios in Section 5.2 and Section 5.3. After that, we describe our runtime memory encryption architecture in Section 5.4 and summarize the chapter in Section 5.5.

The comprised Contributions 6 to 9 can also be found in the publications in [Hor17; Hub17a; Hub18; Hub17b]. The contributions emerged from joint work with Julian Horsch, Sascha Wessel and Junaid Ali. Julian Horsch and Manuel Huber jointly worked on the concepts, implementations and evaluation. Sascha Wessel supported in the work on

the concepts. Julian Horsch mainly contributed to [Hor17]. Junaid Ali contributed to implementational aspects as part of his master's thesis.

5.1 Related Work

In the following, we discuss related work on main memory protection. We start with key hiding techniques, describe hardware- and software-based runtime memory encryption techniques, and present suspend-time memory encryption approaches.

Key Hiding

The following approaches only protect a specific key, for example, the FDE key, in RAM from memory attacks. Approaches for x86 [Gua14; Gua15; Mül10; Mül11; Sim11], as well as for ARM [Göt13] and HVs exist [Mül12]. The approaches either store the key in the CPU/GPU registers, or in the CPU cache, and implement the cipher associated with the key on-chip at the cost of performance. These approaches leave all other assets in RAM unprotected and are hence vulnerable to memory attacks. Hiding or isolating key material from possible memory attacks forms the basis for runtime memory encryption where the memory encryption key must be insusceptible to memory attacks. Some of the software-based runtime memory encryption techniques make use of the proposed key hiding techniques in order to secure their encryption key.

Hardware-based Runtime Memory Encryption

Several hardware-based memory encryption architectures, such as Aegis or XOM, have been proposed [Che08; Duc06; Gut99; Lie03; Lie00; Mau84; Su09; Suh03; Suh07; Suh05; Wür16; Yan03]. These approaches are capable of protecting all main memory. Sensitive data of protected processes is unencrypted exclusively in the processor chip, which is the single trusted component. These hardware architectures are difficult and expensive to realize and usually not available on common consumer devices, but designed for special purposes, such as Digital Rights Management (DRM) protection.

The recently launched AMD SEV technology is designed to encrypt the main memory of x86 AMD virtual server systems [Kap16], protecting from memory attacks and a possibly malicious HV. SEV builds on AMD Secure Memory Encryption (SME). SME makes it possible to transparently encrypt the whole main memory of an x86 system based on an Secure Processor (SP) isolated from the rest of the system. When using SME, the SP creates an ephemeral RAM encryption key and en-/decrypts each page before being stored in main memory, respectively before being loaded from main memory into the CPU. For SEV, the SP ensures to create different main memory encryption keys for each VM and encrypts their memory independently. Research has highlighted different attack vectors on AMD's SME and SEV [Buh17; Het17], which we are going to focus in more detail in Chapter 6. The crucial point about AMD SEV is the missing integrity protection of the encrypted memory pages, making it possible to modify the VM's memory contents or change memory mappings with a HV. This allows, for instance, to alter code execution flow [Het17], or to extract the contents of main memory tricking a service inside the VM into rendering arbitrary VM data in plaintext to the outside [Mor18].

Intel announced an equivalent, Intel Total Memory Encryption (TME) and Multi-Key

TME (MKTME) for server systems [Int]. While Intel has not clearly pointed out an attacker model so far, MKTME does not seem to protect VMs against a malicious HV. The HV retains more privileges regarding key management, and can handle the sharing of memory between VM, or turn off encryption of pages, for instance. The technology also allows for tenant-provided keys for encryption, for example, for NVRAM, and enables the encryption of I/0, of which AMD's SEV is so far not capable. Like SEV, MKTME protects from cold boot and DMA attacks, but the HV remains the trusted entity managing VM encryption. Note that both technologies are not designed to protect VMs from attacks vectors like Spectre or Meltdown.

There also exist processors for consumer devices with extensions to provide secure enclaves, which can be leveraged to thwart memory attacks, such as the ARM TrustZone, or Intel SGX [ARM09; McK13]. These enclaves constitute hardware-protected memory areas to which the OS can move sensitive data, but are limited, for instance, regarding the size of the memory areas. It is thus common to only move smaller amounts of sensitive data of processes to an area in the enclave's hardware-protected memory. Developers need to specifically design enclave-aware, hardware-dependent software and the underlying OS must support the processor extensions. These extensions themselves thus represent building blocks that can be leveraged to protect a system, for instance, for secure encryption key storage while realizing memory encryption inside an enclave.

Software-based Runtime Memory Encryption

We classify software-based memory encryption approaches into either runtime encryption techniques where main memory is encrypted throughout process runtime, like our architecture in Section 5.4, or into suspend-time techniques which rely on process suspension in order to encrypt memory, as presented in Section 5.2 and Section 5.3.

Cryptkeeper [Pet10] is an extension of the virtual memory manager to reduce the exposure of unencrypted data in RAM. The mechanism separates RAM into a smaller unencrypted working set of pages, called the *Clear*, and an encrypted area, the *Crypt*. The Clear is a sliding window of unencrypted pages directly susceptible to memory attacks. By the time the Clear fills up, pages are automatically swapped into the Crypt and decrypted on demand. The Clear is an area susceptible to memory attacks.

HyperCrypt [Göt16a] transfers the concept of Cryptkeeper into a HV to transparently encrypt the full memory of a guest OS. Such approaches require a secure key storage location and isolated encryption environment for the encryption. For full memory encryption techniques, there is a notable performance impact on the system, and the mechanisms usually keep an undefined amount of RAM unencrypted. This means that there is only a statistical probability that the sliding window contains solely non-sensitive main memory contents when a memory attack is executed. In contrast to techniques suspending the system, an attacker could time an attack by observing a running device waiting for a suitable moment where sensitive material is likely unencrypted. Furthermore, compared to approaches where memory decryption is only possible when a user provides a legitimate token, an attacker can also try to gain control over the device via privilege escalation. Even though the attacker might still be unable to read the encryption key, for example, when stored on-SoC, the attacker controls the main application processor and can load any

page unencrypted into the sliding window. As the encryption key is usually ephemeral for main memory encryption and valid only during a particular boot cycle, the extraction of encrypted data itself poses the main target of attackers.

RamCrypt [Göt16b] is an encryption approach for x86-based Linux systems and also transparently encrypting the memory of running processes. For key-hiding, RamCrypt stores the memory protecting key in processor registers. Therefore, deep interference into the kernel's page fault handler is necessary to encrypt pages and to decrypt them when accessed. Processes to be protected have to be marked a priori by setting a flag inside the ELF program header. RamCrypt encrypts anonymous non-shared segments only and there also remain unencrypted pages in the sliding window. This means that, for instance, any file-backed resource in the page cache is not encrypted, leaving a considerable amount of memory unprotected. The cost of encrypting pages on the fly comes with a notable performance impact. As with all other runtime protection mechanisms, a physical attacker gaining privileges on the system can request the decryption of encrypted memory on the main application processor.

Another approach for x86 platforms is presented in [Pap17], which allows full and process-selective memory encryption. The authors store the memory encryption key on-chip and instrument load and store instructions to en-/decrypt main memory when being stored/loaded in/from main memory. For process-selective encryption, the authors propose a new memory allocation for programmers to manage granular application-specific sensitive memory regions. The approach uses process-specific encryption keys, which are wrapped with a master key stored in a pair of the processor's debug registers. In case of full memory encryption, the sliding window never renders sensitive data, but causes significant performance overhead, for instance, about 17% to 27% for HTTP and HTTPS web server applications.

The authors in [Che08] use the processor cache as a secure storage via cache locking. The objective is to protect the processor-memory bus against physical attackers. This results in encrypted RAM for protected processes. However, this idea only exists in full system simulation. An implementation strongly depends on the OS cache routines and locking functionalities adversely affecting performance.

Approaches specifically tailored to the mobile domain have also been developed. Sentry [Col15] presents a runtime memory encryption concept for Android devices. The user has to mark sensitive apps and OS subsystems in the settings menu. When the device gets locked, the mechanism encrypts specific memory of the chosen apps. Sentry creates the cipher key on boot and stores it On-SoC in the iRAM. For apps that run while locked, Sentry reads encrypted memory pages from RAM, decrypts and keeps them inside the ARM SoC with cache line locking. Pages are encrypted on a page-out before writing them back to RAM. With increasing background activity and a full cache, the performance strongly degrades since the mapping of called-in pages to the cache triggers costly page faults. The approach uses ARM specific (legacy) mechanisms originally designed for embedded systems. Hence, the feasibility strongly depends on the architecture and platform specific hardware features. Their prototype on the mobile device does not support cache locking and does hence not realize full runtime encryption nor support background activities, such as incoming phone calls or network data.

CleanOS [Tan12] is a memory encryption mechanism integrated into the Android framework. This approach only works in combination with trusted, cloud-based services for key management to which the phone needs connectivity and only covers parts of all the possibly sensitive data. The *idle eviction* mechanism encrypts data that is not in active use. Afterwards, the key is purged on the device and fetched on-demand from the cloud. The main modifications were made by the introduction of Sensitive Data Objects (SDOs), which represent sensitive user data, and a special garbage collector, eiGC. The latter searches and encrypts SDOs that were not used for a specific period of time. Apps either implement an SDO API to add and register SDOs, or the framework registers default SDOs along with heuristics to identify SDOs. To decrypt the SDOs, a modified Dalvik interpreter faults and retrieves the key from the cloud. Not only regarding the scope of encrypted memory, or key retrieval and availability concerns, but also due to the heuristics and workload to adjust apps, this mechanism represents a scheme prone to leaving sensitive memory unencrypted.

In contrast to CleanOS where the encryption key is offloaded into the cloud, TinMan [Xia15] evicts security-critical code and data to the cloud, making it unavailable on mobile devices. This represents another way of mitigating the effect of memory attacks on specific secrets worth of protection.

In Section 5.4, we introduce a transparent runtime memory encryption architecture with a minimal HV, allowing encryption of unmodified guest OSs on ARM platforms with high throughput, not requiring special registers or hardware-specific mechanisms, using ARM TrustZone as secure environment for key storage and cipher computation. We oppose runtime and suspension-time encryption approaches more to each other in the next paragraph.

Suspend-Time Memory Encryption

Hypnoguard [Zha16] en- and decrypts main memory during OS suspension and wakeup on x86 platforms. The mechanism hooks into the ACPI S3 suspend/wakeup procedure at stages where the OS is not active. At this point, there is no support for hardware devices, such as (VGA/HDMI) displays or (USB/Bluetooth) keyboards. Therefore, their design requires to implement highly hardware-specific crypto routines for hardware accelerators and for drivers to interact with hardware devices, for example, for passphrase input. The encryption key is bound to a TPM, which is used to wrap and unwrap the encryption key making it only present during en- and decryption. The encryption is executed in Intel's Trusted Execution Technology (TXT) environment [Gre12]. This makes the approach in contrast to Section 5.2 more hardware-specific and particularly cumbersome for portability and for interacting with different hardware devices.

In Section 5.2, we focus on the same goal, but design a hardware-independent OS kernel mechanism, which can be integrated into deployed Linux systems. The design extends the scope of the FDE key using it to also encrypt main memory during suspension. After main memory encryption terminates and the system is suspended, the key is removed from main memory. The key is regenerated on wake-up with a passphrase query asking the user for the FDE key. The concept allows for binding the FDE key to a TPM remediating brute-force attacks on the FDE key.

Both the approaches were implemented on traditional x86 computing devices where background activities during OS suspension are not a focal point. Mechanisms designed for mobile devices, as presented in Section 5.3, must however sustain their ability to process background activities, such as incoming data or phone calls.

Transient Authentication [Cor03] is also an approach for x86 platforms to protect processes transparently, but depends on the presence of a hardware token [Cor02]. The token provides fresh cryptographic keys. The concept comes with two protection variants, the *application-transparent* and *application-aware* protection. In the first mode, the system suspends and encrypts in-memory pages when the user removes the token. The only processes that remain running are tasks for transient authentication and OS threads. Despite that the OS is not completely suspended and that management tasks continue running, suspending/waking takes about 8 seconds. Therefore, an application-aware mode is proposed. In the application-aware mode, software developers are responsible for protecting specific applications by utilizing a special API. This allows to selectively protect chosen assets, such as an application's secret key. However, this requires in contrast to our approach new software to be specifically tailored to the approach and existing applications to be modified.

Suspension-based encryption has the advantage that the encryption key must not necessarily be constantly present like in the case of runtime memory encryption mechanisms, but present only during en- and decryption. This allows for more flexibility regarding key management compared to runtime-memory encryption. Further, even when the software stack gets exploited, encrypted information cannot be decrypted due to the unavailability of the encryption key on the system. For these reasons we introduce in Section 5.2 and Section 5.3 two suspension-based mechanisms that can easily be added to existing infrastructures. Section 5.2 can be used to secure traditional desktop devices - in contrast to the existing approaches - with only minimal changes to the software stack and without hardware dependencies. With our mechanism in Section 5.3, we enable the encryption of individual process groups' main memory. This stands in contrast to existing approaches which work on (parts of) the system as a whole, and can thus be applied to containers. In particular, we apply this mechanism to our OS-level virtualization architecture. For systems where suspend-time encryption is not applicable, we propose a transparent runtime memory encryption architecture with a minimal, almost completely guest-agnostic hypervisor for the ARM platform in Section 5.4.

5.2 A Main Memory Encryption Architecture for Suspending Devices

This section proposes Contribution 6, a lightweight architecture for main memory encryption that safeguards unattended devices, such as desktop PCs or notebooks, from memory attacks when a device is fully suspended. This prevents attackers with physical access to an absent user's device from disclosing the secrets in main memory. When the user or the idle system suspends a device, i.e., on Suspend-to-RAM, we encrypt the confidential data in main memory with the FDE key and remove the key afterwards from main memory. We request a passphrase from the user to restore the FDE key and initiate the decryption while waking up the system. The architecture ensures to protect all user space process memory, the valuable assets in kernel memory and ensures to remove other sensitive memory

remnants, such as cipher states. Our primary design characteristics are the following:

Lightweightness. The architecture does not require configuration from the user, leverages the existing OS infrastructure and can be easily integrated into systems in use.

Hardware Independence. The implementation is hardware independent and thereby ensures simple portability to other devices.

Usability. The user is not adversely affected in the workflow and is only required to type the FDE passphrase upon wakeup.

Performance. The implementation sustains the almost seamless suspend/wakeup cycles and uses hardware accelerators for encryption and decryption, if present.

We implement our architecture as a kernel mechanism for x86-based Linux devices and current kernel versions, but its design can likewise be transferred to other OSs and platforms. Additionally, we propose a design variant using Linux as a HV with which protect the full memory space of guest OSs, such as Windows. We also conduct a representative performance evaluation of our x86-based prototype and discuss the architecture's security aspects.

This section is organized as follows. First, we present the design and implementation of our architecture for Linux-based systems in Section 5.2.1 and Section 5.2.2. In Section 5.2.3, we present the design variant with a virtualized environment to protect full guest OSs. We evaluate the performance and security of our prototype in Section 5.2.4.

5.2.1 Basic Design for the Protection of Linux-based Systems

Today's Linux devices usually use FDE in combination with LUKS to protect the confidentiality of persistent data. Our idea is to extend the coverage of FDE encryption of the data (and swap) partition to main memory contents when the device is suspended. When powering on, the system boots normally, i.e., the user is queried for the FDE passphrase after starting the initial ramdisk. By entering the valid passphrase, the system derives the FDE key and boots the OS. Figure 5.1 shows how we integrate our idea into the wake and suspend transitions of an existing system. While the device is actively used, the processes are not encrypted and the FDE key is present to transparently encrypt storage. After an idle time or active suspension, the common ACPI S3 suspend procedure starts. The kernel freezes its tasks, i.e., processes and threads, concurrently by sending every task a signal. This causes them to switch from user space execution to entering a non-schedulable state in kernel space.

At this point of state transition, we make each user space process encrypt its associated memory regions in its own context, i.e., anonymous, mapped, or shared memory and so on. Threads share the memory regions and kernel structures with their parent processes and siblings. To avoid multiple encryptions of the same regions, only the last task (of a process) to enter the *frozen* state encrypts the associated memory regions. A further challenge is that the physical pages to be encrypted can be shared across process boundaries. Therefore, every encrypting task marks its present pages as *encrypted*, if not marked before. Other tasks considering to encrypt a shared page skip already marked pages. As soon as every process has finished encryption, we purge the FDE key and further kernel memory possibly

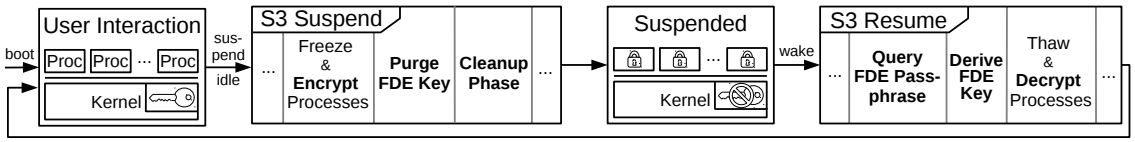


Figure 5.1: Overview on the steps (marked in bold face) introduced to the ACPI S3 suspension and resumption procedures.

containing sensitive data in a cleanup phase. After that, the common S3 suspension continues. All confidential data on the device is encrypted or removed.

Upon waking up the device, the S3 resumption is triggered. Before thawing (i.e., waking) the processes, we query the FDE passphrase from the user, as Figure 5.1 depicts. A secure routine takes the passphrase and derives the key with the Password-Based Key Derivation Function 2 (PBKDF2), which is used to decrypt LUKS headers. The decrypted LUKS header contains the FDE key, which we re-supply to storage encryption (`dm-crypt`) and use for the decryption of the processes while thawing (i.e., resuming) them. During decryption, each previously encrypting task decrypts the same set of pages and resets their flags to *decrypted*. After the decryption, the S3 wakeup continues and the system gets back to operation. The mandatory passphrase query allows to disable the screen lock after suspend requiring the end user to enter only a single passphrase.

We intentionally did not refer to a TPM or to other SEs for key protection to present a hardware-independent solution, especially for cases where a TPM is not required, available, or admitted. The security of the FDE key hence correlates with the complexity of its associated passphrase. However, a TPM can be easily combined with LUKS [Yod]. This implies that our proposed design can be easily adapted for cases where a TPM is desired and thus prevent brute-force attacks on the FDE key. In addition, the concept does not depend on processor extensions, such as Intel TXT or AMD Secure Virtual Machine (SVM), to protect the system.

5.2.2 Implementation

We implemented a prototype for recent Linux kernel versions (version 3.16 and 4.5, 32 and 64 bit, for example). In this section, we focus on the three crucial steps of the implementation: The process *en-/decryption* procedure, the *cleaning* of further sensitive data and the *FDE passphrase query* for restoring the FDE key.

5.2.2.1 Process Encryption and Decryption

During suspension, every frozen task increments a counter we integrate into a structure shared with all tasks of a process. Every frozen task then compares the incremented counter with the number of tasks associated with the structure. In case the values equal, a task marks itself as the en-/decrypting task and starts the encryption of the Virtual Memory Areas (VMAs). The VMAs represent the user space memory regions, such as the stack, heap, code, shared and further anonymous or mapped memory segments.

The tasks responsible for en-/decryption iterate over their VMAs and asynchronously en-/decrypt the associated present pages using the kernel crypto API on all available

cores. The only exception for excluded VMAs are non-confidential ones containing special segments and shared library code. The latter segments solely contain constant read-only data while special VMAs map memory shared with hardware devices, for instance, memory-mapped I/O or DMA memory. These VMAs can not be encrypted, because devices are not aware of the encryption and writing that memory likely corrupts the system. Encrypting the whole process memory as a single chunk would compromise these segments and fault on non-present pages.

The kernel considers a task frozen/thawed when all requests of a task are processed. We use AES in CTR mode as cipher and physical addresses as Initialization Vectors (IVs). Hence, we provide different IVs for all blocks to be encrypted. The crypto API selects the preferred cipher driver and available hardware accelerators, AES-NI instructions in our case.

5.2.2.2 Cleanup Phase

After the encryption, sensitive data may still persist in regions not covered by the tasks' encryption. First, we zero out the FDE key. Second, we remove the remnants of the used ciphers: we zero the utilized cipher structures and also the relevant kernel stack regions using the kernel stack pointer. This makes it infeasible to deduct the encryption key from such information. Third, sensitive data may still exist in free pages, which were previously used by the kernel or user space processes and then freed. To remove these contents, we walk the list of free pages maintained by the page allocator and zero them out, which can be accomplished with very high throughput [Col15]. This step can be omitted when secure deallocation [Cho05] is in place.

5.2.2.3 Passphrase Query

We implement a simple passphrase query in the kernel comparable to [Mül11] and derive the FDE key based on the supplied passphrase. The kernel and its drivers are fully operational at this stage and only user space processes are frozen. We easily reuse the drivers and software stacks in the kernel to display the user a password prompt and to utilize connected keyboards. For using a sophisticated GUI and for re-plugging input devices, for example, (bluetooth) keyboards, during suspension, it is possible to keep relevant processes (`udevd`, `bluez`) unencrypted and to thaw them in time when ensured that the daemons keep no confidential data. This also allows for using SEs, for instance, a smartcard, for two-factor authentication.

5.2.3 Design Variant for the Protection of Other Operating Systems

Our basic design can be easily transformed into a variant which allows to protect the full memory of other OSs, such as Windows. Figure 5.2 gives an overview on the design variant of our architecture, where a minimal Linux serves as a HV for guest OSs using Kernel-based Virtual Machine (KVM). The gray elements in Figure 5.2 constitute optional components, such as multiple guest OSs on the system. The underlying Linux system only appears to the user when asking for the FDE passphrase on boot and wakeup. At any other time, end users work with their OS of choice with almost native performance. This makes it possible to, for example, transparently protect Windows systems without impairing the usability for Windows users. When the device and hence the guest OS gets

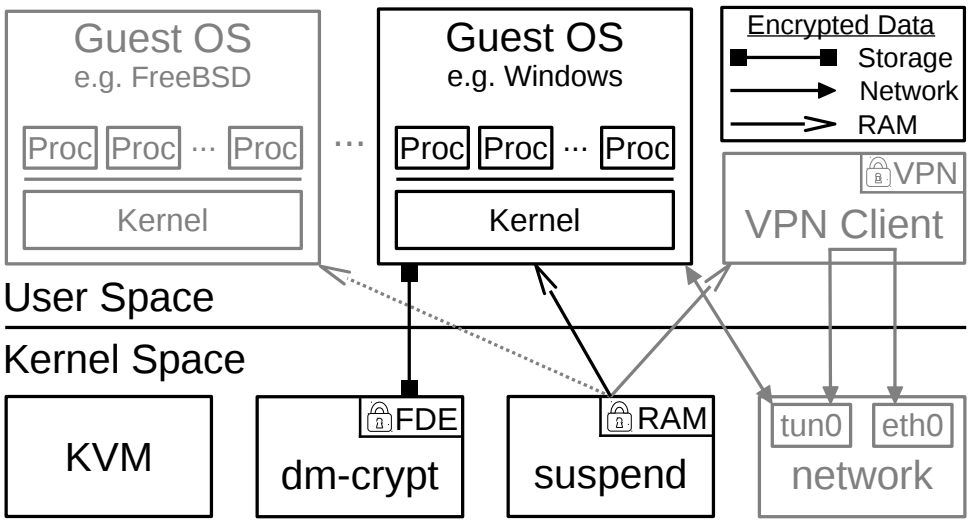


Figure 5.2: Design variant of our architecture to secure virtual guest OSs with KVM.

suspended, the full memory of the guest (including its kernel memory) is hence encrypted with our underlying mechanism. The FDE, i.e., `dm-crypt`, transparently encrypts storage and because we purge the FDE key after suspension, the guest OS's storage is fully secured from memory attacks. In addition, there is no confidential data stored in the Linux HV itself.

To further protect the system, the optional components in Figure 5.2 emphasize that the guest's network traffic can be transparently protected by a user space Virtual Private Network (VPN) client, for example, OpenVPN. The VPN client encrypts and routes the guest OS's network data over a secure VPN tunnel, for instance, through a corporate network. The Linux HV automatically establishes the VPN connection such that common end users do not need to provide further credentials and always benefit from the secure connection. Since the network credentials are part of a host user space VPN process, the corresponding key is also encrypted during suspension.

5.2.4 Evaluation

In the following, we first present our performance measurements before evaluating the security.

5.2.4.1 Performance Measurements

We ran a Debian userland with a Linux 4.5 kernel (4 KB page size) on a Lenovo T450s notebook (Intel Core i7-5600U CPU @ 2.60 GHz, dual core; 12 GB DDR3 RAM 1600 MHz). To fill and stress the RAM in a representative manner, we deployed a desktop environment and installed numerous applications, including business and graphics tools, browsers and virtual machines. Figure 5.3 shows the timing of 100 suspend and wakeup cycles of different sessions we measured using AES in CTR mode with a 256 bit key size. After booting the system, the suspension and wakeup of processes took about 150 ms, where about 125,000

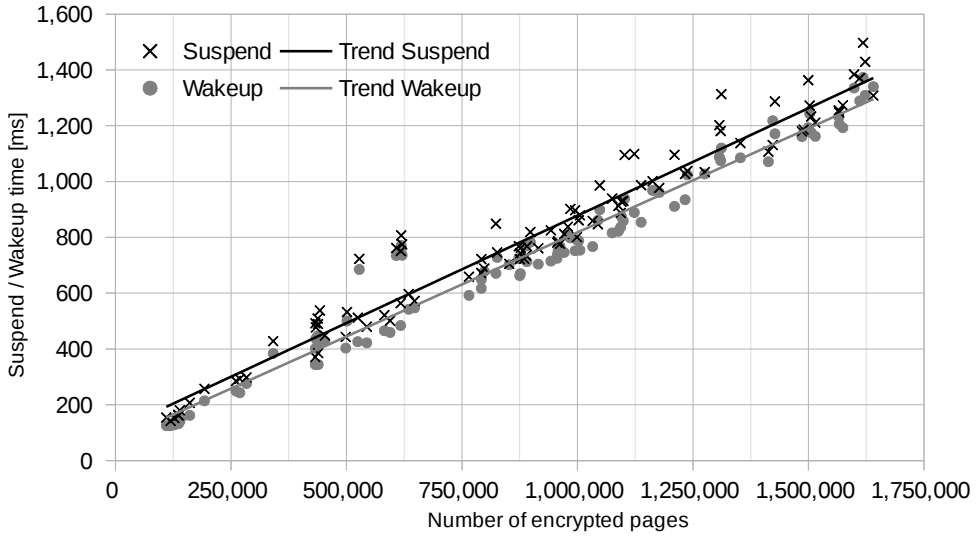


Figure 5.3: Suspend (crosses) and wakeup (circles) times in milliseconds in relation to the number of en-/decrypted pages.

pages (500 MB) were encrypted. The trendlines show the linear increase in suspension and wakeup times when loading more and more applications and data into RAM. When the memory is under high load, i.e., 10-12 GB reserved, the suspension/wakeup took about 1.3 s. In such cases more than 1,600,000 pages (6,4 GB) were encrypted. The non-encrypted memory consists of non-confidential VMAs (special mappings and library code) and kernel memory. With our encryption model, we have fine-grained control over the regions requiring protection and do not need to encrypt the full space. Table 5.1 averages all measurements from Figure 5.3 and shows minimum and maximum values. In an average suspension cycle, 105 processes were suspended which encrypted about 900,000 pages. One cycle involved the encryption of more than 20,000 out of total 25,548 VMAs. The average suspend time was 807 ms and even faster 745 ms for the wakeup. The prototype virtually reached an en-/decryption speed of 4,453 MB/s, resp., 4,824 MB/s. Compared to the maximum of about 2.6 GB/s on one core (measured with `tcrypt` benchmarks), our prototype performs well and the system remained fully stable at all times. Ideas for future speed-ups are, first, to zero out and unmap page-mapped files instead of encrypting them, and, second, to decrypt encrypted pages on the fly after the wakeup. However, both approaches would affect runtime performance, because the mapped pages must be reloaded from persistent storage and in the latter case, pages are decrypted on demand.

5.2.4.2 Security Discussion

In the following security discussion, we consider the attacker types as defined in Chapter 2. This includes our descriptions on the local, remote and physical attacker and our three presented attack scenarios. We assume that the attacker attempts to gain confidential data when a suspended device, not previously tampered with, is unattended. According to Chapter 2, the attacker has sufficient time and can exploit software and hardware

Table 5.1: Minima, maxima and averages over all test runs regarding encryption time, throughput, and the number of encrypted processes, VMAs and pages.

Measurement	Minimum	Maximum	Average
Processes	83	127	105
VMAs Encrypted	11,899	28,245	20,379
VMAs Total	15,241	35,050	25,548
Pages Encrypted	110,148	1,640,439	898,384
Suspend Time [ms]	142	1,497	807
Wakeup Time [ms]	125	1,373	745
Enc. Speed [MB/s]	2,860	5,106	4,453
Dec. Speed [MB/s]	3,081	5,387	4,824

vulnerabilities, for example, via JTAG [Wei12], cold boot [Gut01; Hal09; Mül13], or DMA [Bec05; Boi06; Dev09; Ste13], to gain access to both volatile and persistent memory. However, we make the assumption that the attacker cannot modify the memory to execute evil maid attacks waiting for the user to return (for the latter, we refer to system hardening techniques). A device once tampered with is hence no longer trusted, for instance, after longer absence through theft, loss, or because the user notices the tampering attempt. This either results in an attacker conducting a reading memory attack, see Chapter 2, or in a writing memory attack that is however apparent to the victim. With physical possession of the suspended device, our considered attacker trying to obtain confidential data can possibly access persistent and volatile memory, categorized as follows:

Persistent Memory. Storage is protected by FDE and we removed the FDE key from kernel memory. The only way to decrypt storage is to decrypt the persistent LUKS header of the storage volume. This depends on the complexity of the FDE passphrase. Brute-force attacks take much effort, because the header decryption key is derived using the slow PBKDF2. A high number of iterations for PBKDF2 further slows down the repetition rate. In our design, we emphasized the possibility to easily integrate a TPM or to use an SE to prevent brute-force attacks.

Process Memory. All confidential VMAs were encrypted using AES in CTR mode with unique IVs. For the encryption, the FDE key was used and removed afterwards. The efforts hence coincide with the decryption of persistent memory.

Freed Memory. Since we zeroed out freed pages, there is no data remaining.

Kernel Memory. This part is not encrypted. However, we removed the FDE key, other possible key material, and remnants of our cipher operations.

In sum, the attacker not only has to possess the device for long time, but also invest remarkable effort to retrieve confidential data as the attacker is assumed to be unable to break cryptographic primitives.

5.3 A Main Memory Encryption Architecture for Containers

In this section, we present Freeze & Crypt (F&C) as Contribution 7, a flexible main memory encryption architecture for OS kernels. F&C adds efficient and transparent process memory en- and decryption on the granularity of process groups to OS kernels. We use F&C to protect unattended or stolen mobile devices from memory attacks. We design F&C at the example of the Linux kernel to comply with multiple platforms, kernel versions and incur only minimal changes sustaining the kernel's common operability. We integrate F&C into the kernel building upon the existing cgroups freezer subsystem. The cgroups mechanism, already introduced in Section 3.2, itself allows for the dynamic formation of groups of processes, to which we refer to as cgroups in the following. The cgroups freezer subsystem, referred to as freezer in the following, allows for freezing and thawing of cgroups, i.e., for their suspension and resumption.

When freezing a cgroup, the kernel sends a signal to all processes in the cgroup. Each of those processes reacts to that signal by entering the so-called refrigerator in kernel space. The refrigerator is a loop function which ensures that processes neither execute in user space nor react to external events. When frozen, the user space part of a process has no means to access or alter its memory space. The refrigerator is thus a proper place to temporarily alter the processes' memory without side effects occurring on its user space part. In particular, we let the processes en- and decrypt the segments of their memory space, for example, the heap, stack, code or anonymous segments, on their own and in parallel. This makes our approach especially efficient on multi-core systems. Upon thawing a cgroup, the processes decrypt their memory before we allow them to leave. For the parallelization, F&C synchronizes the processes and threads inside the refrigerator, as they operate on shared resources like physical memory pages or kernel structures. The key we use for en- and decryption of a cgroup is present only during freezing and thawing and may change on each freeze. While our design keeps F&C independent of a hardware platform, it can also be implemented on other OSs supporting process suspension and process groups.

As Contribution 8, we demonstrate the utility of our F&C by implementing a prototype and combining it with the secure virtualization architecture from Chapter 3. We use F&C to amend the protection of data confidentiality from local and remote attackers with the protection of suspended containers from physical attackers. The secure virtualization architecture is especially suitable for F&C as it enables secure encryption key management with an SE preventing brute-force attacks on encryption keys. We wrap the encryption keys using the SE during the time containers are encrypted, allowing only legitimate users in knowledge of the SE's passphrase to decrypt containers. We conduct a thorough security and performance evaluation to demonstrate the practical usability on mobile devices.

This section is organized as follows. In Section 5.3.1, we first present the design of F&C. Next, we elaborate on the implementation of our prototype for the Linux kernel in Section 5.3.2. We present an overview of the combination of F&C with the virtualization architecture in Section 5.3.3. Our performance evaluation and security discussion can be found in Section 5.3.4 and Section 5.3.5, respectively.

5.3.1 Memory Protection Concept

We first present an overview on F&C's overall design and the involved OS components in Section 5.3.1.1. Subsequently, we elaborate the synchronization of the processes during main memory encryption in Section 5.3.1.2. In the following, the term encryption also applies for decryption and we differentiate only where decryption differs from encryption.

5.3.1.1 Design of Freeze & Crypt

Based on the Linux kernel's cgroups freezer, F&C allows for the dynamic creation of cgroups containing the processes and threads whose memory space we encrypt. From here on, we subsume processes and threads under the term *task* and only differentiate when relevant. For every cgroup, we use an independent, ephemeral key for each encryption pass. We categorize the memory space of processes into the following segments:

Text. The executable, read-only code of a task.

Data. Writable, as well as read-only data belonging to the task's code. Contains statically-allocated variables initialized at compile-time.

BSS. Zero-initialized, writable data. Contains the at compile-time uninitialized statically-allocated variables.

Heap. A task's dynamically allocated memory.

Stack. Local variables of functions and their parameters, etc.

Anonymous Mappings. Large, dynamic memory allocations a task can map into its process space and share with others.

File Mappings. All parts of memory-mapped regions. For example, large writable files, but also read-only files, such as code of shared libraries. We further differentiate between regular and non-regular mappings, as well as between their access permissions. Non-regular mappings are, for example, memory-mapped devices, DMA or IPC resources.

Special Mappings. The *timers*, *vectors*, *vdso* and *vsyscall* segments. They contain no confidential user data. The kernel manages these segments as system-wide shared kernel resources providing them to tasks as kernel interfaces.

Except for special mappings, a task can load or store confidential information in all other segments. F&C thus allows to select all other segments for encryption except the special mappings. Depending on the use case, fewer segments can be critical and thus omitted from encryption. We outline the critical segments for the protection of memory on devices running the secure virtualization architecture in Section 5.3.3.

States of a Cgroup

Figure 5.4 depicts the four different states of a cgroup along with its tasks. On the system, we have an arbitrary number of cgroups in any of the states, as well as tasks not assigned to a cgroup, which can potentially become part of a freezer cgroup or form a new cgroup. This allows us to create disjoint cgroups, to freeze or thaw multiple cgroups at the same time, and to assign different encryption keys.

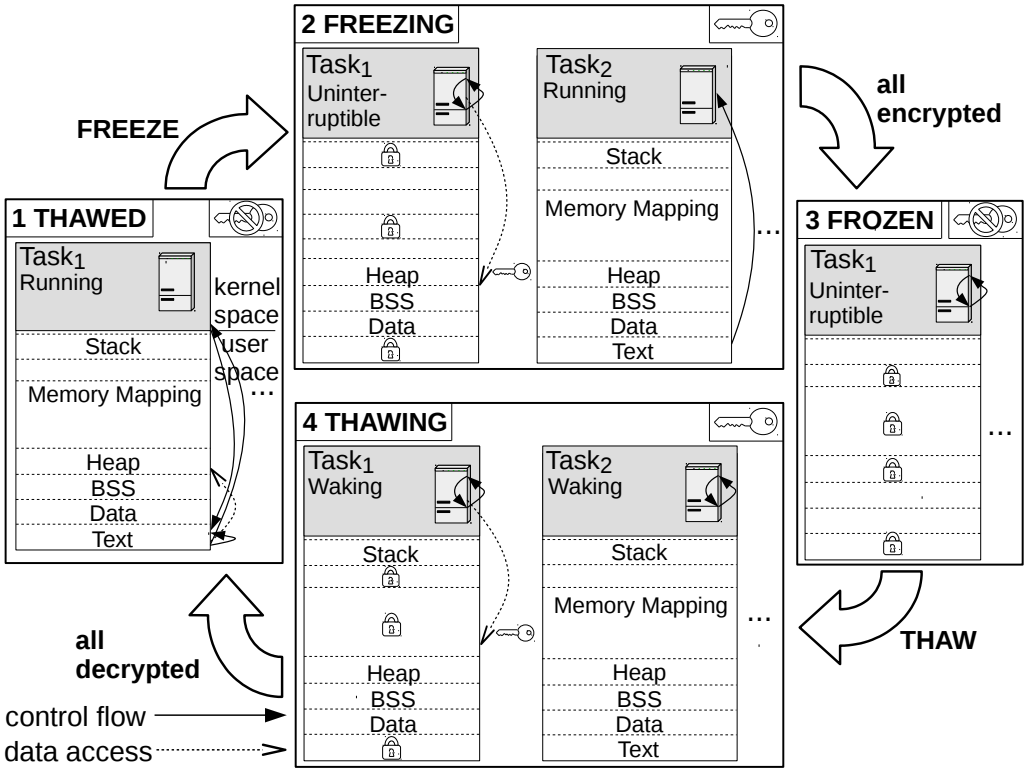


Figure 5.4: The four states of the freezer along with the behavior of tasks in the different states.

Thawed. In the default state, multiple tasks are *running*, such as $Task_1$ in Figure 5.4, and the cgroup has no encryption key assigned. The cgroups mechanism ensures the inclusion of future child tasks to the cgroup by default. The arrows demonstrate that tasks execute their code in user space, access their memory segments and possibly jump between kernel and user space when making system calls.

Freezing. After starting the freeze, the cgroup enters the *freezing* state and has an arbitrary encryption key assigned. Tasks in the refrigerator go into an *uninterruptible* state and dwell inside until the cgroup is thawed. $Task_1$ already entered the refrigerator and is currently encrypting its memory segments. The arrow for the running $Task_2$ shows that it is just jumping from its execution in user space to kernel space into the refrigerator. In the refrigerator, the user space part of a process has no means to access or alter its memory. The tasks are unable to react to incoming IPC or external events, such as kill signals. The kernel buffers such data and events, enabling tasks to process them after thawing.

Frozen. After finishing the encryption, the cgroup is in the *frozen* state and has purged its encryption key. Its tasks are all in the *uninterruptible* state, stuck in the refrigerator in kernel space off the run-queue. Once scheduled, the kernel immediately calls the

scheduler again to switch to another task. All tasks have encrypted the previously specified memory segments.

Thawing. After starting the thaw, the cgroup is in the state *thawing* and decrypts with the same key as used for freezing. The cgroup brings its tasks into a *waking* state. When scheduled, the tasks simultaneously decrypt their segments before leaving the refrigerator. $Task_1$ in Figure 5.4 is about to decrypt its segments, while $Task_2$ has already finished. We retain $Task_2$ in the refrigerator for synchronization until all tasks of its cgroup finish decryption. This causes tasks depending on each other to be released simultaneously.

Extension of the Freezer

We extend the freezer's initialization, freezing and thawing functionalities to manage the protection of cgroups. Figure 5.5 illustrates these functionalities in the form of a sequence diagram. Bold elements represent the new functionality we introduce to the freezer.

Initialization of F&C. Privileged processes in user space can manage the cgroups functionality via an interface to the kernel, which allows for creating cgroups and adding or removing tasks. To initialize a freezer cgroup, a privileged process adds tasks to the cgroup, here $Task_X$ and $Task_Y$. We extend the interface with the option to associate the freezer with an arbitrary key and the list of memory segments to be encrypted. The concept intentionally leaves the key management open to the specific use case F&C gets applied. A privileged entity in user space must ensure to provide F&C the keys for encryption and to securely store the keys for encrypted cgroups. The key can, for example, originate from password derivation, a TPM or an SE, as described for the secure virtualization architecture in Section 5.3.3.

Freezing Procedure. As depicted in Figure 5.5, a privileged process starts the freeze and thaw of cgroups. The freezer changes its state and signals its tasks to enter the refrigerator. Every task notifies the freezer after finishing its encryption. When the freezer has received all encryption notifications, the cgroup goes into the state *frozen* and the freezer purges the encryption key. To erase potentially remaining sensitive remnants, the freezer also zeroes out pages freed by running or terminated processes, as well as the used cipher kernel structures and the relevant kernel stack range in memory after the encryption. Finally, the freezer notifies user space about the finished encryption of the cgroup. The standard freezer only toggles its state when user space requests the actual state from the freezer. In F&C, the subsystem actively manages its state to be able to purge the key and other remnants as early as possible. We also actively notify user space, for example, the entities managing the cgroups encryption, about the change of state. We show the usefulness of this feature in the example of the secure virtualization architecture.

Thawing Procedure. Before triggering the thawing of a cgroup and therefore starting the decryption, a privileged process passes the corresponding key to the freezer. The freezer wakes every included task. When scheduled, this normally causes a task to leave the refrigerator. With F&C, the tasks first decrypt their previously

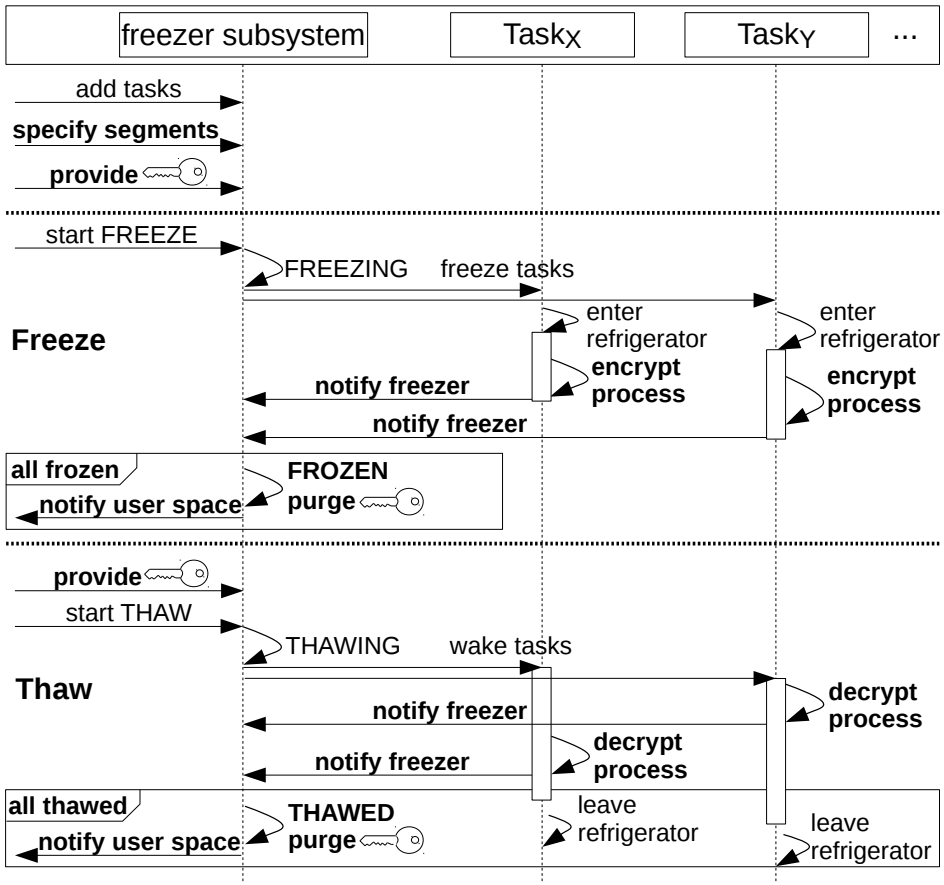


Figure 5.5: Extensions (bold) to the initialization, freezing and thawing procedures of a cgroup with its tasks for Freeze & Crypt.

encrypted memory segments and notify the freezer. Unlike the original freezer, we do not immediately switch to the state *thawed*, but wait until the last task finishes decryption and notifies the freezer before switching states. Then, the tasks safely leave the refrigerator when scheduled and resume their execution. The freezer purges the decryption key, other remnants and notifies user space. The key and traces of it are hence only present in the freezer during en- and decryption.

5.3.1.2 Synchronization of Tasks

Since the tasks of a freezing cgroup enter the refrigerator in parallel and share resources, we synchronize their concurrent encryption. The synchronization of shared resources is important, because some tasks may not yet be suspended or not be part of the encrypting cgroup. In these cases, we ensure that shared resources are only encrypted when all resource-sharing tasks enter the refrigerator. The kernel's memory management is responsible for resource sharing between tasks. Shared resources can be ASs and physical pages, as shown in Figure 5.6. ASs describe the different memory segments a task has, for instance, the

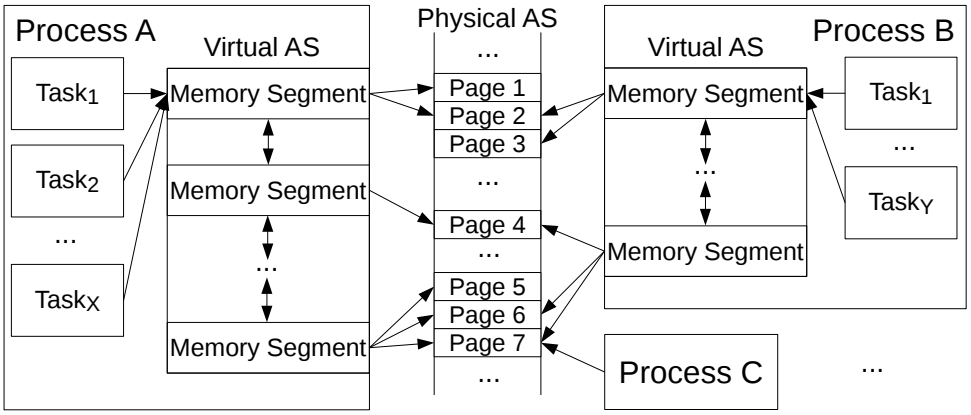


Figure 5.6: Relationship of processes and threads, their memory mappings and the sharing of resources.

heap, stack, or the code segment. Each process has its own virtual AS, presenting the virtual memory layout of the segments. Memory pages in the AS point to physical pages in main memory when a mapping exists. Physical pages are shared when mappings from multiple ASs point to the same physical page. In the following, we first focus on the shared ASs before addressing shared physical pages.

Shared Address Spaces

A task can spawn threads and fork new processes. The kernel assigns a forked process its own AS. When spawning, the parent process shares its AS with the spawned thread. This is illustrated by Figure 5.6 where the tasks of process A and B share their ASs. Our goal is to ensure that every AS is encrypted by exactly one task to avoid multiple encryptions of the same AS. Without synchronization, sharing tasks would encrypt the same AS repeatedly when entering the refrigerator, and tasks might be accessing memory that is already being encrypted.

Figure 5.7 focuses on the chronological sequence of tasks entering the refrigerator with processes A and B belonging to the same cgroup. Upon freezing, the tasks enter the refrigerator at an undefined point in time in an order determined by the scheduler. The first task to enter the refrigerator, here $Task_2$ of process A and $Task_1$ of process B, predetermines the segments to be encrypted. The last entering tasks, $Task_X$ and $Task_Y$ respectively, encrypt the previously identified memory segments. Only the last task can safely encrypt the segments, because at that time all other tasks sharing the AS have entered the refrigerator. At that point in time, the user space parts of the tasks are inactive and thus not accessing, allocating or freeing memory.

We let the first entering task identify the segments to be encrypted, because an AS can have a large number of segments. While some of them may be special segments or segments that were not selected for encryption, the identification of segments creates a list of those segments to be encrypted. This allows the last arriving task to quickly start encryption based on the list. We ensure that encryption does not start unless all segments

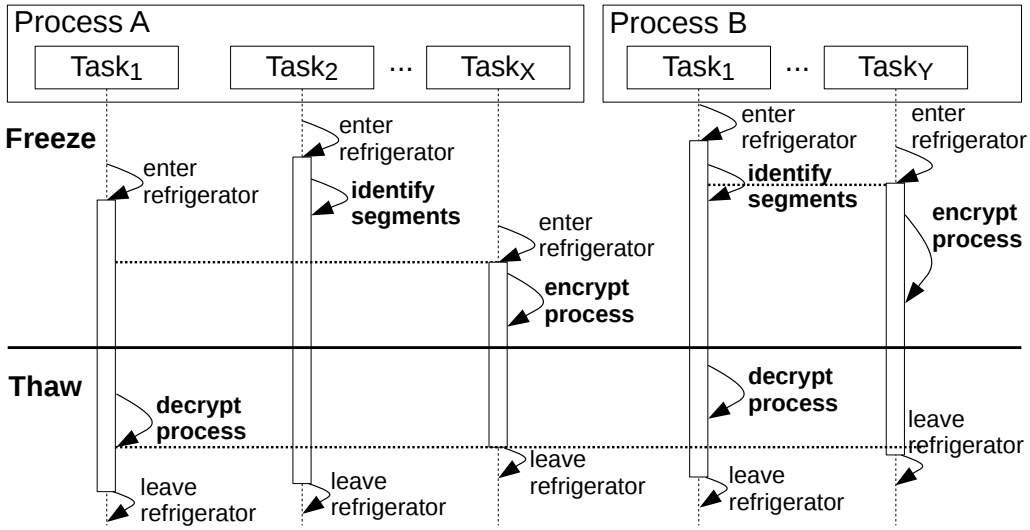


Figure 5.7: Synchronization of tasks entering and leaving the refrigerator during freeze and thaw.

are identified. As depicted in Figure 5.7, $Task_Y$ of process B encrypts its AS in parallel to $Task_X$. Both tasks start encryption after all other tasks sharing their AS entered the refrigerator. In reverse, only the first scheduled, waked task of a process decrypts the segments. Once decryption of all process spaces is completely finished, we allow all tasks to leave the refrigerator. After that, tasks get back to user space operation and may access their ASs and allocate or free memory.

Shared Physical Pages

The tasks use virtual addresses during their encryption. However, the physical pages effectively encrypted can be shared between process boundaries complicating the encryption process. Hence, we have to determine whether the page to be encrypted is shared with other processes before encrypting it in order to prevent the corruption of other ASs. Shared pages are contained in more than one AS and can be further categorized. First, these can simply be pages intended to be read-only for all tasks. Second, these can be writable Copy-On-Write (COW) pages, which are shared between processes for the time they only read the page. As soon as a sharing process writes the page, COW allocates a new page to ensure that the task writes on a separate copy of the page. Third, shared pages can represent shared memory for IPC, where distinct processes work on the same set of physical pages. Fourth, pages could have been merged via the kernel's Kernel Samepage Merging (KSM) mechanism, which searches identical pages in process space and merges those pages to one shared page to save memory.

Our goal is to ensure that pages are encrypted only once and to exclude pages from encryption shared beyond the boundaries of the cgroup. Before a task encrypts a shared page, we thus ensure that the page is not referenced by other tasks possibly not part of the cgroup or not yet in the refrigerator. F&C ensures encrypting these shared pages only

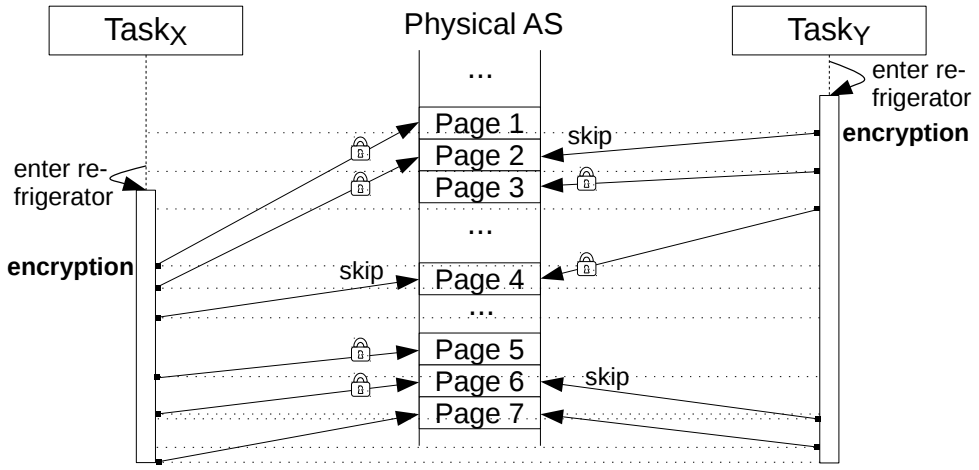


Figure 5.8: Example for the synchronization of tasks during physical page encryption.

once. We also ensure that page encryption does not trigger COW on shared pages. This would cause a task to encrypt a copy of the original physical page and the sharing tasks to later encrypt the original page again possible causing page replication. When thawing, an encrypted page can always be decrypted as long as no other task is currently decrypting it.

Figure 5.8 provides an example for the encryption procedure showing the two encrypting $Task_X$ and $Task_Y$ from Figure 5.6 and 5.7 with the same physical AS layout. $Task_Y$ enters the refrigerator before $Task_X$ and first considers encrypting page 2, but skips. The page is shared with process A only, but $Task_X$, part of process A, is not yet inside the refrigerator. Then, $Task_Y$ encrypts page 3, because process B is its only user. Next, $Task_Y$ encrypts page 4, because the page is shared with process A only and in the meantime all tasks of process A entered the refrigerator. Afterwards, $Task_X$ starts the encryption of its AS beginning with page 1, which process A exclusively uses. In the next step, $Task_X$ considers page 2 for encryption and encrypts it, because $Task_Y$ previously skipped the encryption. Then, $Task_X$ considers encrypting the shared page 4, but realizes that it is already encrypted. Still scheduled, $Task_X$ encrypts its exclusively used page 5 and continues with page 6. At the time of encryption, page 6 is shared with process B, but not yet encrypted. Shortly after that, $Task_Y$ gets scheduled and skips page 6, because it is currently being encrypted by $Task_X$. Finally, $Task_X$ and $Task_Y$ attempt to encrypt page 7. According to Figure 5.6, process C is also using that page. Depending on whether process C is part of the cgroup and already in the refrigerator, either one of the two tasks encrypts the page or both skip it.

5.3.2 Implementation

In this section, we describe the implementation of our prototype for the Linux kernel on ARM and x86, tested with 3.4, 3.10 and recent 4.x kernel versions. We start with the management of F&C from user space. Afterwards, we focus on the extensions we make to memory management and on the synchronization of tasks during freezing and thawing.

Then, we elaborate the procedure used by a task to encrypt its memory space.

5.3.2.1 Interaction with User Space

In order to pass the key and the list of segments from user space to the kernel, we create additional files in the cgroups virtual filesystem. We read the key and list of segments we receive from user space via file handles similar to the existing freezer state change functionality. However, getting notified in user space about changes in filesystems (via inotify) does not work with the cgroups virtual filesystem's pseudo files. Instead, user space must periodically read the state file to check whether the freezer changed state. To avoid this polling, we use the *eventfd* and the cgroups notification API to explicitly notify user space when freezing and thawing terminates.

A limitation of F&C is that our current prototype allows passing arbitrary keys from user space only for disjoint freezer cgroups. In this case, disjoint means that the cgroups share no pages with each other. An example are, for instance, the containers of the virtualization architecture, see Section 5.3.3. Using the same key for non-disjoint cgroups ensures correct decryption independent of the encryption order in contrast to using different keys. For the example of two cgroups A and B sharing pages with each other, the first freezing cgroup A would encrypt its memory with its own key, but leave shared pages unencrypted, because the tasks of the other cgroup may still use the page. When cgroup B encrypts its memory space, it would eventually encrypt the shared physical pages with its own key when cgroup A is still frozen. If both keys are identical, both cgroup A and B can decrypt in any order. When using different keys, the user space entity managing the cgroups must ensure to thaw reversely to the order of encryption. Otherwise, cgroup A would decrypt shared pages with its own key. A possible solution for this would be to use introduce a shared key for shared pages. Another direction can be to use commutative encryption algorithms to relax this restriction. This would allow to shuffle the order of decryption while using different keys. The restriction of this approach is that despite the order of decryption becomes irrelevant, we still have to keep cgroups sharing pages in the refrigerator until the depending cgroups are thawed in order to complete the decryption.

User space must also avoid the thawing and freezing of non-disjoint cgroups at the same time, because encrypting tasks could potentially clash with decrypting tasks when sharing physical pages. These limitations have to be taken into account when triggering process group encryption from user space. Containerization where the sharing of physical pages is avoided ensures to create disjoint cgroups.

5.3.2.2 Memory Management

A process and its spawned threads share the kernel structures which describe a task's memory layout. Figure 5.9 shows relevant components the Linux kernel uses for memory management and how the tasks share resources. The illustration outlines the sharing of resources at the example of processes A and B. Each task has its own process descriptor for process management which contains, for example, the name and PID. We temporarily modify the light gray colored components during encryption, like VMAs and Page Table Entries (PTEs). Medium-gray colors denote components to which we introduce new functionality for handling shared resources.

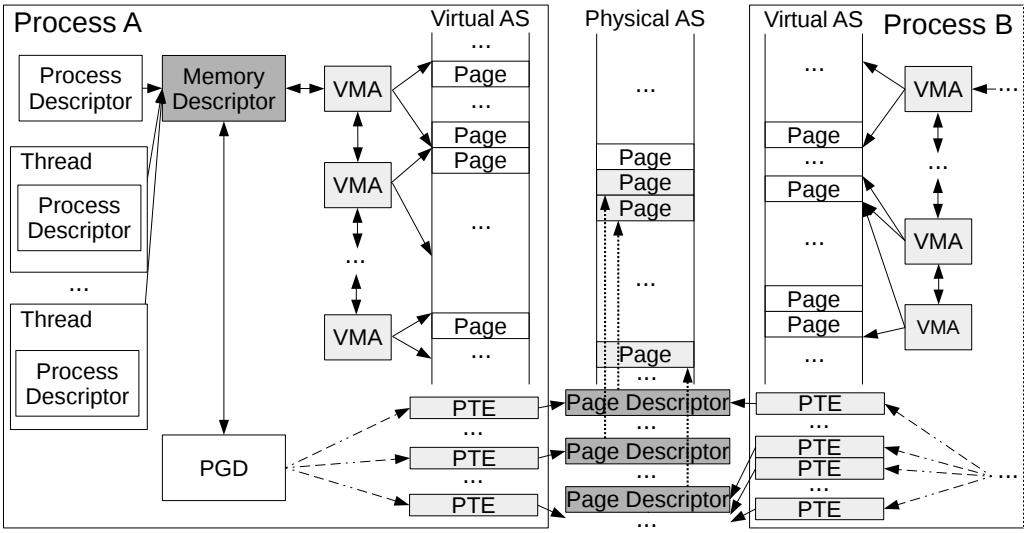


Figure 5.9: Linux memory management data structures and our modifications highlighted in gray colors.

The process descriptor points to a memory descriptor, which describes the virtual memory layout of a process. The memory descriptor tracks the number of users it has, i.e., the number of threads that share the memory descriptor with the parent process. We extend the memory descriptor to additionally count the number of frozen sharing tasks which are tasks that have already entered the refrigerator. This way, the tasks are able to determine whether they are the first or last one to enter the refrigerator. The memory the descriptor points to is reflected by a linked list of VMAs representing the memory segments we selectively encrypt. For the encryption, we make non-writable VMAs temporarily writable.

The memory descriptor keeps a reference to a Page Global Directory (PGD) used for translating a process' virtual addresses to corresponding physical addresses. Based on a page walk from the PGD traversing the Page Upper Directory (PUD) and Page Middle Directory (PMD), we determine the PTE corresponding to a virtual address if mapped, i.e., if the page exists in main memory. A PTE has several values tracking the page's state, for instance, if the page is shared between processes, or if the process has accessed or written the page. By writing a physical page, the kernel also possibly modifies the PTE's values. Before the encryption of a physical page, we thus save the PTE's values and restore them right after the page's encryption. This especially prevents file-backed pages accidentally being made persistent through the page cache by not setting the dirty flag.

A present PTE maps one physical page by referencing a so-called page descriptor, which describes one specific physical page in memory and which directly points to the page's physical address. We set a flag in the page descriptor indicating a lock on physical pages in memory during encryption. By locking the page, we make sure that the locking task obtains exclusive page access. Furthermore, we extend the page descriptor with the functionality to mark a page as encrypted. With this functionality and page locking, we ensure that tasks encrypt pages only once. To keep track of the entities referencing the page, the page

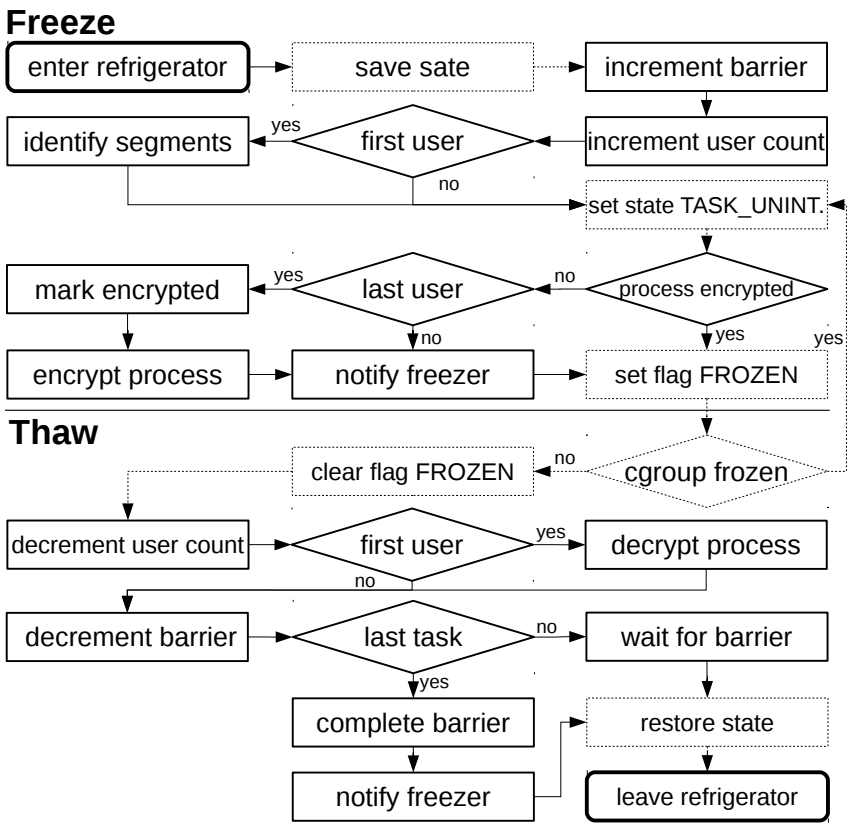


Figure 5.10: Extended procedure for the synchronization of tasks entering and leaving the refrigerator.

descriptor holds a reverse map to referencing VMAs. Since VMAs have a back reference to their memory descriptor, we are aware of all tasks, possibly outside the cgroup, referencing the page described. This allows us to determine whether the shared pages a task references may be encrypted or not.

5.3.2.3 Synchronization of Tasks

Figure 5.10 depicts a sequence diagram, which describes how a task runs through the refrigerator. We highlight the new functionality with continuously lined boxes, while the standard freezer functionality is represented in dotted boxes. A task entering the refrigerator first saves its current task state. We then increment a cgroup-wide barrier we newly introduce, which counts the tasks entering the refrigerator. When thawing, the barrier forces thawing tasks to wait in the refrigerator until decryption of the whole cgroup completes. In the next step, we increment the frozen user count of the task's memory descriptor. The first user, i.e., the first task, identifies the memory segments in the process space, which were selected for encryption. After possibly identifying the memory segments, a task goes as usual to the state `TASK_UNINTERRUPTIBLE`. If the process is not yet encrypted, the task checks if it is the last entering user of the memory descriptor. If yes, the task

marks the memory descriptor as encrypted and encrypts the identified memory segments. After that, or in cases where the task was not the last user of the memory descriptor, the task notifies the freezer about being frozen. Then, the common freezer procedure executes. The task flags itself `FROZEN` and ends up in a loop, checking if its cgroup is frozen or not. When scheduled, the frozen task executes the common functionality and indicates the scheduler to switch to another task.

When thawing a cgroup, the freezer wakes the cgroup's tasks, leaves the frozen state, and each task clears its `FROZEN` flag. The tasks normally leave the refrigerator and restore their previous process state. However, F&C handles the decryption of the tasks before leaving. For this purpose, thawing tasks first decrement the frozen user counter. This ensures that only the first thawing task using the memory descriptor decrypts the associated memory space. Otherwise, the task skips decryption. In the next step, every task decrements the cgroup-wide barrier. The last task about to leave the refrigerator completes the barrier. All other tasks waiting at the barrier are finally free to leave the refrigerator. The last task also notifies the freezer about the terminated decryption of the whole cgroup.

5.3.2.4 Process Space Encryption

For the encryption in main memory with the kernel crypto API, we apply the asynchronous bit-sliced AES CTR implementation using NEON instructions with a 256 bit key size. The CTR mode achieves especially high performance on multi-core systems through its parallelization. We use the physical page addresses as IVs, resulting in distinct IVs for each encrypted block. During the encryption, a task iterates over the previously identified VMAs of its memory descriptor. The task first checks the VMA's write permissions. If a VMA is not writable, the task makes the VMA writable. The task then encrypts the VMA page by page. After encrypting the whole VMA, the task restores the VMA's write permissions, if necessary. A task's encryption terminates after encrypting the last VMA.

Figure 5.11 presents the scheme of the encryption procedure on page level. The page level encryption procedure starts by checking if the virtual page to be encrypted is physically present in main memory. We skip non-present pages. As swapped pages are also flagged non-present, we do not encrypt swapped pages. For the encryption of swap, we refer to standard Linux swap encryption. On a present page, the task tries to acquire our lock. Failing to do so indicates that the page is currently being encrypted by another task. Hence, the task skips the encryption of this page.

The next step in Figure 5.11 differs depending on whether we are en- or decrypting the cgroup. In the decryption case, we only need to check whether the page was already decrypted. If so, the page simply gets unlocked, otherwise we mark it as decrypted and start its decryption right away. This step is simplified in the illustration, which omits the check and possible page unlocking.

In the encryption case, we are obliged to make sure the page is ready for encryption. This means to first check if the page descriptor was already marked as encrypted. If this is the case, the task releases the lock and skips the page. If the page was not already encrypted, the task then checks if the page descriptor is referenced more than once. If there is only one PTE referring to that page, the task immediately considers the page ready for encryption. Multiple references indicate that the page is shared and may not be ready. In

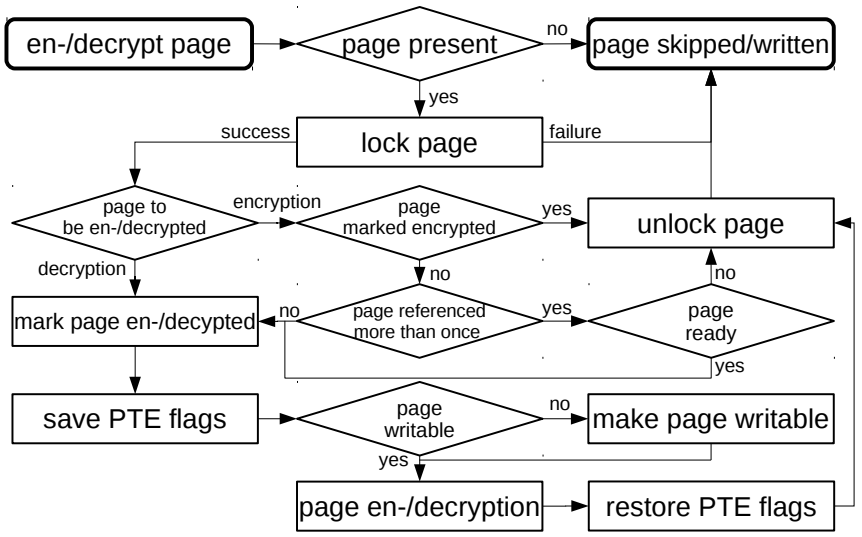


Figure 5.11: Schematic procedure for the encryption and decryption of physical pages.

this case, the task must specifically check the page's readiness for encryption, because it is shared across AS boundaries. Considering non-ready pages for encryption would corrupt the AS of other processes. The task thus releases the lock on a non-ready page and skips it. Another task of the cgroup considering that page for encryption will ensure the page's encryption later if the page is only shared with processes within the same cgroup.

After marking a page as en-/decrypted, the procedure continues equally both for en-/decryption, as shown in Figure 5.11. In case the page is ready, the task marks it as encrypted. Before writing the page, the task stores the PTE's flags. These flags indicate whether the page is, for example, writable, dirty, or young. The task checks whether the page is writable or not, because writing a read-only page causes COW in the triggered page fault. This would cause a replication of that page and the encryption of the copy. Since the task made sure the page is only used within the cgroup, it makes the PTE writable before encryption to circumvent the page fault. In the next step, the task encrypts the physical page and subsequently restores the PTE's flags. This ensures that PTE flags remain unaltered. The task finally unlocks the page.

5.3.3 Combination with the Secure Virtualization Architecture

We combine F&C with the secure virtualization architecture from Chapter 3 to establish a platform which protects confidential data on mobile devices both against physical and remote attackers. With the combination, we also demonstrate the capability of F&C on smartphones in practical use. We use Nexus 5 smartphones running multiple Android containers based on the virtualization architecture. When a mobile device with the virtualization architecture is actively used, one Android container is in foreground, while one or more containers run in background. We encrypt background containers and only leave actively used containers unencrypted. After the device is idle for a certain period, we ensure to encrypt all containers in order to protect all sensitive data from memory

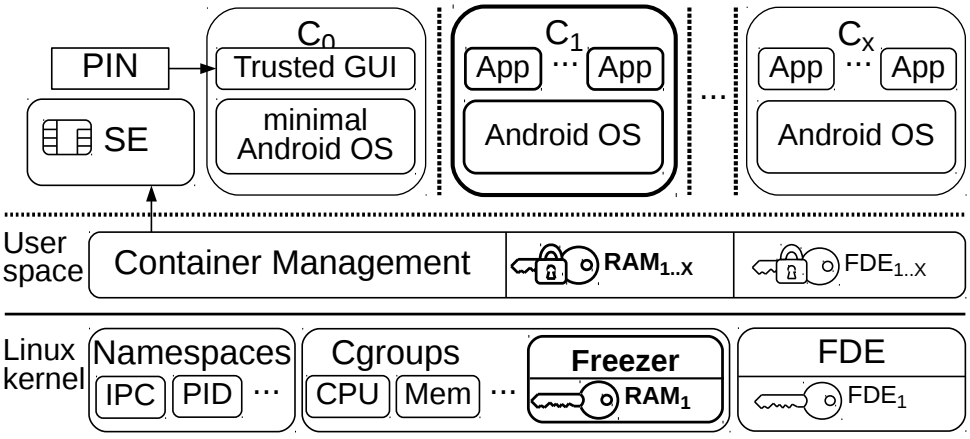


Figure 5.12: Most relevant components of the secure virtualization architecture for mobile devices along with our extensions.

attacks. We first describe our extensions to the virtualization architecture to leverage F&C in Section 5.3.3.1. Then, we focus on the key management and describe how we employ F&C to protect the containers' sensitive data in memory in Section 5.3.3.2. Section 5.3.3.3 details how we handle background events for frozen containers, such as incoming phone calls or alarms.

5.3.3.1 Extension of the Virtualization Architecture

In this part, we describe Contribution 8, the integration of F&C into the secure virtualization architecture for mobile devices. Figure 5.12 shows the components of the secure virtualization architecture that are most relevant for the extension with F&C. We refer to Chapter 3 for more details on the virtualization architecture. The illustration depicts a scenario with running containers $C_{0..n}$ on top of the trusted components in user and kernel space. The $C_{1..n}$ are the isolated Android user containers with apps and possibly sensitive data, for example, a private and a corporate container. $C_{1..n}$ store their persistent data in images protected with kernel-based FDE. Therefore, the CM stores wrapped, persistent disk encryption keys for $C_{1..n}$. We denote these keys with $KFDE_{1..n}$ for $C_{1..n}$. These keys can only be unwrapped using the SE, which the CM communicates with. For F&C, we assume that incoming sensitive data for $C_{1..n}$ is encrypted and that encryption terminates inside $C_{1..n}$. We need to encrypt those $C_{1..n}$, which process sensitive data, such as corporate secrets, and can leave others unencrypted. We do not encrypt C_0's RAM contents, because it contains no sensitive user data. We extend the CM with functionality to configure F&C and to generate, wrap and unwrap the RAM encryption keys, denoted by $KRAM_{1..n}$ for $C_{1..n}$, using the SE similar to the wrapping of $KFDE_{1..n}$. $KFDE_{1..n}$ are unwrapped after the start of $C_{1..n}$, part of the CM and thus not covered by $C_{1..n}$'s encryption. These keys thus need to be as well protected from attacks. We remove unwrapped $KFDE_{1..n}$ at the moment we encrypt the previously started $C_{1..n}$.

The bold-faced components in Figure 5.12 depict the scenario where C_1 is in foreground,

while the other containers are in background. The illustration shows the freezer using unwrapped $KRAM_1$, which indicates that the system has locked C_1, is about to switch to C_0 and to encrypt C_1 in background. Meanwhile, the FDE module solely keeps unwrapped $KFDE_1$. The kernel does not require $KFDE_{2..n}$ to be unwrapped, because $C_{2..n}$ are frozen and cannot access their filesystem. After freezing C_1 completes, the CM also removes C_1's unwrapped $KFDE_1$ from the kernel.

Like on common smartphones, we require users to re-authenticate to start or resume a suspended, encrypted container. The virtualization architecture suits F&C, because it builds on an SE with two-factor authentication for secure cryptographic key management, for instance, a smartcard via NFC or a microSD. This allows us to securely wrap encryption keys with the SE. While using an SE prevents brute force attacks on the encryption keys, it can weaken usability when re-authenticating. However, the usage of an SE was the preferred choice due to possibly high security requirements in production scenarios. For other use cases, our system can easily be adapted to a less secure, but more usable scheme, such as PIN or password-based key derivations or swipe patterns.

5.3.3.2 Container Encryption and Key Management

For the encryption of $C_{1..n}$, we consider all selectable memory segments described in Section 5.3.1.1 relevant for the virtualization architecture except the following:

Read-only executable file-mapped segments. This is shared library code, which does not contain sensitive user data in our system, but solely shared constant data. Since each container features a full userland, $C_{1..n}$ include separate libraries, for example, libc. The data segments of libraries do not fall in this category and are hence encrypted.

Writable non-regular file-mapped segments. These segments constitute memory areas of memory-mapped devices, IPC resources, or DMA files where drivers share memory with hardware devices. Encrypting such memory can corrupt the memory space on most platforms, for example, because hardware devices are not aware of the encryption.

All other segments must be protected, because processes might store or load sensitive data there. The diagram in Figure 5.13 describes the procedure for protecting $C_{1..n}$'s sensitive data with F&C. The illustration shows the crucial steps of starting, encrypting and decrypting C_i.

Start C_i. To start C_i, we enter the PIN of the present SE in the Trusted GUI of foreground C_0. The GUI passes the information to start C_i with the PIN to the CM. The CM generates a fresh random $KRAM_i$ using the kernel random number generator seeded with hardware entropy based on hardware random numbers. The CM unlocks the SE and uses it to wrap the generated $KRAM_i$. At this point, the CM holds both the wrapped and unwrapped $KRAM_i$. The unwrapped $KRAM_i$ is stored to be able to immediately encrypt C_i without user interaction and without the SE when locking or switching back to C_0. Reversely, the wrapped $KRAM_i$ is stored for the next decryption pass of C_i using the SE. Next, the CM starts C_i in foreground, which brings C_0 to background. Therefore, the CM creates C_i's namespaces, configures its

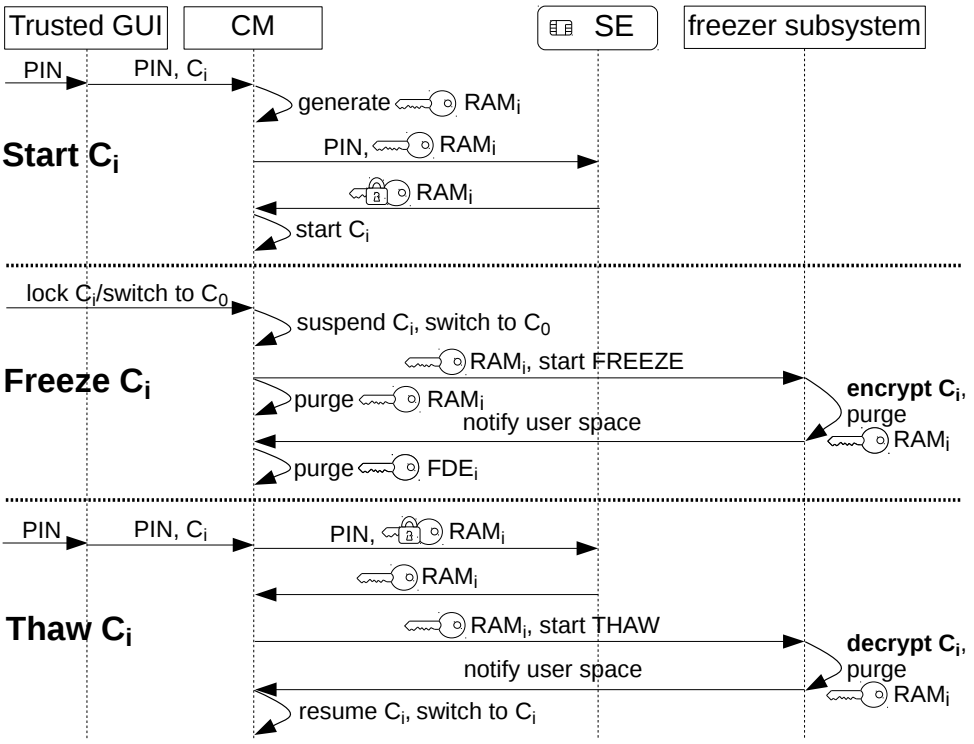


Figure 5.13: Chronological sequence for starting, freezing and thawing containers along with involved key management operations.

cgroups and mounts its images. The CM also specifies the segments to be encrypted. Starting C_i includes unwrapping C_i's $KDFE_i$ using the SE to provide the key to the FDE module for accessing the encrypted data image. This aspect is omitted in Figure 5.13.

Freeze C_i. We freeze and hence encrypt C_i's RAM when the user actively switches back to C_0, or when the user or the system locks C_i, causing a switch back to C_0. According to Figure 5.13, the CM then suspends C_i and immediately switches to C_0. The CM provides the freezer with the stored, unwrapped $KRAM_i$ and starts C_i's freeze. This triggers F&C, which encrypts C_i. In the meantime, the CM purges its stored, unwrapped $KRAM_i$ from user space. After the encryption, the freezer purges its utilized $KRAM_i$ in the kernel and notifies the CM. The CM then purges the no longer required, unwrapped $KFDE_i$ in user and kernel space. We destroy the encryption keys by overwriting them in RAM and by flushing the corresponding caches. After that, the CM stores only the wrapped $KRAM_i$ and $KFDE_i$, leaving no trace of C_i's volatile and persistent data. All confidential assets in kernel memory are deleted and the non-encrypted segments contain no confidential information. In other use cases, the entity managing the encryption can purge further assets in kernel at this point if present, for example, the credentials of IPSec, which is a kernel level VPN mode.

Thaw C_i. Only to start and to thaw C_i, i.e., when decrypting and putting C_i to foreground, the user has to provide the PIN of the SE to the Trusted GUI. The CM unwraps the wrapped $KRAM_i$ using the SE and provides the freezer with the unwrapped $KRAM_i$. Afterwards, the CM triggers the thaw. The freezer decrypts C_i, purges its utilized key in the kernel and notifies the CM. Then, the CM resumes C_i, switching it to foreground. To simplify matters, Figure 5.13 omits that the CM also unwraps the wrapped $KFDE_i$ and provides it to the FDE module before thawing C_i. The CM keeps its user space copy of the unwrapped $KRAM_i$ for the next freeze. At this point, the CM could also generate a new unwrapped $KRAM_i$ with a wrapped counterpart for the next freeze. This would prevent replay attacks swapping old portions of encrypted RAM. However, such scenarios are not part of our attacker model from Chapter 2 and we consider this threat negligible.

5.3.3.3 Background Activities

In this part, we describe how background activities can be handled for encrypted, suspended containers. When a device running the virtualization architecture is suspended, the main application processor sleeps like on regular smartphones. For background activities, devices sustain their connection to external sources via hardware components and interrupt controllers. In case of an event, for example, causing a notification in the Android OS, the main application processor and hence the kernel and processes get woken. With the virtualization architecture, incoming events for encrypted containers are, similarly to common mobile devices, processed by our underlying kernel. Due to our virtualization architecture's hardware device virtualization infrastructure, the kernel forwards all incoming events to the virtualization components. This enables the components to handle background activities for frozen C_i even though C_i is inactive and encrypted until the next unlock. The architecture virtualizes hardware devices either in user space or directly in the kernel. By extending these virtualization components, we have the possibility to buffer incoming events and to notify the user in the non-encrypted C_0.

User-Space Virtualized Devices

For this class of hardware devices, multiplexers in C_0 virtualize device functionality for all containers. Multiplexers virtualize those devices in user space for which a part of the driver functionality is implemented in user space, for instance, the radio interface as part of the RIL, or Wi-Fi functionality. Our virtualization architecture thus receives incoming short messages and phone calls for possibly frozen C_i via the radio interface in the kernel and hands them over to the multiplexers in C_0. This allows us to extend our architecture by adding functionality to the multiplexers to notify the user of events for frozen C_i inside C_0. For this purpose, we can use regular Android notification intents in the multiplexers to raise a notification in C_0's GUI application. The multiplexers must buffer C_i's data until C_i thaws and processes the data, for instance, by reading on a socket terminating in C_0. We verified the functionality of our extension with various incoming user-space virtualized data sources. For phone calls and short messages, users were notified in C_0 and able to take incoming calls or receive short messages seamlessly right after thawing.

Kernel-Space Virtualized Devices

Processing data on behalf of frozen C_i for hardware devices virtualized in the kernel, such as the alarm or networking devices, can be achieved in the same way. We receive alarm events and incoming network data packets for frozen C_i in the kernel, which buffers the data. We can use UNIX domain sockets with an endpoint in C_0 to inform C_0 to raise a notification in its user interface. In particular, this allows to reliably notify the user about expired timers and alarm clocks previously set in frozen C_i.

Full Background Operability

For all incoming data, we can notify the user in C_i. We can use the multiplexers to show the user a specific notification for different events depending on the incoming data. In practice, only the handling of outgoing traffic for C_i is conceptually confined. Outgoing traffic can, for instance, be responses to incoming messages via the network interface, or be keep-alive packets to remote network servers. For the networking, apps mainly use the Google Cloud Messaging (GCM) infrastructure and receive data, such as instant messages, only when the device reports back to GCM. From the perspective of remote entities, this makes frozen C_i appear to be temporarily lacking connectivity. The remote services not aware of the container encryption buffer their data and send it when C_i reports back. This makes the virtualization layer unable to retrieve further incoming network events.

As network connections are usually end-to-end encrypted, we can neither inspect plaintext data received in C_0 nor keep the connections alive on behalf of C_i. For full network traffic handling, we could virtualize the connection management for C_i in C_0. The downside to this is that this would require to keep the involved connection credentials in plaintext into C_0. Because this makes the credentials susceptible to memory attacks, this is not an option. A more secure possibility regarding memory attacks would be to leverage a remote backend with a copy of the connection credentials. This enables, for instance, the CM to forward the encrypted data to the backend, which can decrypt and interpret the end-to-end encrypted data. The backend can then notify the CM about the particular event and handle the remote connections on behalf of C_i, for example, to the GCM services. The CM in turn can raise an Android intent in C_0 to notify the user about any ongoing network-related events. This would make our prototype fully background-operational. However, keeping sensitive user data in a backend can raise privacy concerns and is thus only advisable for fully corporate-managed containers, i.e., a corporate container. For the users' private containers, this solution is less advisable. Further this causes communication overhead on the device and requires a comparably high implementation effort to realize this functionality. For this reason, our current solution keeps all used connection credentials safe inside all frozen C_i. This constitutes a trade-off between functionality and security. After thawing, we always observed the intended functionality of the apps to continue seamlessly and apps quickly retrieved their data from remote services.

5.3.4 Performance Evaluation and Statistics

In this section, we present our performance results and statistics on the tasks, VMAs, and pages encrypted with F&C on the secure virtualization architecture. We deployed the virtualization architecture with two full-fledged Android 5.1.1 containers in addition to the

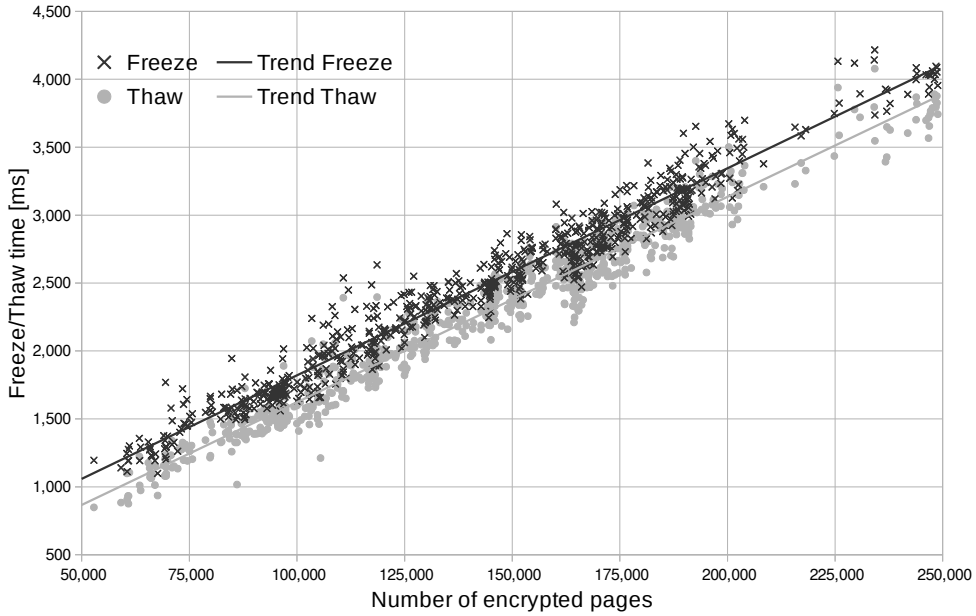


Figure 5.14: Measurements of container freeze (crosses) and thaw (circles) times in milliseconds related to the number of encrypted pages.

management container on top of a single Linux 3.4 kernel (4 KB page size; as on stock Android, no swap or KSM) on Nexus 5 devices (Quad-core 2.3 GHz Krait 400 CPU). This resembles a realistic scenario with a private and business domain. For the evaluation, we intentionally encrypt both containers to challenge F&C with multiple, large cgroups. For common use, encrypting only the business container processing corporate secrets may be a more appropriate choice with higher usability. We adhered to the selection of segments for encryption as described in Section 5.3.3.1.

We identified several test users to utilize F&C on their Nexus 5 smartphones running the virtualization architecture to generate our statistics. To deploy F&C on the phones, we provided the users an update with our modified kernel and virtualization layer enabling main memory encryption support after a device reboot. The users were able to seamlessly continue utilizing their two containers with their data and dozens of apps, for instance, social, business, mail, or media apps. Thawed containers continued running stable, even after a long freeze. The evaluation period also incorporated a number of especially challenging test cases with background events, where users stressed the memory limit or received phone calls in a frozen container. With these test cases, we verified that F&C reliably works even when the system is under heavy load, that the phone is still practically usable and that it remains fully stable.

To gather the statistics, we introduced a performance profiling mechanism to F&C. We extended our prototype to gather the most important statistics in the kernel during encryption, such as the duration of the encryption or the number of encrypted pages and tasks. The extension writes the statistics to a pseudo file after each en-/decryption pass.

Table 5.2: Statistics on the average number of encrypted tasks, VMAs, and pages along with the types of memory regions.

Tasks Total	**Threads**	**Processes**	
1,043	991	52	
VMAs Total	**Skipped**	**Encrypted**	
27,410	14,019	13,391	
Pages Total	**Skipped**	**Encrypted**	
380,545	238,090	142,455	
Encrypted Pages Total	**File-backed**	**Anonymous**	**Text/Data/ Stack/Heap**
142,455	71,606	68,976	1,873

This allowed us to query and store the statistics for every en-/decryption pass in a file, which we pulled from the devices at the end of the evaluation phase. Figure 5.14 illustrates the measured en-/decryption times in relation to the number of pages. The black crosses and gray circles refer to container freeze and thaw times in milliseconds. In total, we have more than 730 container en- and decryption samples where our users ran arbitrarily many apps on the smartphone for an indefinite period of time. The more apps run, the more pages are present, the more pages we encrypt. On average, about 142,500 pages were encrypted in parallel on the four available cores requiring 2,468 ms for freezing and 2,266 ms for thawing. In extreme cases where the Android low memory killer is triggered active and a container exhausts all of its resources, the duration increases roughly to fairly, still practicable, 4,000 ms. When starting the freeze of C_i, the CM directly and quickly switches back to C_0, see Section 3.3. This means that a user intending to switch to another container C_j can start entering the SE's passphrase in C_0 during C_i's encryption in background. Since typing takes some time, the additional cost of a switch usually amounts to only the time required for thawing C_j. Both trend lines in Figure 5.14 point out the linear growth of encryption times and show that decryption is faster than encryption. This is reasonable due to the additional synchronization effort during encryption. The variance originates from the many different events and processes that the kernel schedules in this complex system. The measured encryption time comprises the instant of writing to the freezer state file until user space receives the notification from the freezer. In between, the performance mainly depends on the synchronization inside the cgroup and on the scheduler.

In sum, our solution is suitable for daily use in environments where the protection of sensitive data plays a crucial role. Our prototype device ran without hardware cryptographic accelerators. We expect considerable performance boosts for F&C on devices making benefit of hardware cryptographic accelerators, for example, as available on the x86 architecture, or on devices based on the ARMv8 architecture.

Table 5.2 shows further statistics we gathered. On an average encryption cycle, 1,043 tasks were running, including 52 processes spawning 991 threads. This points out the large

size of the cgroups and that users were running many apps. The second row shows that during freezing, the 52 encrypting tasks identified 27,410 VMAs, where 13,391 were subject to encryption. The remaining 14,646 VMAs represent the non-sensitive segments. Freezing tasks altogether had about 380,500 virtual pages mapped on average. When multiple PTEs point to the same physical page, there are numerous duplicate pages. This is why the total number of encrypted physical pages, about 142,500, is on average clearly smaller and many pages, about 238,000, could be skipped. The average number of encrypted pages accounts for about 560 MB of memory. The bottom row shows the classification of the encrypted pages into the different VMAs. Most pages are either part of file-backed or anonymous segments. Only a small number of pages belong to the stack, heap, data and text segments. A future improvement would thus be to zero out and unmap file-backed segments instead of encrypting them. This comes later at the cost of runtime performance, but probably remains barely perceptible and can clearly improve suspension and resumption.

To measure the effects of F&C on power consumption, we separately conducted power measurements. The additional battery drain depends on the frequency of suspending and resuming containers, as well as on the amount of memory to be encrypted. For our measurements, we automatically switched between the two user containers and the management container, suspending and resuming in 10 second intervals for a 100 times while holding a wake lock. We conducted this process on devices with the same software configuration with and without F&C. We independently repeated that experiment 5 times with fully charged new phones. Prior to each of the repetitions, we started a varying number of applications to test under scenarios from low to high main memory consumption. To be able automatically switch between containers, we adapted F&C to use a fixed key. To track the battery drain from user space, we used the Linux kernel's interface to the *power supply class*. We measured an average battery drain of about 2.5% without and no more than 4.0% with F&C. There is hence little perceptible effect on power consumption under normal phone use.

5.3.5 Security Discussion

In our conceptual security discussion, we consider an attacker who aims at extracting sensitive data from an unattended and non-tampered device under protection of F&C. The attacker obtains physical access to the protected device, has sufficient time and the ability to access both all of its volatile and persistent memory. The local, remote and physical attacker and scenarios one to three in accordance with the attacker model from Chapter 2 are thus in scope. For accessing the memory, the attacker may exploit hardware and software vulnerabilities, for example, through DMA, JTAG or cold boot attacks. This enables the attacker to obtain and analyze its full memory contents. This also allows the attacker to compromise non-suspended parts of the system including the OS kernel. However, the attacker is unable to execute evil maid attacks, i.e., to covertly deploy backdoors on the device waiting for the user to return. This implies that a device once tampered with is not trusted again, for instance, after theft or loss, or because the user notices the tampering attempt. This means that a writing memory attack or successful remote attack as described in Chapter 2 does not remain unnoticed and is thus precluded.

Regarding the virtualization architecture, the adversary may be in possession of the SE,

but lacks knowledge of the SE's passphrase and is unable to unveil the secrets stored on the SE. Furthermore, the attacker has no means to break state-of-the-art cryptographic primitives. An attack scenario may possibly occur in sensitive corporate or governmental domains where the device owner uses a corporate and private container. In this case, the attacker aims to retrieve sensitive data stored in the corporate container after gaining physical access to the device.

In the following, we first discuss the security aspects of F&C itself on the basis of a generic process group and then extend our considerations to the virtualization architecture. As a last point, we elaborate on the precluded unnoticed tampering of protected devices in the absence of their users.

5.3.5.1 Encrypted Process Groups

F&C encrypts the pages of the selected memory segments in RAM using AES in CTR mode and uses its unique physical address as IV for each page encryption. Identical pages are consequently encrypted with a different outcome and since the attacker cannot break cryptographic primitives, encrypted pages reveal no sensitive information. After encrypting a cgroup, we purge the key used in the freezer, as well as other AES remnants and freed pages. When the key is removed in user space as well, the adversary has no means to decrypt protected pages. This means that the attacker can only attempt to obtain sensitive data in non-encrypted pages and segments. F&C allows for the selection of all memory segments for encryption, except the special segments very unlikely to contain confidential data. For encrypted segments, we leave only those pages unencrypted which are shared with unencrypted, running processes. When making sure that processes which share sensitive data with other processes are contained in the frozen cgroups, we do not leave a single sensitive page unencrypted. Even a reduced set of selected memory segments can cover all sensitive areas when assuming knowledge of the segments where processes on the system store their sensitive data.

5.3.5.2 Secure Virtualization Architecture

Regarding the secure virtualization architecture, the goal of the attacker is to obtain the containers' confidential data both from persistent storage and RAM. The security of the platform itself against attackers compromising software components at runtime without memory encryption, for example, for C_0 and the CM, was already discussed in Section 3.3.5. F&C encrypts containers when the user actively switches them to background, locks the phone, or leaves it idle for a specific amount of time. Based on this, we discuss the scenarios where the device is already suspended, where a container is active and unlocked, and where a container is locked, but not yet completely suspended.

Suspended Device

This attack scenario is most likely, because smartphones are suspended most of the time implying that the containers are already encrypted. In this scenario, the attack thus starts when the device is fully suspended. At this point in time, all containers, except for C_0, are encrypted. Containers share no sensitive data with other entities, because the memory usage of physical pages is tied to container boundaries. Since we encrypt all relevant memory segments, see Section 5.3.3, we fully cover the containers' sensitive data in RAM.

F&C purges the unwrapped RAM encryption key in the kernel. Persistent memory is always encrypted due to FDE where the kernel stores the unwrapped FDE key.

On the virtualization architecture, the unwrapped FDE key in the kernel and the unwrapped RAM encryption key in user space are the only assets not protected through the encryption. However, at the moment the container encryption terminates, the CM purges these unwrapped keys, preventing the attacker from decrypting any persistent and volatile memory. The wrapped key complements can only be unwrapped using the SE. The attacker is possibly in possession of the SE, but lacks knowledge of the passphrase and cannot brute-force the SE. Hence, wrapped encryption keys are securely stored in main memory. This means that we need no special key storage, such as CPU registers. The CM only keeps the unwrapped key counterpart in RAM during encryption and when the container is unencrypted. Relocating the unwrapped key for that time, for example, to memory protected through the ARM TrustZone, does not increase our platform's security, as sensitive data is in plaintext anyway.

Being in possession of the device, the attacker can hence only wait for incoming data, such as short messages or network traffic. Since sensitive data is generally end-to-end encrypted and encryption terminates inside the container, the attacker has no means to decipher that data. Unencrypted data, for example, short messages, can already be intercepted before it reaches the mobile device independent of memory encryption.

We assured to leave no sensitive data behind by reading out process spaces as privileged user and by analyzing memory dumps of locked devices with the tool Volatility and with cold boot attacks, such as presented in Chapter 4. On common devices, we were able to easily recover vast amounts of sensitive data, such as passwords for Exchange accounts, FDE keys and further credentials, see Section 4.6. Even though in knowledge of the sensitive assets, we had no means to detect any sensitive data on devices protected by F&C. Listing processes during forensics analyses of the system, one finds that F&C even disguises the processes' arguments and environment variables, since these are part of the encrypted process stack.

Unlocked Foreground Container

In this scenario, we assume that the victim was actively using a container until the moment the attack starts, for instance, in case of coercion or a violent offence. During that time, the actively used container is not encrypted until the system triggers the suspension. When the attacker prevents the container from locking itself, the attacker can directly interact with the container and gather all its data without carrying out a memory attack. With a memory attack, the attacker is able to read the plaintext memory encryption key of the unlocked foreground, which however renders no additional attack vector, as the other containers are encrypted with independent ephemeral, random keys.

Regarding the frozen background containers, the adversary has no influence on their processes, memory contents are not shared between containers and the keys to decrypt pages of frozen containers are not present. The background containers thus remain securely protected.

Locked Foreground Container

A locked foreground container is either just suspending or soon about to suspend after a certain idle time. As in the previous scenario, background containers remain securely protected. When suspension is ongoing, the encryption process terminates quickly even for fully loaded containers, as evaluated in Section 5.3.4. This means that the time frame to extract valuable plaintext data or the encryption key is very likely to be impractical when the attacker gets control over the device at that point in time.

The scenario where the device is not yet suspending the container is similar to the previous scenario with the exception that the foreground container is locked. The attacker can constantly interact with its screen lock app, managing that the locked foreground container never suspends. In the next step, the attacker can either try to exploit the software layer to get access to the non-suspended container, or try to conduct a memory attack to acquire the container's data or its memory encryption key. Configuring containers with a short inactivity timeout until suspension mitigates the occurrence probability of this scenario.

5.3.5.3 Unnoticed Tampering

A memory-encrypted device which the attacker is able to tamper with in the absence of its owner was not part of our security discussion. This would include an attacker who is capable of unnoticedly gaining full privileges or of unnoticedly replaying memory contents, for example, by carrying out a *writing* memory attack. This model is especially hard to defend against and requires additional tamper-resiliency regarding both soft- and hardware level attacks. In case of runtime memory encryption techniques, which transparently encrypt main memory and possibly keep a working set of RAM unencrypted, this scenario would even make the attacker immediately able to extract all sensitive data, because the container remains fully operational and can be resumed at any time. The attacker can either wait for the main application processor to load sensitive memory into the unencrypted working set, or just directly use the acquired privileges to read the memory. Our encryption based on container suspension has the advantage that containers cannot resume without intervention of the legitimate user unwrapping the memory encryption key. In this case, the attacker must wait for the user to provide the key. The attack must be carried out in a way that the user remains unsuspecting, because otherwise the user is unlikely to resume the container.

5.4 A Runtime Memory Encryption Architecture with a Minimal Hypervisor

The main memory encryption architectures presented in Section 5.2 and Section 5.3 protect against memory attacks while the system, respectively containers in case of Freeze & Crypt, are suspended. The architectures enforce that memory can only be decrypted when its user provides a cryptographic token, for instance, an SE or a key derived from a passphrase. However, main memory is only fully protected when the memory attack takes place after suspension is completed. This is effective for scenarios where it is likely that attackers do not acquire access to actively running devices, for example, to unlocked smartphones or idle, but non-suspended notebooks. We elaborated on this in our security discussion in Section 5.3.5. In scenarios in which systems have no suspension features, or in which an attack is likely to take place at any time, for instance, when attackers may get

hold of smartphones in current use by their owners, suspension-based memory encryption may not be applicable or only insufficiently protect the devices. To provide continuous protection in such scenarios, we introduce an architecture for transparent runtime main memory encryption, which we call TransCrypt. This represents Contribution 9. We briefly describe the design of TransCrypt in Section 5.4.1, and its implementation and evaluation in Section 5.4.2.

5.4.1 Design of TransCrypt

We base TransCrypt on a minimal HV which acts as a privileged entity responsible for the transparent encryption of the full memory contents of an arbitrary guest OS. We design TransCrypt to not require any changes to the guest OS and to be almost agnostic to the specific OS running. The HV restricts the guest's direct access to physical main memory to a small and scalable set of unencrypted memory pages, which forms a subset of all the guest's physically allocated pages. The subset of unencrypted pages is commonly referred to as a *sliding window* or *working set*. This working set dynamically changes based on the guest's page accesses. The HV enforces the page access restrictions by making pages accessible/inacessible to the guest by mapping/unmapping the guest's pages in the system's page tables, e.g., nested page tables. The HV maintains these tables for guest memory management. All guest-allocated pages outside the working set are transparently encrypted and unmapped by the HV, which ensures that encrypted pages are no longer directly accessible by the guest. The guest trying to access an encrypted page will be interrupted. This triggers the HV, which decrypts and maps the page on the fly, making the page again directly accessible and thus part of the working set. As soon as the working set is full, the HV evicts, i.e., encrypts and unmaps, the least recently mapped page from the working set. This process is summarized in Figure 5.15. During the boot of a guest OS, the working set will quickly fill and cause most of the guest memory to be encrypted at all times. TransCrypt generates an ephemeral encryption key inside a TEE before booting the guest and exclusively executes page en-/decryption inside the TEE. This protects the encryption key itself from memory attacks at all times. For the page encryption, we propose AES-CBC with physical page base addresses as IVs. This ensures that all pages are encrypted with unique IVs.

TransCrypt potentially allows the HV to dynamically adapt the working set size when the system is under high load to reduce the performance impact caused by the runtime page en-/decryption and re-mapping. An implementation used in productive environments must carefully address the choice of the maximum working set size. The higher the maximum set size is chosen, the less decreases the system's performance. However, this increases the probability that a memory attack reveals sensitive data to the attacker reading the working set. Furthermore, the attacker might be able to take influence on the load of the system to cause the HV to deliberately extend the working set at the time of a memory attack.

TransCrypt supports efficient multi-core operation, where each core manages its own share of the global working set [Hor17]. Similar to suspend-time memory main encryption architectures, the encryption of special pages, i.e., devices mapped to memory, and DMA memory shared with hardware devices, is not possible, because the hardware periphery is neither aware of the encryption, nor in possession of the memory encryption key. TransCrypt

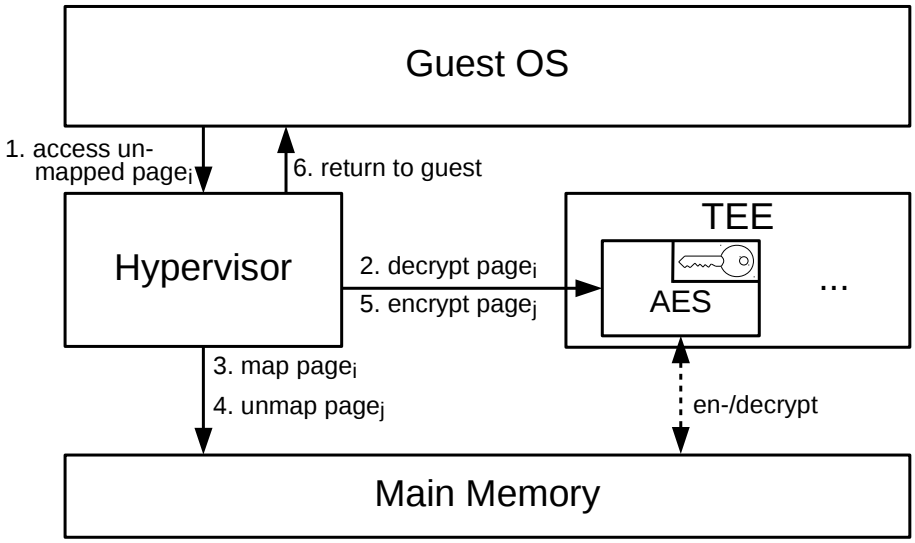


Figure 5.15: Eviction, mapping and en-/decryption of main memory pages from the working set using a minimal HV.

intercepts the guest's access to special pages and excludes them from encryption. This is the only step not independent of the particular guest OS, because TransCrypt requires specific information about the guest's mappings of special pages [Hor17].

5.4.2 Implementation and Evaluation

We realized TransCrypt on the ARM architecture using its hardware support for virtualization. This enables the HV to control the guest's physical page mappings using the Second Level Address Translation (SLAT) virtualization feature. This ensures that each guest access to a page unmapped by the HV triggers a page fault in the HV. Furthermore, using the ARM architecture makes it possible to leverage TrustZone as TEE. We implemented a prototype on a dual-core ARM Cortex-A15 developer board with the ARMv7 virtualization extensions. Our HV uses a page size of 4 KB and does not support multiple guests at the same time. As guest OS, we ran an unmodified Linux 3.0.31 kernel with an Android 4.1.1 environment on top. Running only a single guest does not require to virtualize devices to support multiple, concurrently running guest OSs. This enables us to keep the HV implementation minimal, at about 4,000 lines of code in our case. The dynamic working size set adaption and cipher implementation in the TrustZone for page encryption are not part of our prototype we developed. Instead of using the TrustZone, we realized page encryption directly in the HV as a software AES implementation.

In our evaluation, we identified a reasonable trade-off between performance and the working set size that makes it improbable that a memory attack reveals sensitive data to an attacker [Hor17]. For the performance measurements, we used the CoreMark [Emba] and Antutu [AnT] benchmark tools. Antutu represents a highly stressing real-world test comparable to a demanding app like a 3D graphics game. As the underlying use case, we

defined a typical smartphone usage scenario where a user fetches mails every 30 minutes within 24 hours with the standard Android mail app using an Android mail account with credentials stored. The mail app loads address-password combinations in plaintext into memory, in our case on three different physical pages. Note that such pages are likely to be heap pages and be accessed more frequently, because these may contain other data structures used by the mail app. We determined the working set size that causes these pages to be quickly evicted after being accessed, i.e., as quickly as possible after a fetch. For a working set size of 10,000, we found out that the sensitive pages containing the address-password combinations are inside the working set for about 1%, or less then 15 minutes, throughout the whole 24 hours while sustaining more than 80% of native performance with the Antutu benchmark and almost native performance with CoreMark. For example, for a working set size of 36,000 pages, we achieved around 88% of native performance with the Antutu benchmark, but the sensitive pages were inside the working set for about a quarter of the day yielding a relatively high probability to be acquired in plaintext with a memory attack. With a working set size of 10,000, the sensitive pages were inside the working set for less than 15 minutes throughout the whole day, still achieving about 82% of native performance.

We expect our prototype to produce even better results on devices based on the ARMv8 architecture, which brings hardware AES support and a more efficient infrastructure for SLAT Translation Lookaside Buffer (TLB) invalidation. This makes it possible to even further reduce the working set size.

5.5 Summary

In this chapter, we tackled Challenge 4 by introducing different architectures for main memory encryption. With Contribution 6, we first presented an architecture easily deployable on existing x86 devices This architecture reliably protects confidential data on suspended devices against physical attackers. We used the Linux kernel's FDE key to encrypt the confidential data in volatile memory and purged the key when suspension completes. With an average suspend and wakeup time of about 800 ms and rare peaks of less than 1.5 seconds on mid-performance devices with large main memory, our hardware-independent design forms a fast, secure and portable solution. Existing Linux systems need no more than a kernel update. To wake up from suspend, the end user solely needs to pass the FDE passphrase. This not only preserves high usability of the system, but also prevents end users disabling the OS lock screen from opening doors for everyone after resumption of a suspended device. For end users of Windows systems, for example, we proposed a transparent and lightweight virtualization design variant where the full memory of guest OSs likewise benefits from memory encryption. Further, we emphasized the possibility to combine our architecture with a TPM or SE to prevent brute-force attacks on the FDE key derivation.

In the second part of this chapter, we presented F&C with Contribution 7, a main memory encryption architecture for process groups, along with its successful application to protect mobiles device from physical attackers. F&C builds upon the freezer functionality of the Linux kernel making processes en- and decrypt their memory efficiently in parallel with a transient key. We synchronized the encrypting processes, ensured that frozen processes

do not touch their memory and that external events, such as IPC, are deferred. The prototype we developed can be employed throughout different platforms, kernel versions and allows for the selection of the process groups and memory segments to be encrypted from user space.

With Contribution 8, we combined the memory encryption architecture for process groups with the secure virtualization architecture running multiple Android containers. The platform allowed us to leverage its virtualization and secure key management infrastructure with F&C. This let us realize a fully functional system that thwarts physical attackers while defending against local and remote attackers through container isolation. We encrypted the containers that are not in active use and in order to maintain their background functionality, we informed the user with notifications about incoming events for encrypted containers. In our security and performance evaluation, we showed that the encryption provides strong security for unattended devices and containers not in use. The average en- and decryption time of less than 2.5 seconds makes the prototype practical for use in environments where the confidentiality of data plays a major role.

In the last part of this chapter presenting Contribution 9, we achieved transparent runtime memory encryption with TransCrypt. TransCrypt represents a memory encryption architecture for the ARM platform with an almost completely guest-agnostic HV. Because the encryption key is used throughout the whole runtime of the system in contrast to suspend-time main memory encryption, we require a secure environment for key storage and the cipher implementation, on-chip RAM and the TrustZone on ARM platforms. In contrast to suspend-time main memory encryption architectures, attackers able to time and repeat memory attacks on the target device pose a big challenge. This comes from the fact that the device remains completely operational when the attacker gets hold of the device. This allows the attacker to fully interact with the device, for instance, to wake up a smartphone from suspension, possibly causing apps to fetch remote contents or synchronize with a backend. Especially during this period of time, the attacker might be able to successfully extract sensitive contents. It is thus especially hard to defend a system against attackers with runtime memory encryption mechanisms. We further investigate runtime memory encryption techniques in the presence of attackers acquiring privileges on a system in the following chapter. We show at the example of cloud server technology that even with transparent hardware-based runtime memory encryption support to protect against compromised HVs and physical attackers, the extraction of main memory contents is still possible.

CHAPTER 6

Main Memory Extraction from Encrypted Virtual Machines

In this chapter, we focus on Challenge 5, the memory extraction from platforms with hardware extensions for runtime main memory encryption. We investigate the feasibility of memory extraction at the example of AMD SEV [Advc; Kap17], a technology available on the market and finding its adoption in cloud and virtual server environments. The SEV hardware extension transparently encrypts main memory at runtime on a per-VM granularity. With VM encryption, SEV not only aims to protect from physical attacks, but also from privileged attackers controlling the HV, for example, a malicious administrator or an attacker breaking out of the VM. This chapter demonstrates that sustaining the confidentiality of main memory contents at all times against such privileged attackers poses a severe challenge for the architectural design of runtime memory encryption mechanisms. The contents are based on the work in [Mor18] and [Mor19]. In this chapter's introduction, we first relate the main memory encryption architectures we proposed in Chapter 5 to software-level attacks and then to the AMD SEV technology. Afterwards, we highlight our contributions and the organization of this chapter.

Our methods for main memory encryption presented in Chapter 5 are software implementations, which protect from the presented physical attacker reading main memory under certain conditions. At the example of Freeze & Crypt, the memory contents of a container are fully protected when the container is completely suspended. In case of TransCrypt, confidentiality is achieved with a high probability: when the working set of unencrypted pages contains no sensitive contents at the time of a reading memory attack. In case an attacker not only uses physical interfaces to read out memory contents, but also to gain privileges on the systems, the attacker can directly access the system's memory contents via the OS. With the security model of TransCrypt, being in charge of the OS or HV would allow the attacker to decrypt and read all the VM's main memory contents. In case of Freeze & Crypt, the attacker cannot directly read out the contents of suspended and thus encrypted containers as the decryption key is not present. However, the attacker can read out memory contents of the containers that are not suspended. For these reasons, the scope of our encryption architectures was to only protect against reading memory attacks, see Chapter 2. Considering the example of common multi-tenant systems without memory encryption, the confidentiality of sensitive VM data similarly depends on both the HV's integrity and on the operator's trustworthiness. Unfortunately, this dependency is prone to getting infringed by different attack vectors. Examples are attacks by other tenants exploiting software-level vulnerabilities to escape their sandboxed VMs [Micc; VMw; Xen], attackers with physical access conducting a writing memory attack to gain privileges, for

example, via DMA [Bec05; Boi06; Dev09], or simply a malicious operator using the HV to read the VM's memory.

In contrast to our architectures, AMD SEV aims to protect the main memory of VMs at all times from powerful adversaries. By implementing main memory encryption completely in isolated hardware, AMD SEV not only protects the memory contents of VMs from physical attackers without privileges on the system, but even from a HV-privileged, possibly physical adversary. This allows to reliably protect the memory of VMs running in cloud and virtual server environments from malicious administrators and provides customers the certainty that their VMs' memory contents are encrypted at all times. SEV transparently encrypts all the system's VMs using an SP, a microcontroller coprocessor based on the ARM architecture, also referred to as Platform Security Processor (PSP). On current AMD hardware, the SP is realized on a 32-bit Cortex-A5 core and makes use of TrustZone [Kap16]. The SP uses individual, ephemeral keys for the encryption of each VM's main memory, provides the plaintext contents only to the CPU and ensures that the keys never leave the SP. To attest tenants that their VMs' memory is indeed encrypted, SEV includes a cryptographic protocol to remotely verify VM encryption on an SEV-enabled platform. In this chapter, we demonstrate that it is nevertheless possible for privileged adversaries to extract memory contents in plaintext from VMs.

After presenting related work in Section 6.1 and background on AMD SEV and SLAT in Section 6.2, the first main part of this chapter deals with Contribution 10. We present our framework for main memory extraction from platforms backed with the AMD SEV technology [Mor18]. We call the framework *SEVered*, enabling to extract the memory contents of SEV-encrypted VMs running on top of a malicious HV. While SEVered allows the extraction of data, it does not provide concepts for the targeted extraction of secrets. In Section 6.4, the second part of this chapter, we focus on Contribution 11, based on [Mor19]. We present a method that extends our memory extraction framework with the capability to specifically extract targeted secrets like TLS, SSH, and FDE keys, in short time instead of extracting large amount of VM memory to find secrets. This not only reduces the time required for the extraction, but also the overall incurred footprint of memory extraction. We complete the chapter by presenting countermeasures in Section 6.5.

Contributions 10 and 11 resulted from joint work with Mathias Morbitzer and Julian Horsch published in [Mor19; Mor18]. All authors collaborated in the different phases of the work. While Mathias Morbitzer is the main contributor to the implementation and evaluation in [Mor18], Mathias Morbitzer and Manuel Huber contributed as first authors equally to the implementation and evaluation in [Mor19].

6.1 Related Work

In the following, we present related work on memory extraction and protection of encrypted VMs. While there are established VMI frameworks [Lib; Rek; Theb] for data analysis and extraction for unencrypted VMs, the systematic extraction of memory from encrypted VMs has not been subject to extensive study. On AMD SEV platforms, the SP protects page encryption and the corresponding keys from the HV. This makes it infeasible to directly read memory contents from SEV-enabled VMs as long as the SP cannot be compromised [CTS18]. However, Payer [Pay] early pointed out the missing integrity protection on AMD

SEV platforms.

Buhren et al. presented an attack [Het17] to gain remote code execution with user privileges on an SEV-enabled VM. Their approach exploits VM memory remapping to modify the control flow of an SSH service. The first step is an off-line tracing of the system call sequences performed during an SSH login on a comparable, unencrypted VM. The goal of this analysis is to determine the behavior of a VM accessing the login information of the SSH session, the *credentials data structure*. The next step is to wait for a victim user to login to the SSH service. With the information gained in the off-line analysis, the authors identify the memory page containing the user's login information. They then try to illicitly login by remapping the valid user's credential data structure to the one the SSH service creates during the illicit login attempt. This allows the attacker to re-use the victim user's login information. In their evaluation, they achieved a success rate of around 23%. The low rate was primarily caused by the fact that the SSH service may store the *credentials data structure* at different offsets within the page. As a condition for a successful attack, the SSH service must have stored both the victim user's and attacker's credentials data structures at the same page offset. Besides being quite invasive, this approach requires access to a comparable VM, detailed analysis of the SSH service, user interaction, and data incidentally be stored at specific offsets.

The attack described in [Du17] follows the same goal of gaining remote code execution on an SEV-encrypted VM, but does not exploit remapping. Instead, the authors describe a *ciphertext block move* attack, which also exploits the missing integrity protection. The authors argue that it is possible to move memory contents in physical memory. This is because the Host Physical Address (HPA) is not part of the AES-based encryption scheme itself but is incorporated into the encryption result in a later step with a reversible physical address-based tweak algorithm that uses static parameters. After reversing the tweak, ciphertext can be moved and the tweak re-applied with the target HPA. The authors describe a method that moves the pages to exploit an SSH process. Both the approaches in [Het17] and [Du17] were, to the best of our knowledge, not confirmed on real SEV hardware. The *ciphertext block move* attack could possibly be leveraged for the memory extraction as an alternative to the remapping in SEVered. This could cause the service returning responses to our requests with an unmodified HPA mapping, but instead with different memory contents at the HPA.

Published after [Mor19; Mor18], the authors in [Li19] presented a method using unprotected I/O operations as en- and decryption oracles. The HV is not only responsible for a VM's memory management, but also for its I/O operations. Because not all I/O data can be end-to-end encrypted, the HV can directly read or write on unencrypted I/O data in DMA buffers when entering and leaving the VM. An example is network packet header information of SSH traffic. On the one hand, this allows a malicious HV to modify incoming I/O data in DMA buffers before it is passed into the VM. On the other hand, this alows allows crafting a decryption oracle. For this purpose, the authors combine page tracking [Mor18] and the ciphertext block move attack [Du17]. Using page tracking, they identify the encrypted page in the VM that contains the network packet that the VM writes to the unprotected I/O buffer. In particular, they identify the encrypted network packet header. Using the ciphertext block move attack, they move an arbitrary ciphertext

block of the VM to overwrite the identified network packet header. This causes the moved ciphertext block to appear to the HV in plaintext instead of the original network header. In the last step, the HV restores the network header with proper values. Their paper contains an extensive evaluation and discusses several mitigation approaches including a temporary software solution. An advantage over SEVered is that their attack is more stealthy, because no network traffic has to be crafted by the attacker, and because the attack is not directly perceivable, as the network traffic is ultimately unmodified, unlike in SEVered. A drawback of their approach is that they depend on traffic caused by SSH connections from outside (such as through an administrator's keystrokes), which allowed to extract only 16 bytes per transmitted network packet.

Also after [Mor19; Mor18] was published, a new class of attacks against encrypted VMs, called *inference attacks*, was introduced by [Wer19]. Inference attacks allow application fingerprinting and, depending on available side channels, extracting sensitive memory as well as injecting data into VMs. The authors combined system call monitoring of VMs using hardware debug registers with monitoring the general purpose register contents on SEV platforms. This allows to profile instructions and data accesses in the guest. On SEV where, unlike in SEV Encrypted State (SEV-ES), the Virtual Machine Control Block (VMCB) containing the general purpose registers is not encrypted upon a VM exit event, inference attacks allow to read TLS traffic in plaintext or to inject attacker-controlled data when a guest reads from disk. For the latter, this might be an attacker's public key in case of an SSH server. This can ultimately lead to unconstrained SSH access to the VM. Because SEV-ES eliminates the general purpose register side channel, the described attacks are rendered ineffective on platforms with the ES extension. Nevertheless, the authors demonstrated that application fingerprinting is still possible. Leveraging Instruction-Based Sampling (IBS) hardware features, they showed how to identify applications and their version running in encrypted VMs. This method can, for instance, be leveraged as a preliminary step towards a remote attack on application layer, or to undermine a VM's Address Space Layout Randomization (ASLR) protection, as the authors argue.

On the side of defenses, Fidelius [Wu18] is a software-based extension to SEV. This extension is a privileged module separate from the HV that restricts the HV from accessing specific critical resources with *non-bypassable memory isolation*, for instance, to prevent replay attacks. The authors provide a VM lifecycle concept that describes how to start Fidelius and provide tenants proof that the system runs Fidelius in addition to SEV. This requires trusting the Fidelius module instead of the operating HV.

Intel announced the implementation of its own hardware-based memory encryption approach called MKTME [Int]. According to our understanding of the specification, MKTME in not designed to protect from a malicious or compromised HV, but only from memory attacks from outside. The HV remains, for instance, capable of enabling or disabling the encryption, or to handle the sharing of memory with other VMs.

6.2 Background on AMD SEV and SLAT

This section provides background information on AMD SEV and the SLAT concepts of HVs.

SEV

The AMD SEV technology allows for the transparent encryption of main memory of individual VMs. SEV primarily targets server systems and builds on the AMD SME technology, which provides transparent full main memory encryption. While the goal of SME is to protect systems against physical attacks on the main memory, SEV tries to additionally protect memory of individual VMs against attacks from other VMs and from a malicious HV. The SEV encryption is executed by a hardware AES engine located in the memory controller. The keys for the encryption are created and managed by an additional component, the AMD SP. All keys are ephemeral and never exposed to software on the main CPU. In contrast to SME, SEV uses different keys for each VM and for the HV. Additionally, a VM running on an SEV-protected system can request encryption and receive proof that its memory contents are being encrypted, which establishes trust between its owner and the remote operator. While SME was first integrated into AMD's Ryzen CPUs, SEV was introduced onto the market with the EPYC processor family. The mainline Linux kernel provides necessary software-level support for SEV.

The SEV-ES feature [Advc] complements SEV. While SEV encrypts the VM's memory, SEV-ES encrypts all information about the VM's state that is not required by the HV to work properly. The information contained in the VMCB is therefore split into an unencrypted *control area* and an encrypted *save area*. The save area contains all VM registers which are therefore protected from a malicious HV.

SLAT

AMD SEV integrates with the existing AMD hardware virtualization technologies marketed as AMD-V. An integral component of hardware virtualization is an additional address translation, often named *nested paging* or SLAT [Adva]. While non-virtualized systems simply translate virtual addresses directly to physical addresses, a hardware-virtualized system distinguishes between three different types of addresses. When the VM accesses a Guest Virtual Address (GVA), the guest-controlled first level translation translates the address to a Guest Physical Address (GPA). The GPA is then translated to a HPA using the second-level translation controlled by the HV. SLAT is completely transparent to the VM. This allows running multiple VMs that use the same GPA space while separating them in physical memory. With SEV enabled, the first level translation from GVA to GPA in the encrypted VM is non-accessible to the HV. But the HV is still responsible for managing physical memory for its VMs and remains therefore able to restrict access and change second-level mappings from GPAs to HPAs. Since there is no integrity protection in SEV, the HV can use SLAT to transparently switch a GPA to HPA mapping to a different HPA page belonging to the same VM.

6.3 Design and Implementation of the Memory Extraction Framework SEVered

We discussed that AMD SEV transparently encrypts the VMs' memory on a per-page basis using the SP, but misses to protect the integrity of encrypted memory. As the HV is in charge of the SLAT on SEV platforms, i.e., of VMs' second-level memory management, the HV remains capable of modifying the GPA to HPA memory mappings of the VMs. This allows a malicious HV to alter a VM's memory layout and thereby influence the code the

VM executes and the data it accesses. The missing integrity protection was also referred to in [Pay] and exploited in [Het17] to obtain user privileges via an SSH service on an AMD SEV-enabled VM.

With SEVered, we exploit the remapping capability to trick a service inside the VM offering a resource, such as an HTML page or a file offered for download, into accessing and responding a different area of VM memory. This allows the extraction of arbitrary VM memory by repeatedly requesting the same resource from the service while remapping its memory with the HV. According to the protection AMD SEV seeks to offer, we may well assume the attacker to be in charge of the HV. This situation is exemplified with Figure 6.1, where a target VM offers a resource to the outside via a remote service. The illustration shows that the VM's OS kernel is responsible for the GVA to GPA mappings, which is due to memory encryption opaque to the HV. Further, the illustration depicts the HV modify a second-level mapping in the HPA space to make the service deliberately return a different resource.

SEVered neither requires physical access to the system, nor VMs colluding in the attack. SEVered only requires a victim VM to offer a remote service, which is accessible to the adversary, such as a web or SSH server. In virtual server and cloud environments, remote communication services are typically available to the outside, and the services typically provide resources for remote peers. We designed SEVered as a two-phased attack, starting with a resource identification phase before the subsequent memory extraction phase. In the following, we first describe the two phases and then present the implementation and evaluation of our prototype.

6.3.1 Resource Identification Phase

The main challenge to be able to extract memory from an AMD SEV-protected VM is to accurately identify the location of the resource that can be used for remapping in memory. In particular, this means for the malicious HV to determine the GPA to HPA mappings pointing to the physical memory pages containing the resource. As described, this resource might be an HTML page offered by a webserver, which we can request from outside. As AMD SEV is typically utilized in server environments, the resource identification should even work reliably when the VM is under high load from remote peers accessing different services and resources at the very same time the attack takes place.

SEVered iteratively determines the GPAs pointing to the resource in memory based on a *page tracking* technique. We use the HV's SLAT capability to invalidate all the PTEs of the target VM right before we make a request to the target resource. This removes the *present* flag from all the VM's PTEs and causes a page fault in the HV for each subsequently accessed page. During our immediately following request, the VM *must* access the requested resource in memory to serve our request, and we will therefore receive one or more page faults in the HV caused by the access to the requested resource. For each faulting page access during our request, we re-validate the PTE and record the accessed GPA in the HV as *candidate* for the target resource. Note that a resource can be contained within a single page, but also spread over multiple pages, depending on the resource's size.

Executing this step only once likely renders a large set of candidates reflecting all the code pages the VM executes and all the data pages it accesses while processing our request.

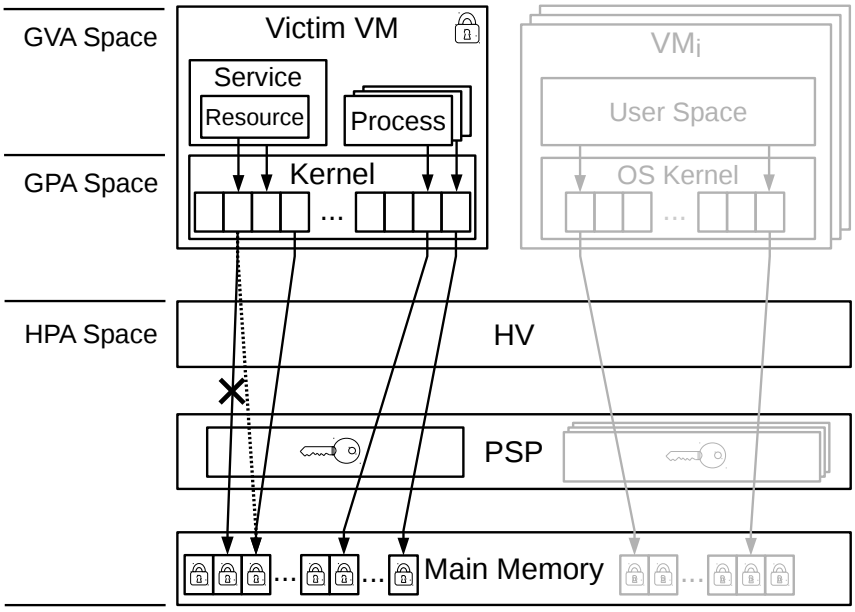


Figure 6.1: Modification of a VM's memory memory mappings using a HV on an SEV-enabled platform.

With a high number of concurrent accesses, for instance, caused by high load caused by other remote peers, this set tends to be even larger and fuzzier. SEVered uses an iterative, probabilistic approach in which we repeatedly invalidate all PTEs while making requests, alternating between requesting the target resource and a *different* resource from the same service. By properly intersecting and subtracting the sets of tracked page accesses, SEVered converges after a varying number of iterations to an accurate set of highly *likely candidates* that represent the target resource [Mor18].

Intersecting the set of candidates from all the tracking phases of our requests to the target resource clearly reduces the set of likely candidates down to exactly those pages accessed every time the VM serves the same request. The intersection clearly reduces the fuzziness caused by other activities inside the VM. However, the set of likely candidates still includes the numerous code and data page accesses made for serving a request in general. This means that the set of likely candidates does not yet explicitly represent the target resource. This is why we complement our requests to the target resource with the requests to a different resource from the same service. In these cases, the set of tracked pages is *unlikely* to contain the target resource when requesting a different resource, but will contain all the pages accessed for serving a comparable request. This insight allows filtering and further reducing the set of likely candidates, but turns our approach into a probabilistic model.

The degree of unlikeliness that our request to a different resource involves a page access to our target resource depends on the concurrent activities the VM executes. For instance, when another remote peer requests the target resource while we track pages accesses for

our request the different resource, we will find the target resource in the set of *unlikely candidates*. The probability that this situation occurs likely increases with increasing load on the target VM. However, by applying a flexible number of iterations until we reach a threshold for the probability of having correctly identified the target resource in the set of *likely candidates*, we minimize this uncertainty [Mor18]. In case SEVered fails to correctly determine the GPA to HPA mapping representing the resource, the attack will fail. Instead of remapping the memory of the target resource, we would unpredictably alter the VM's memory layout. This would likely introduce unexpected behavior inside the target VM, for instance, causing the web service to crash. In such cases, the victim might notice the attack, and the attacker must repeat or continue the resource identification procedure to refine the set of likely candidates.

6.3.2 Data Extraction Phase

After determining the mapping of the resource in memory, the attacker can repeatedly request the resource and use the HV to remap the resource to arbitrary VM memory until the desired memory contents are extracted. After that, SEVered restores the original mapping.

SEVered relies on the target resource to have a constant mapping inside the VM throughout the identification and data extraction phases. Cases where the mapping changes can be due to the OS performing memory maintenance operations, for instance, when the VM runs out of memory and swaps contents. For this reason, a resource that is *sticky* in memory represents a suitable target. An example is a file-backed resource, such as an HTML-page, which is under normal conditions sticky, but when the VM is low on memory, the page may be removed from memory and when it is accessed again, it is re-read from disk while the OS kernel creates a new mapping.

During the data extraction phase, other peers requesting the same resource will also receive the remapped memory contents. As this can cause attention, the HV can either block requests to the VM in the meantime, choose a resource that is not frequently accessed, or wait for an adequate point in time where the service is less frequently accessed.

Not only the throughput of the service, but also the size of the resource plays an important factor for the throughput to be achieved during the extraction phase. The larger a resource, the more memory pages can be remapped at a time and the less requests are required to extract the desired amount of memory. The most suitable resources are covering one or more full pages in memory, such as HTML pages or files offered for download. Having only access to a small-sized resource, for instance, a data structure within a heap page, SEVered may not be able to extract the full memory contents, but when swapping mappings, only the respective parts of other pages. Furthermore, small sized-resources are likely to contain other data that is accessed by the service or by other processes in the VM, such that the re-mapping can corrupt the memory space of the VM. Note that the attacker can disguise the attack pattern by making all requests from different, independent peers during both the phases.

6.3.3 Implementation and Evaluation

We implemented a prototype of SEVered on an AMD EPYC 7251 processor with SEV fully enabled, running Debian Linux in version `4.13.0-rc1` and QEMU in version `2.9.50` with the SEV patches provided by AMD [Advb]. We introduced SEVered's code into KVM, which we use as malicious HV. This included our page tracking mechanism, which we based on KVM's infrastructure for guest write access tracking [Gua], and the code for modifying the guest VM's memory mappings. Our host system works on a page size of 4 KB.

In our evaluation [Mor18], we used two web servers, Apache `2.4.25-3` and nginx `1.10.3-1`, and one SSH server, OpenSSH `7.4`. We introduced a noise model with different noise levels representing the load on the target VM caused by concurrent accesses from the outside to the VM's different services. As a target resource, we used a webpage covering exactly one file-backed page in memory. On each noise level, we executed a randomized access pattern for the target resource and for different resources on the target service, as well as for resources of the other services. Our evaluation considered noise levels from 20 to 50, representing the number of concurrent accesses per second from arbitrary peers both during the identification and extraction phases. Our results show [Mor18] that the set of likely candidates for the target resource converges even under high load after about 20 iterations for the web servers, which we made in around 20 seconds, to less than 5 highly likely candidates in our case. For the SSH server, we required more than a hundred iterations at a noise level of 50, but on average only 46 iterations, took about 111 seconds at a noise level of 40. For all noise levels, we had a high probability that other peers request the same resource during our identification phase, which introduced noise into the sets of unlikely candidates in our identification algorithm and thus affected the required number of iterations to reach convergence. We recorded on average about 8,300 to nearly 16,000 different page accesses per request in case of the web servers and between 19,000 and 25,000 different page accesses for the SSH service.

These results show that SEVered is capable of efficiently eliminating candidates in the large set of page accesses and of reliably determining the resource in memory. We were *always* able to identify the target resource for all services. In case of the web servers, we converged to two, respectively three candidates, to be highly likely to represent the target resource. Among these candidates, the target resource was *always* the last tracked candidate, making our attack very reliable. This behavior represents cases where a service first opens a socket before finally transmitting the requested content. The code pages for opening the socket are accessed on every request and thus also highly likely to be part of the set of our highly likely candidates.

For our single page-sized resource, we achieved an extraction speed for the different services between about 40 and 80 KB per second. In practice, the extraction speed both depends on the size of the target resource determining the number of required iterations, and on the response times of the service.

6.4 A Method for the Targeted Extraction of Secrets

Attackers are most likely interested in high-value resources, such as in TLS or SSH keys of web or SSH servers, or in a VM's FDE key. However, the encryption prevents the attacker

from locating the VM's most valuable resources in memory prior to extraction. Naive extraction of VM memory using SEVered until finding secrets takes considerably more time than a targeted extraction of the memory regions where specific secrets are located. In the worst case, extracting those secrets requires a full dump of the VM's memory. This can take a significant amount of time, depending on the size of the attacker-controlled resource and throughput of the service. For example, SEVered reached an extraction speed of about 80 KB/s with web servers providing a resource covering exactly one memory page. In this scenario, it takes more than 7 hours and requires 524,288 requests to extract all memory contents of a VM with 2 GB of main memory. During this time, other clients requesting the same resource also receive arbitrary contents, making full memory extraction conspicuous.

In the following, we provide with Contribution 11 a method for the targeted extraction of key material using SEVered. This method makes HVs capable of quickly locating and extracting specific secrets from SEV-enabled VMs. Our method has two phases, the *observation* and the *retrieval* phase. In the observation phase, we exploit the fact that the HV is able to observe certain events triggered by VMs. These *observable events* can, for instance, be page faults which the HV handles but also I/O events like network traffic or disk writes. We observe and combine such events to identify a minimal set of VM memory pages likely to contain the targeted secrets. Second, in the retrieval phase, we iteratively extract and analyze the identified set of pages on the fly until we find the targeted secret. For this phase, we use the SEVered attack, but could potentially leverage other vectors allowing memory extraction from SEV-encrypted VMs. Like SEVered, our method neither requires breaking SEV's cryptographic primitives, nor control over the SP. Likewise, our method requires control over the HV, i.e., a malicious administrator or a compromise of the HV.

Our targeted extraction approach offers an inconspicuous, reliable and efficient method to steal various secrets from encrypted VMs. We demonstrate the potential of our approach by extracting TLS and SSH keys from a VM's user space memory, and FDE keys stored in the VM's kernel space. We conduct our experiments on an SEV-enabled EPYC processor running Apache and nginx web servers as well as the OpenSSH server. To show that our approach can cope with real-world scenarios where VMs can be under varying levels of load, we base our experiments on a load model in which multiple independent clients concurrently access the VM's services during our attack.

6.4.1 Design of the Method

Our concept for the targeted extraction of secrets from SEV-encrypted VMs has two phases: In the first phase, we start by *observing* the page accesses of the targeted VM in the HV until an event occurs which indicates the VM's recent use of the targeted secret, e.g., a TLS or SSH key. In the second phase, we *search* the VM's memory for the secret by systematically extracting and analyzing the set of observed page accesses. This section describes both phases in detail.

6.4.1.1 Observation Phase

The goal of the observation phase is to narrow down the set of VM memory pages possibly containing the targeted secret. We start the phase at an arbitrary point in time by tracking

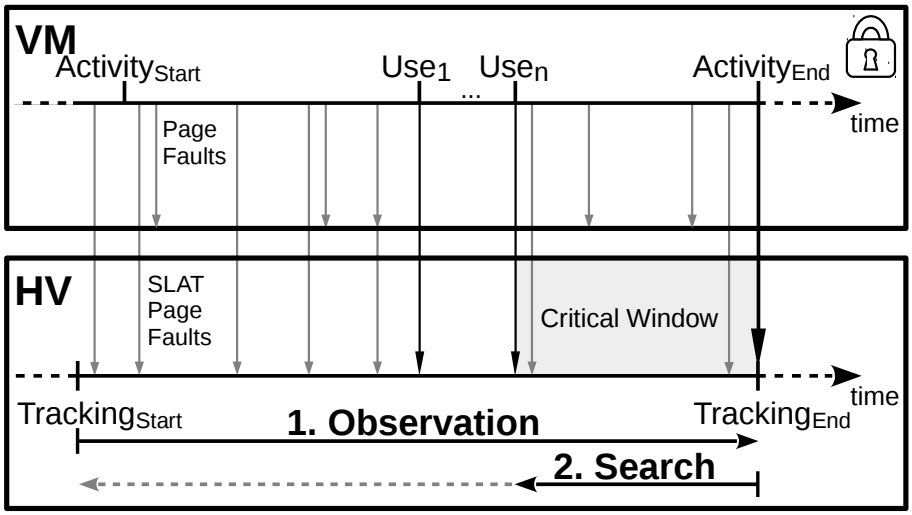


Figure 6.2: A HV first observing an activity inside an encrypted VM and then searching for the targeted secret. The vertical lines crossing the VM boundary into the HV box depict the events observable outside the VM.

the VM's page accesses in the HV until observing the end of a particular *activity*. This activity *must* make use of the targeted secret *at least once*. The start of the activity, denoted by $Activity_{Start}$, does not need to be observable by the HV. In contrast, the end of an activity, called $Activity_{End}$, *must* be a HV-observable event. This event indicates that the VM *recently* used the secret one or multiple times, denoted by $Use_1..Use_n$. As soon as we observe $Activity_{End}$, we stop tracking, denoted by $Tracking_{End}$. We do not actively attempt to trigger $Activity_{Start}$ in order to interfere as little as possible.

To start page access tracking, denoted by $Tracking_{Start}$, the HV invalidates all the target VM's GPA to HPA mappings. As a consequence, each of the VM's page accesses causes an observable event, a SLAT page fault. For each SLAT page fault, we record the GPA as well as the time and type of the page access (read, write, execute) in a list and re-validate the mapping. The re-validation *clears* the page from tracking. This means that each accessed page triggers exactly one page fault and that we track the page *exactly once*, namely the first time it is accessed after $Tracking_{Start}$. The tracking enforces that accesses to the secret will inevitably be recorded. Note that the secret can be contained in a single page or span over multiple pages and can have multiple occurrences on different pages. An example for an activity is a TLS handshake as part of a request to a web server. The server uses the targeted secret, in this case its TLS private key, to authenticate itself to a client during the handshake. The HV can observe $Activity_{End}$ by monitoring network traffic, waiting for the packet the VM sends to complete the handshake.

Figure 6.2 depicts an attack scenario with the target VM and the HV in the upper and lower box, respectively. The illustration shows the start and end of a VM's activity along with events triggered by the activity, such as $Use_1..Use_n$. The vertical arrows crossing the upper and lower box represent the events observable from the HV. These are, for instance, SLAT page faults, network packets or disk I/O. Some of the vertical arrows do not cross

the boundary of the VM. These are events not observable by the HV, for example, page faults handled by the VM or possibly $Activity_{Start}$. Some of the events may be related to concurrent activities, and multiple other activities may potentially make use of the secret as well, cases which are not depicted in Figure 6.2. The illustration emphasizes that $Tracking_{End}$ concludes the observation phase right after $Activity_{End}$.

When starting tracking between Use_n and $Activity_{End}$, we do not observe any of the page accesses to the secret. This means that we are unable to find the secret in the later search phase, requiring to repeat the attack. This is why we call the timespan between Use_n and $Activity_{End}$ the *critical window*. The critical window size is an important factor regarding the quality of the attack. The smaller the critical window the higher the probability that the attack succeeds. Further, a small critical window means quick termination of page access tracking after Use_n. This causes Use_n to be tracked *at the very end* of the phase, and likely only a few more pages to be tracked after Use_n. We evaluate the critical window size for different scenarios with various levels of load in Section 6.4.3.

It is *not* necessary to synchronize the start of the observation phase with a possibly non-observable $Activity_{Start}$. If $Tracking_{Start}$ takes place *long before* $Activity_{Start}$, the observation phase might take longer, but since every page is tracked only once, this does not lead to a persistent performance impact. On the other hand, if $Tracking_{Start}$ takes place *after* $Activity_{Start}$ (but not inside the critical window), the tracking period will be shorter and likely output less tracked page accesses.

To conclude, the result of the observation phase is a list of pages in which the page with the targeted secret is contained at least once as long as $Tracking_{Start}$ is not inside the critical window. The set of pages in the list is significantly smaller than the whole set of the VM's pages.

6.4.1.2 Search Phase

The goal of the search phase is to extract the targeted secret from the VM's memory as quickly as possible, i.e., with a minimal number of memory requests. The input to the search phase is the list of tracked pages acquired during the observation phase. It is unknown which of the page accesses in the list correspond to $Use_1..Use_n$. The naive extraction of all pages in the list would still require a fairly high number of memory requests to find the secret. In the following, we describe our approach for a more efficient extraction.

The search phase starts right after $Tracking_{End}$, as depicted at the bottom of Figure 6.2. We know that $Activity_{End}$ indicates *recent* use of the secret. This means that Use_n must have occurred shortly before $Tracking_{End}$. For this reason, we consecutively extract the tracked pages *in backward order* until we find the secret. We thus start the extraction with the *most recently* tracked pages. This backward search is shown by the arrow directed to the left at the bottom of Figure 6.2. We analyze extracted memory chunks for the presence of the secret *on the fly* to be able to terminate the extraction procedure as early as possible. On the fly means we search the latest extracted memory chunk for the secret while we request the next chunk. When finding the secret in the chunk, we terminate the search phase, otherwise we request another chunk. The actual analysis is specific to the targeted secret and described in Section 6.4.2 for different secrets.

We propose an optional *preprocessing* step before the extraction to further minimize

the number of memory requests. Preprocessing *filters* page accesses from the list, which *cannot* represent a use of the secret, and *prioritizes* accesses that are *likely* to represent a use. The ability to filter and prioritize depends on the use case, in particular, on the specific activity and secret. In most cases, the secret is a data structure on a page in non-executable memory, allowing to filter all execute-accesses from our list. The page is likely to be read, but may also be written during an activity. Depending on the use case, it is also possible that the secret resides in a read-only area, or represents confidential code. The information about this can often be acquired prior to the attack. A further possibility for preprocessing is to conduct a representative offline access pattern analysis for the activity to observe the expected timing of $Use_1..Use_n$. An offline analysis is more representative the more the hardware platform and the software configuration inside the VM resemble the attack target. With the gained timing information, an attacker can further filter or re-prioritize pages in the list.

Extracting the secret from the encrypted VM using SEVered requires the secret to remain at the same location during the attack. This means that the secret must not be erased or moved to a different HPA by the VM's kernel before the search phase terminates. We show that the secrets we chose for extraction always fulfilled this requirement and investigate preprocessing possibilities as part of our evaluation in Section 6.4.3.

6.4.2 Key Extraction Scenarios

In the following, we describe the application of our concept for the extraction of targeted secrets at the example of private keys and symmetric FDE keys. We focus on the aspects from Section 6.4.1 that are specific for the type of secret. These aspects are the activities with their events and the on the fly analysis.

6.4.2.1 Private Keys

For the extraction of private keys, we focus on the example of web server TLS keys. These keys are resources located in a VM's user space and highly sensitive. Web servers use these keys to establish authenticated TLS channels with clients. An attacker can make use of a stolen private key for identity spoofing and deceive clients for fraud or data exfiltration.

Events. $Activity_{Start}$ is the start of a TLS handshake. The handshake can be part of an HTTPS request or be directly triggered by a client. Use_i represents a server's use of the TLS key for its authentication during the handshake. The exact moment of use depends on the key exchange method. For instance, in case of an Elliptic-Curve Diffie-Hellman Ephemeral (ECDHE)-based key exchange algorithm, this is the moment of signing curve parameters. For an RSA-based key exchange, this moment is the decryption of the premaster secret encrypted by the client with the server's public key. $Activity_{End}$ happens when the VM sends the client a specific network packet during the handshake. We observe these packets with network monitoring tools. The *change cipher spec* packet is an indicator independent of the specific key exchange algorithm. Depending on the algorithm, packets sent earlier may be usable indicators as well. Note that we can also observe or even trigger $Activity_{Start}$ ourselves in this scenario. We discuss this aspect in Section 6.4.4.

On the fly analysis. The public key and its length are part of the server's certificate and known in advance. When using RSA, the private components of the key are the factors p or q of known length dividing the modulus of the public key. For every extraction request we make, we traverse the extracted chunk of memory and check if it contains a contiguous bit sequence that divides the modulus without remainder. If so, we found either p or q and can instantly determine the other factor. Otherwise, we request the next chunk of memory. Analyzing a chunk this way usually takes less time than memory extraction with SEVered, see Section 6.4.3.

The same approach can be used for extracting SSH private keys. In the SSH scenario, the SSH server must also use its private key for authentication during the SSH handshake when establishing a session. We evaluate the extraction of TLS and SSH keys using the Apache, nginx and OpenSSH servers in Section 6.4.3.

6.4.2.2 FDE Keys

The normal approach when using SEV is to first perform an attestation of the platform. The attestation proves to the tenant that the VM has been started with SEV enabled. After a successful attestation, the tenant provides the FDE key in encrypted form to the VM [Advc]. This protects the key from eavesdropping adversaries in the network and from being read by the HV. Thereafter, the FDE key is present in the VM's memory and can be extracted with our approach. The FDE key is particularly important, because it allows attackers to decrypt the VM's persistent storage gaining access to further valuable secrets.

Events. The corresponding activity is a disk I/O operation. The trigger for $Activity_{Start}$ is not observable by the HV and unlike in the TLS key scenario, $Activity_{Start}$ can have many different triggers. The trigger can, for instance, be data uploaded to a service, a request to a web server being logged, or an operation of the VM's OS involving disk I/O. The event Use_i is the VM's use of the FDE key to en- or decrypt disk content to be read or written. We observe $Activity_{End}$ by monitoring the VM's disk image file in the HV.

On the fly analysis. We can be sure that we found the secret as soon as we are able to successfully decrypt the VM's persistent storage. Traversing extracted memory chunks and naively trying each possible sequence as key leads to an inefficient approach. Our goal is thus to first identify key candidates in extracted memory chunks. For this purpose, we search the extracted memory for characteristics specific to FDE keys based on the following two criteria.

First, the FDE key is stored in the VM's kernel in a specific data structure. This structure has various fields, some of which must have certain value ranges, for instance, kernel addresses pointing to other kernel objects. Our first criterion for a key candidate is thus the identification of possible FDE key structures in extracted memory chunks.

Our second criterion is based on the statistical properties of the FDE key. Because FDE is usually AES-based, the kernel derives round keys from the FDE key and

keeps them in *AES key schedules* in memory. The round keys have common statistical properties that can be identified with linear complexity. The first-round key is the AES key itself. We use the tool `aeskeyfind` [Pri] to search memory chunks for AES key schedules. Note that candidates that turn out to be false positives are possibly symmetric keys used for other purposes and might also be valuable secrets. The traversal of memory chunks based on these two criteria takes considerably less time than the extraction of memory with SEVered, see Section 6.4.3.

We evaluate the FDE key extraction scenario as part of the following section.

6.4.3 Implementation and Evaluation

In the following, we first define performance indicators and then present our prototype and test setup. Based on that, we evaluate the extraction of TLS, FDE and SSH keys, as discussed in Section 6.4.2. In the final part of our evaluation, we present strategies for optimization with preprocessing and summarize our results.

6.4.3.1 Performance Indicators

The key factors we investigate are the *success probability* and the *attack time*.

Success Probability

As discussed in Section 6.4.1, the critical window size is the factor determining the success probability of our attack. The smaller the critical window, the smaller the probability that the observation phase ends without having tracked the access to the secret. In our evaluation, we present the success probability for the tested scenarios and provide an upper bound on the size of the critical window. We call the upper bound the *reaction time* of our attack. The reaction time is the sum of the critical window (the time frame between Use_n and $Activity_{End}$) and the time our prototype requires to detect $Activity_{End}$ and stop tracking (the time frame between $Activity_{End}$ and $Tracking_{End}$).

Attack Time

We divide the total attack time of a full attack into its three components: the time required to setup SEVered prior to extraction, the duration of the observation phase and the duration of the search phase.

Setup of SEVered. The time required to setup SEVered is evaluated in [Mor18] and is thus not subject of our evaluation. Setting up SEVered usually takes less than 20 seconds, depending on the load of the VM. After setting up SEVered once, we can arbitrarily extract the victim VM's memory and repeat our attack when necessary.

Observation phase. The main factor for the duration of the observation phase is the frequency of the targeted activity. For instance, a web server under high load will often make TLS handshakes while SSH logins generally occur less frequently.

Search phase. The duration of the search phase is mainly determined by the amount of memory that has to be extracted until the secret is found. This is driven by the number of pages we track within the reaction time frame. The reaction time not only provides an upper bound on the critical window, but also serves as indicator for the expected number of tracked pages.

In our evaluation, we investigate the number of pages that have to be extracted, and the duration of the observation and search phase. We call the combined duration of both phases the *attack time*.

6.4.3.2 Prototype and Test Setup

We implemented our prototype including the functionality required for SEVered based on KVM. To start and stop page tracking and change mappings, we extended the KVM API with additional calls, in particular, with *KVM system ioctls* [Thea]. This allows us to launch the attack from user space by communicating with the KVM kernel module. For page access tracking in KVM we used the technique from [Gua; Mor18]. While tracking is active, we record all tracked pages in a list in kernel memory. Upon the call to stop tracking, KVM returns the list of tracked pages to user space.

We ran KVM on Debian with a page size of 4 KB using an SEV-enabled Linux kernel in version `4.18.13` and QEMU `3.0.50`. We used an AMD EPYC 7251 processor with full support for SEV. We created a victim VM with 2 GB of memory and one of the four available CPU cores. We deployed Apache `2.4.25-3` and nginx `1.10.3-1` for the TLS key scenarios, and OpenSSH `7.4` for the SSH scenario in the VM. The FDE scenario is independent of a service, because the FDE key is a kernel resource exclusively used by the OS. We deployed eleven different web resources on each web server. We used 4096-bit private keys for TLS and SSH and a 256-bit symmetric FDE key for storage encryption with AES-XTS. As target for memory extraction with SEVered, we used a page-sized web resource served by nginx.

To capture the handshake messages for the TLS and SSH key scenarios, we used `tcpdump` with `libpcap`, a library for network packet capturing. For TLS, we captured the *change cipher spec* packet the services send to conclude a TLS handshake (filter `tcp[37] == 0x04`). For SSH, we captured the *new keys* message, which concludes the SSH handshake (filter `tcp[37] == 0x15`). We patched `libpcap` to execute a system call for $Tracking_{Start}$ the moment packet capturing begins, and a call for $Tracking_{End}$ the moment the filtered packet is captured. This tight interconnection minimizes the reaction time. To monitor disk I/O events of the VM in the FDE key scenario, we used the tool `inotifywait` to observe *inotify* events. In particular, the *notify* option allows to detect disk writes on the VM's disk image file. We modified `inotifywait` to issue the calls for $Tracking_{Start}$ right before starting to watch events and for $Tracking_{End}$ as soon as an inotify event is identified.

In real-world scenarios, a tenant's VM can show higher or less activity depending on the load caused by its clients. To simulate this behavior, we executed all our tests based on a *load model* with various *load levels*, representing low to high load. In our model, a load level of nine, for instance, refers to nine requests per second to the VM. We randomly alternate between the services for each request. With a probability of $\frac{300}{301}$, we make a request to one of the resources offered by one of the two web servers with equal probability. With a probability of $\frac{1}{301}$, we initiate an SSH login with a user remaining logged in for two minutes. Compared to the number of web server requests, we execute only few SSH logins, as these usually happen less frequently than requests to a web server. The average duration of the observation phase thus lies in the range of a few seconds to a few hundreds of milliseconds for the web servers and in the range of a few minutes to tens of seconds for

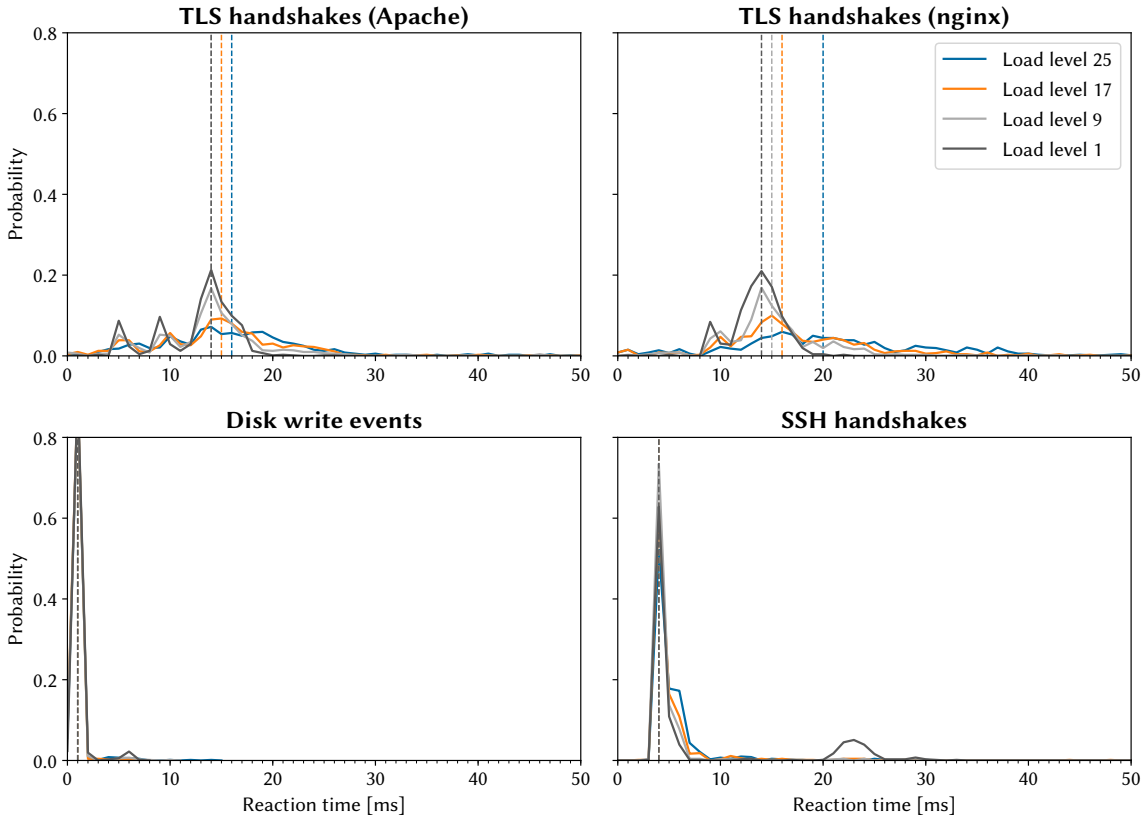


Figure 6.3: Distribution of the reaction times for all scenarios and load levels. The X-axes show discretized time steps of one millisecond while the Y-axes are normalized to one.

SSH. Note that `sshd` forks a new process for each new SSH connection. When the session terminates, the process exits and purges its SSH key. This means that the search time must be less than two minutes to extract the SSH key before the forked process exits.

We conducted 2,000 independent iterations of our attack for each of the four scenarios on four load levels: level *1*, *9*, *17*, and *25*. We started our attacks at random points in time while the VM processed requests according to the specific load level of our model. As an initial preprocessing step before the search phase, we filtered all execute-accesses. In our scenarios, all secrets are data structures located on non-executable memory pages.

6.4.3.3 Success Probability and Reaction Time

In this part, we investigate the success probability and reaction times. The four diagrams in Figure 6.3 illustrate the distribution of measured reaction times for each scenario. The four graphs in each diagram represent the four load levels. The X-axes are discretized in steps of one millisecond, and the Y-axes are normalized to one. The vertical dashed lines show the median reaction times over all repetitions for each level, providing an upper bound on the median critical window size.

The results for Apache and nginx TLS handshakes are depicted in the top row of Figure 6.3. Both diagrams show a clear peak for the two lower load levels, indicating a reliable reaction

time when the VM is not under high load. For the lowest load level, we can even observe that the reaction time never exceeded 21 milliseconds with Apache and 22 milliseconds with nginx. For higher load levels, more concurrent activities are executed by the VM, and the measurements are more dispersed over time. Consequently, it becomes likely that more pages have to be extracted in the search phase until the secret is found. This led to a maximum reaction time of around 50 milliseconds for both nginx and Apache in rare cases. However, the median reaction time increased to about only 20 milliseconds for nginx, and to about 16 milliseconds for Apache. As the reaction time is an upper bound for the critical window, the latter is smaller than tens of milliseconds for both Apache and nginx. We achieved a very high success rate of around 99.99% for both web servers on all load levels, meaning that we started tracking inside the critical window only in a few cases. The high success rate indicates that the upper bound we measured is a very conservative estimate. This comes from the fact that our prototype requires some time to actually stop tracking and (especially for the TLS scenarios) to recognize $Activity_{End}$. Note that if $Tracking_{Start}$ occurs inside the critical window of a TLS handshake, we still have the chance to observe Use_i of other handshakes being concurrently processed on higher load levels where lots of handshakes are made each second. The critical window can thus be even smaller on higher load levels.

The bottom left diagram in Figure 6.3 for disk write events shows that our implementation achieved an extremely fast reaction time of one millisecond in the median for each load level. Only in a few cases, we encountered a slightly higher reaction time. In contrast to the TLS key scenarios, the behavior was generally independent of the load level. In the TLS key scenarios, the network packets must first be sent by the VM to the network interface, on which the HV executes more time-consuming network packet capturing. The interception of disk write events is less complex and introduces less delay. The success rate for FDE key extraction was about 99.99%, indicating a very small critical window, as confirmed by the upper bound in the graph.

The bottom right diagram in Figure 6.3 shows that the reaction time for SSH handshakes was four milliseconds in the median and mostly independent of the load level. We encountered only a few samples going up to about 30 milliseconds. As for the TLS scenario, this indicates a small upper bound on the critical window and a possibly quick extraction. Accordingly, our attack had a success rate of 99.98%.

6.4.3.4 Attack Time

This part investigates the attack times for each scenario. Table 6.1 summarizes the relevant statistics for the median number of pages to be extracted and the median attack time for every scenario and load level. For both Apache and nginx, the median number of pages to be extracted until finding the TLS key increased between low and high load levels. We measured an increase of the median from 102 to 301 pages (i.e., 408 to 1,204 KB of memory) for nginx, and from 128 to 171 pages (i.e., 512 to 684 KB of memory) for Apache. Additionally, the Median Absolute Deviation (MAD) increased from 5 to 160 from low to high load for nginx, respectively from 21 to 109 for Apache. The median number of extracted pages was particularly small compared to the median number of total tracked pages, which was for both cases between 1,691 and 2,085 (not listed in Table 6.1). The

Table 6.1: Statistics for the median length of the observation and search phases, and for the median number of extracted pages with the median absolute deviation for the different scenarios and load levels.

Use Case	Load Level	Median Page No	MAD Page No	Median Time	
				Observ.	Search
TLS (nginx)	1	102	5	1.46s	17.72s
	9	116	19	0.37s	15.48s
	17	165	69	0.32s	18.61s
	25	301	160	0.31s	32.71s
TLS (Apache)	1	128	21	1.42s	21.90s
	9	137	40	0.37s	17.95s
	17	154	80	0.33s	17.44s
	25	171	109	0.32s	18.65s
FDE	1	70	8	2.43s	12.24s
	9	71	9	2.15s	9.34s
	17	70	8	2.08s	7.84s
	25	69	9	2.04s	7.37s
SSH	1	7	1	193.36s	1.33s
	9	7	1	27.23s	0.97s
	17	7	1	16.19s	0.83s
	25	7	1	14.41s	0.80s

median duration of the search phase was between about 15.5 and 32.7 seconds for nginx, and between about 17.5 and 22 seconds for Apache. We measured an average extraction time of around 123 milliseconds for a single page with our SEVered implementation and setup. We measured this time to fluctuate quite frequently in the scale of a few tens of milliseconds. This is why a higher median number of extracted pages did not affect the duration of the search phase in a clearly linear way. The on the fly-analysis for a single memory page took about 50 milliseconds. This means that the page extraction performance is the limiting factor of our attack. The higher the load, the less time we required for the observation phase, which ranged from 1.46 to 0.31 seconds in case of nginx, for instance. This is because the probability of quickly observing $Activity_{End}$ increases with a high frequency of requests. To summarize, we measured an attack time between about 16 and 33 seconds in the median for the web services.

For the FDE key scenario, the amount of pages that had to be extracted was very small and mostly independent of the load level. Accordingly, the median number of extracted pages was between 69 and 71 for the different load levels (i.e., 276 to 284 KB of memory) As in the TLS key scenario, this number is small compared to the median number of total tracked pages, which was between 2,526 and 3,433. The overall duration of the search

phase was between about 7.4 and 12.3 seconds. The on the fly analysis for a single memory page took only about 2 milliseconds on average. We mostly identified the key as part of the AES key schedule, and only occasionally by the kernel data structure, see Section 6.4.2. We measured a slightly decreasing observation time from 2.43 to 2.04 seconds. This indicates that the VM's OS is regularly writing pages to disk, in our case mostly regardless of the load level. In sum, the attack time was between less than 9.5 and 14.7 seconds in the median.

In the SSH scenario, we merely had to extract seven pages in the median with a MAD of one. This is a particularly small number, especially compared to the median number of 10,102 to 11,094 total tracked pages, omitted from Table 6.1. We measured a median duration of the search phase of about 0.80 to 1.33 seconds. This means that the attack works reliably assuming that the SSH connection lasts at least 1.33 seconds. Similar to the TLS key scenario, the on-the-fly analysis of a memory page took about 50 milliseconds. With our load model, the observation time of about 14 to 194 seconds was comparably high for the SSH scenario. This is another reason why the number of extracted pages was especially low for the SSH case. In long observation phases, we already tracked a high number of pages before Use_n, making it very unlikely that many pages are tracked within the reaction time frame at the end of the activity.

6.4.3.5 Optimization with Preprocessing

As discussed in Section 6.4.1, preprocessing with prioritization and filtering is an optional optimization before the search phase. Preprocessing usually requires a priori knowledge about the use case and behavior of the VM, which may not always be available. This behavior may also vary between different hard- and software configurations. For our evaluation, we already used the knowledge that all secrets are data structures located on non-executable memory pages. This allowed us to filter execute-accesses from the list of tracked pages. The amount of pages to be extracted was thereby reduced by about 22% on average over all samples.

Use_n was a read-access in 96% of our attacks for the TLS handshakes and in 93% a write-access for the SSH handshakes. For disk write events, Use_n was always a read-access. Whether the page containing the secret is tracked as read- or write-access depends on the other data located within the page. The type of access thus cannot be predetermined with certainty. Filtering of write-accesses could significantly reduce the attack time, but could also reduce the success probability. Also, prioritizing the extraction of read-accesses over write-accesses in the list would boost the attack in most cases, but could also introduce costly outliers.

Another possibility for prioritization is knowledge about the reaction time, as shown in Figure 6.3. The graphs for the two web server scenarios show that the reaction time was rarely less than eight milliseconds before $Tracking_{End}$. Re-arranging these early accesses further back in the list of tracked pages can thus reduce the amount of pages to be extracted until the secret is found. The same observation can be made for the SSH scenario, where $\mathrm{Use_n}$ never happened less than three milliseconds before $Tracking_{End}$. However, in this case the number of pages to be extracted is already so small that further optimization may not be required.

The reaction times in Figure 6.3 can also help to determine a good criterion for restarting the attack when a secret has not been found after a certain number of extracted pages. For instance, page accesses tracked later than 30 milliseconds before $Tracking_{End}$ are likely exceeding the reaction time frame and thus unlikely to be a candidate for Use_n. This can be used as a criterion to detect that $Tracking_{Start}$ was inside the critical window and to restart the attack from the observation phase.

6.4.3.6 Summary

For all evaluated scenarios, both performance indicators are very promising. We found that the critical window was very small in all cases. $Tracking_{Start}$ was thus inside the critical window only in a few cases, resulting in a very high success probability throughout all scenarios and load levels. The most important factor for the attack time, the duration of the search phase, was also very small. Extracting all memory from our VM with 2 GB of main memory would take more than 7 hours with SEVered [Mor18]. Assuming that a key can reside not only on one but on several pages, the naive extraction would require several hours on average to find the key. Our approach can extract secrets faster by several orders of magnitude.

In cases when $Tracking_{Start}$ is inside the critical window, the attack fails and we extract all tracked pages without finding the secret. In such cases, we have to repeat both the observation and search phase. To avoid a lengthy extraction of all tracked pages in unsuccessful attempts, the search can be canceled early when the likelihood of finding the secret drops, according to our evaluated distribution. For the following search phase, all pages extracted in the previous attempts can then be excluded from extraction given that the secret does not change its location. In sum, our results have shown that our prototype is able to quickly and reliably extract different sensitive secrets and performs well even under high load.

6.4.4 Discussion

In the following, we discuss further important aspects of our attack:

Overhead

The overhead caused by the tracking itself is limited, because each accessed page only triggers a SLAT page fault once. We neither detected perceivable effects like delays in response times in the HV nor inside the VM. The host system and VM remained stable even on the highest load level. We measured only a small additional delay of web and SSH server responses when tracking was active.

Low Memory

When the VM is low on memory, its kernel might try to free memory by swapping out pages, by unmapping file-backed pages, or by killing processes. A page containing the secret might then be re-used by another process or by the kernel during our attack. In such a case, we are still able to extract the memory contents of the page, but its contents might have already been overwritten. We did not encounter such cases in our tests.

Triggering Activities

In our concept, we start tracking at an undefined point in time and do not actively trigger activities to interfere as little as possible with the VM's normal operation. Our concept worked well in our evaluated scenarios, because we extracted frequently used secrets. In the SSH scenario, however, the key may be used rather infrequently. This is, for instance, the case when an administrator logs in to a web server for maintenance only from time to time. When a secret is rarely used but the attacker requires the observation phase to be as short as possible, the attacker can consider the active triggering of an activity. In the SSH scenario, an attacker can actively start a login procedure without a user account. SSH servers use their key for server authentication and wait for the user to authenticate with a default timeout of two minutes. An attacker can thus initiate a login and extract the SSH key before the session timeout without waiting for a legitimate user to login. Note that active triggering might increase the probability of the attack being detected and might not always be possible.

Portability

We expect that our approach can be transferred to other scenarios and configurations than evaluated. Our approach does not depend on specific service or library versions. Furthermore, our approach is not tied to a specific SEV processor and mostly independent of the VM's performance and OS. Our approach can also be leveraged to extract other types of memory, such as confidential code, documents or images. The performance of our approach can differ on systems with other hard- and software configurations. However, we expect the performance to vary only slightly assuming that $Tracking_{End}$ can be observed quickly. We ran several tests in which we assigned our VM more memory, multiple cores and in which we configured the web servers to utilize a high number of worker processes. The performance indicators remained coherent with our evaluation results in all runs.

6.5 Countermeasures

A countermeasure against memory extraction from encrypted VMs is to prevent the SEVered attack. While several software-based countermeasures against SEVered could be deployed [Mor18], these cannot reliably protect from SEVered. This implies that only a modification of AMD SEV itself can prevent attacks exploiting the missing integrity protection. An effective solution would be to complement AMD SEV with a full-featured guest memory page integrity and freshness hardware protection comparable to Intel SGX. This works for Intel SGX, because the computation model is designed to protect the memory of enclaves where comparably small amounts of privileged code are executed on demand, compared to full-blown VMs on AMD SEV platforms. For AMD SEV, a relatively high silicon cost would have to be expected to realize a similar protection. A low-complexity yet efficient modification to prevent the remapping could be to combine the hash of a memory page's content with the guest-assigned GPA and a nonce into the en-/decryption. The nonce ensures the freshness preventing old pages for the same GPA to be replayed into guest memory. The hash and GPA combination bind the page content to the actual mapping preventing the remapping with other pages.

A further aspect is that our attack relies on targeted secrets to remain at their memory

location during our attack. Purging secrets in memory after use would cause the search phase to fail. We found TLS and FDE keys to always remain at their memory location in our tests. However, in case of SSH keys, the processes forked by the SSH daemon for initiating new SSH connections purge their private key when a session terminates and then exit. This means that an SSH session must remain open until the secret is extracted. This, for instance, requires a user to remain logged in or a login attempt to remain pending over the time of the search phase, which is less than 1.5 seconds in our case.

Systematically purging all sorts of secrets from main memory after use would require adapting existing software. For some secrets, purging might not be feasible. An example is the FDE key, which is constantly required for disk I/O. A more promising solution is to relocate the most valuable secrets from main memory to dedicated hardware. Since SEVered can only extract contents from main memory, storing secrets in hardware would prevent them from being extracted by a malicious HV. This can, for example, be realized using HSMs. Additionally, hardware-based disk encryption can be used to protect the FDE key.

More generally, attacks like SEVered exploit side-channels that are very hard to close. SEVered exploits the page-fault side-channel, which exists for the HV to manage guest memory. [Li19] uses I/O side-channels, whereas [Wer19] leverages the general purpose registers and the performance measuring subsystem, i.e., IBS to deduct information from guest VMs. SEV-ES closes the general purpose register side-channel by encrypting the VMCB where general purpose registers are stored upon a VM exit. Making SEVered harder to apply could also be achieved by eliminating the page offset and access type information upon page fault, for instance, which is information the HV not necessarily requires. Future SEV generations become more resilient against attacks when further reducing side-channels. A possible step would be to disable features providing side-channels such as IBS.

To prevent memory moving attacks, a design with authenticated encryption [Li19] is required, or the introduction of stronger tweak functions that may vary between system starts as a mitigation. This is a point that AMD plans to address in future versions [Li19]. Defeating the I/O side-channel with a practical solution is a still unsolved problem. A possible solution was raised in [Li19] where the HV would never be able to observe unencrypted I/O data transitioning into and out of the HV. In a nutshell, the solution introduces a trusted third party as an I/O proxy and modifies SEV's I/O model. Using the SEV APIs, a shared key between the proxy and VM, which the HV cannot learn, is negotiated for I/O data encryption. This key can be used to securely pass I/O data between VM and proxy, more particularly between SP and proxy, where the keys are stored. For the example of network I/O, the HV would no longer be able to read headers of in- or outgoing network packets, as these would be encrypted with the key only known to proxy and VM. The HV is responsible for I/O packet forwarding and requesting their mapping into the VM using the SP for en-/decryption. This approach, however, comes with extremely high costs in performance and applicability.

6.6 Summary

In this chapter on Challenge 5, we first presented our framework for main memory extraction on SEV-encrypted VMs as part of Contribution 10. Our results demonstrate that attacks

leveraging our framework, which we called SEVered, are feasible in real scenarios and that the full memory of an encrypted VM can be extracted in reasonable time. Important aspects are the stickiness of the target resource's mapping in the target VM and the reliability with which the resource can be identified by the HV. As long as the resource mapping remains constant, the resource can be repeatedly requested from outside and swapped with the VM's other memory contents. The goal can either be the extraction of the full VM's contents, or when specific secrets like a TLS key are targeted, the extraction of memory until the desired resource is found. Our results demonstrate that SEVered reliably works with different services even when the targeted VM is under high load from other remote peers. Higher load complicates the accurate identification of the GPAs representing the resource in the VM, taking SEVered more time for its identification phase. SEVered does not rely on a specific service or resource, only on a resource the service offers to the outside and keeps constantly mapped in memory throughout the extraction. The current prototype is capable of extracting arbitrary memory of the targeted VMs at a scale of tens of kilobytes per second, but the design allows for a higher scale depending on the size of the resource used as basis for the attack and targeted service's throughput. However, SEVered is limited to extracting the memory of a single target VM. Requesting another VM's memory would expose scrambled data only, because the SP uses different, independent keys for each VM. This means that for extracting memory from different VMs, SEVered needs to be ran against each VM individually.

In the next step, we extended our framework with Contribution 11 with a method for the efficient, targeted extraction of secrets from SEV-encrypted VMs. Compared to time-consuming, bare memory extraction with SEVered, our two-phased method for the efficient extraction of secrets unobtrusively and quickly exfiltrates secrets with a high success probability. In the first phase, we track the page accesses of an encrypted VM until detecting an event indicating that the VM recently accessed the specific secret. In the second phase, we leverage SEVered to systematically retrieve the tracked pages and simultaneously analyze their contents to quickly identify the secret. We presented various use cases for highly sensitive secrets commonly found in VMs in cloud scenarios. We performed an evaluation for these cases on a fully SEV-enabled EPYC processor with varying levels of load, usually caused by independent clients not involved in the attack. Our results show that we are able to extract TLS keys after a handshake in less than 15.5 seconds in the median on lower load levels and in no more than about 32.7 seconds in the median on our highest evaluated load level. The extraction of the FDE key after a disk write event took between less than 7.4 seconds and 12.3 seconds in the median. The extraction phase for SSH keys after an SSH handshake took about 0.8 to 1.35 seconds in the median. We expect that our approach can be used for the extraction of further types of secrets, which we are going to investigate in future work.

By transparently encrypting main memory leveraging hardware separated from the main application processor, AMD SEV raises the bar for attackers targeting confidential VM data. With our framework, we have demonstrated that extraction is nevertheless possible and that it is thus especially hard to defend against privileged attackers targeting confidential data - even with hardware extensions for main memory encryption in place. We discussed several approaches for countermeasures in Section 6.5.

CHAPTER 7

Conclusion and Future Work

In the following, we first discuss the outcomes of our work elaborated in the previous chapters. We structure this discussion along the challenges presented in the problem statement in Section 1.2 and refer to our contributions. Towards the end of this section, we discuss future work referring to both our proposed security architectures and our main memory extraction frameworks.

Challenge 1: Design of Architectures for the Secure Isolation of System Resources
We started with the design of a secure virtualization architecture for the isolation of system resources in Chapter 3 (Contribution 1). We based the architecture on OS-level virtualization to isolate resources at container boundaries [Hub16a]. This thwarts runtime attacks of local and remote attackers aiming to gain privileges to access confidential data in memory. We realized the architecture for mobile devices, enabling to run multiple Android containers in parallel on a single device (Contribution 2). We showed that it is practicable to operate, for instance, a business container with a special set of applications processing confidential data together with an unconstrained container for private use. While the user was allowed to run any application in the untrusted private container, secrets in the business container remained secure despite a possibly compromised private container. The hardened kernel and strong isolation between the containers lay the foundation for data confidentiality during runtime, making it hard to break the isolation. FDE for container storage using an SE for protecting the encryption key secures persistent data from being read out by physical attackers. The performance evaluation on smartphones showed that the architecture is an efficient means providing users with almost native experience. Our design for the mobile domain included the secure virtualization of hardware devices for containers, as well as a secure switching mechanism between untrusted containers, making the architecture suitable for use in practice.

In Chapter 3, we demonstrated that the architecture we developed is not limited to the mobile domain, but can be transformed to comply with other platforms and with other use cases (Contribution 4). Based on the former architecture, we therefore proposed a secure virtualization architecture for the embedded domain. We developed a mostly platform-independent prototype and applied it in a concrete IoT use case [Bro18]. This use case, the IDS, is a data exchange platform enabling transfer and trade across the boundaries of organizations. The core components of this distributed network are the trusted connectors gathering, processing and sharing data with other, possibly unknown connectors. The challenge for such an environment was to lay foundations for data confidentiality within and between connectors. In contrast to mobile devices, there is

usually no end user interaction. Further, less hardware device virtualization is required to operate the containers. Containers mostly used the network interface to gather data from sensors. Inside each container, one specific service operated isolated from other services, where each service, possibly originating from third-parties, is constrained by our architecture to only access data from strictly defined resources, such as from a specific sensor, or cyber-physical device. The TPM we employed instead of an SE for protecting the persistent storage suits to this scenario, binding storage decryption on container start to a known-good system state. Furthermore, we used the TPM for enabling remote attestation to control the exchange of confidential data between connectors. Data is only transferred to known-good connectors ensuring data confidentiality within the ecosystem.

Challenge 2: Applicability of Architectures for Secure Resource Isolation in Operational Environments

Decisive for the applicability of the secure virtualization architecture in productive environments are not only aspects like the achieved device-centric security, performance and usability, but also to design a surrounding environment and processes, i.e., an ecosystem, to securely operate devices running the architecture. Our security concept for the trust ecosystem along with the IDS scenario in Chapter 3 referred to a decentralized, heavily interacting embedded device use case [Bro18] (Contribution 4). Part of this was the secure enrollment and provisioning, as well as the establishment of secure communication channels and of data usage control rules between devices. This included a holistic security concept for the whole ecosystem. An important aspect was ensuring the trustworthiness of service and lower layer implementations running on the devices. Therefore, we introduced certain authorities to the ecosystem, which are allowed to sign services and software updates for the architecture based on a PKI. The result is that devices accept only signed services and updates from approved parties after successful verification.

As further part of Chapter 3, we presented the basis for these concepts tailored to use cases in the mobile domain with different user identities [Wes15] (Contribution 3). This entailed identity management including a process for associating end users with an SE, with a device and with containers. This also allowed to transfer containers between different devices using our remote management backend. With secure enrollment and provisioning, the ecosystem design made devices running the virtualization architecture ready for use in corporate environments where the protection of confidential data plays an important role.

Challenge 3: Main Memory Extraction on Conventional Platforms

Our memory extraction framework based on the cold boot attack in Chapter 4 demonstrated the potential of memory attacks on mobile devices [Hub16b] (Contribution 5). We showed that it is possible to directly extract the main memory of mobile devices without acquiring privileges on the system during its runtime. Our method preserved the full contents of main memory of the running system after a cold boot. Even after we triggered our minimal application for data requests and transfer to a connected host system, the mobile device's previous memory state was almost completely sustained. The minimal application overwrites no more than two pages of memory at locations, which contain no confidential, but static, publicly known data. As the entire kernel and user space memory was accessible,

we presented a method for the systematic extraction of memory contents for forensic data acquisition. Amongst others, we showed an example where we read out the account credentials of an exchange application, or the last network connections made on the device, in very short time. Another result was that our proposed virtualization architecture is incapable of protecting against this type of memory attack, and in particular against physical attackers. We showed this by extracting container FDE keys from devices running the virtualization platform. This motivated the urge for protection mechanisms against physical attackers, and thus the design of architectures for main memory encryption.

Challenge 4: Design of Architectures for Main Memory Encryption
To defend systems from physical attackers, for instance, acting via DMA or cold boot vectors, we introduced architectures for the encryption of main memory. Encryption of main memory does not prevent the attack vectors themselves, but mitigates their effect, because the attacker can only read encrypted main memory contents.

We started with the protection of common x86 devices in Chapter 5 (Contribution 6). We incorporated main memory encryption into the suspend and resume functionality of the OS, making it also possible to apply the encryption to VMs running on a HV [Hub17b]. While we sustained the common user experience, we also achieved a high degree of performance, measured on a mid-performance 12 GB notebook where en- and decryption took about 800 milliseconds on average. The encryption of main memory during the suspension of a device protects in cases where a device's user is absent, either because the device is left unattended or gets stolen. At that time, the system can not reconstruct the main memory contents and can not get back to normal operation unless the user returns to unlock the system. Memory attacks can only yield encrypted main memory contents.

The presented approach is only applicable on platforms with full suspension features, like notebooks with common OSs. On smartphones, there is no full suspension, because the device must be able to wake up autonomously and process incoming data, such as phone calls, or messages originating from the network. In Chapter 5, we thus advanced with a memory encryption architecture for process groups, i.e., for specific parts of a system. We combined the memory encryption architecture with our previously presented secure virtualization architecture for mobile devices [Hub18] (Contribution 7 and Contribution 8). We encrypted the containers that are not actively used and thus in background. When the device is not used for a certain period of time, all of a user's containers are ensured to be switched to background. We kept the design of the memory encryption architecture independent from the virtualization architecture by constructing the encryption based on generic process groups instead of tailoring it directly to Android containers. This makes it possible to use the encryption architecture in other scenarios.

The goal for the virtualization architecture combined with memory encryption was to design a system resilient against both remote attackers and against direct attacks on memory. The existing infrastructure made it possible to adapt the virtualization architecture such that the device can still process background events while containers are encrypted and to use the SE for decryption to protect the memory encryption keys. The management container not comprising confidential user data always remains unencrypted and handles the background events for encrypted containers. The management container

can thereby notify the user of events, such as incoming data and calls, for background containers. With an average total wake up and decryption time of about 2.5 seconds for an encrypted container, the approach preserves a high degree of usability when selecting only the container with the confidential information for encryption.

Later in Chapter 5, we presented an architecture for runtime memory encryption, for instance, for systems where suspension is not intended, or where the full functionality of the device needs to be preserved at all times (Contribution 9). We introduced a minimal HV running a single guest VM [Hor17]. The HV transparently en- and decrypts the RAM contents in a TEE, the ARM TrustZone on our platform. While a small, varying part of the main memory is kept unencrypted at all times, most of the memory is encrypted with a key only present in the TEE. The approach thus makes a trade-off in terms of security and performance impact. The larger the set of unencrypted memory, the less memory needs to be en- and decrypted during the VM's runtime. We determined a pivotal point achieving a high level of security while only sacrificing a moderate share of the VM's performance in benchmarks.

One outcome of our proposed architectures was that attackers gaining privileges on the target system are hard to defend against. In case of our suspension-based architectures, the attacker would be unable to read plaintext memory contents given that evil maid attacks are not possible, e.g., an attacker deploying a backdoor in the absence of the user. However, for our runtime memory encryption approach, an attacker gaining privileges on the system can directly read memory contents as the system remains fully operational. We further investigated this topic at the example of hardware-based memory encryption architectures in the subsequent part of the thesis.

Challenge 5: Main Memory Extraction on Platforms with Hardware-Based Memory Encryption

Runtime memory encryption has the advantage that sensitive memory is likely to be encrypted at all times. However, because the system is always fully operational and responsive, an attacker could gain privileges and force the system to a state where it presents parts of the main memory in plaintext, or wait for certain parts of memory to be decrypted, even without knowledge of the encryption key. The defense against a powerful attacker on runtime encryption systems, possibly gaining supervisor privileges, is thus an especially difficult topic not only in the case of our proposed architectures.

We emphasized this in Chapter 6 on the example of AMD's SEV hardware extension, which transparently encrypts main memory of VMs on the basis of a dedicated coprocessor, the AMD SP, using different, ephemeral keys for each VM. The technology strives to keep the main memory of a system's VMs confidential even when an attacker with control over the underlying HV is present. The SEV technology is designed for server systems hosting VMs of remote customers. These VMs in turn usually provide a service to the outside, which public peers, for example, in case of a web server, can access. We observed that SEV misses to protect the integrity of encrypted pages. This makes it possible to modify the VM's mappings to physical pages with the HV (Contribution 10), or to move physical memory. The memory extraction framework we constructed [Mor18] made use of this observation and modified the memory mappings of different services, such as web servers,

to respond us with arbitrary memory contents of the VM. After tracking the page access patterns of a VM upon several requests to a service's resource, we were able to pin down the requested resource in the VM's memory, such as a website. We also demonstrated that our framework performs under real conditions where a web server is under high load, frequently accessed by multiple clients. For instance, when 50 additional requests per second from other peers to Apache and nginx web servers were made, we sustained a low number of repetitions for page access pattern tracking until we reliably identified the target resource in memory. We then modified the memory mapping for this resource pointing to another region of the VM's memory. By repeatedly accessing the resource and re-mapping the memory, we were able to dump the full memory of a VM across a number different services, also while other peers accessed the server simultaneously.

Since the extraction of a VM's full memory contents can leave a large footprint in case of many repeated requests and re-mappings, we extended the memory extraction framework with a method for the specific extraction of targeted resources, leaving a minimal footprint only [Mor19] (Contribution 11). We demonstrated the capability of the method at the example of kernel and user space secrets used by different services. We extracted the secrets by systematically reducing the number of page candidates in which they can be contained. For example, we extracted the TLS and SSH server keys of the nginx, Apache and OpenSSH servers, respectively, on the basis of observing only a single handshake. We also extracted FDE keys based on observing the VM's disk write activities. One of the pages accessed during the handshake or disk write activity inevitably contains the targeted key. Analyzing the VM's page access timings and types, we were able to reduce the amount of candidate pages to a minimum. This resulted in an extraction phase with only a minimal amount of repeated requests and re-mappings, and with extraction times of way less than a minute.

Future Work

The relevance of the protection of confidential data pursued in this work is reflected by past and current developments made by major hardware manufacturers. In terms of isolation, developments were on the one hand the introduction of hardware support to relocate sensitive execution contexts of processes in separate hardware locations. This protects data and execution from possibly malicious, privileged entities, for example, using the ARM TrustZone, Intel SGX, or isolated security coprocessors. On the other hand, there is the introduction and further development of TPM and SE applications. These hardware building blocks were amongst others designed to provide secure storage locations in the presence of physical attackers. The introduction of AMD SME/SEV was motivated by the fact that physical attacks on main memory, or privileged entities on the system circumventing isolation mechanisms, are a real threat. Recently, Intel announced the introduction of Multi-Key (MK)/TME [Int], a hardware extension for transparent runtime main memory encryption, as another example for the relevance of the presented topics.

Our work on the analysis of the access patterns of encrypted VMs can be continued to learn more about the execution flow of encrypted VMs, for example, to ultimately gain code execution on the VM. Our framework can also be developed towards locating the VM kernel with its modules in physical memory, which can at least help defeat the

random Kernel Address Space Layout Randomization (KASLR) physical kernel memory shift. Future topics in the field of encrypted VMs are to critically revisit the design and implementation of hardware-backed memory encryption mechanisms for possibly required improvements. For the Intel MK/TME platform, an open point is, for instance, to determine its security properties and to investigate with which means attackers can acquire memory protected by MK/TME. Another possible problem of hardware security extensions are the interfaces between the untrusted system and the isolated extension's runtime environment. These interfaces need to be properly defined and implemented, preventing attackers from exploiting trusted environments via these interfaces. This has, for example, been shown for the TrustZone interface [Ros14], or recently for AMD's SP [Good], where the SEV keys are stored.

Another path for future work is leveraging hardware extensions to enable the design and implementation of novel applications and systems to solidly protect secrets and execution flows. An example can be to construct a secure platform for embedded systems, which may be based on a custom chip and board architecture providing a strong security infrastructure, for instance, for connectors in the IDS or for critical infrastructures. The platform concepts can be leveraged to design and implement building blocks to improve trust in diverse systems, such as with secure boot, runtime integrity verification, secure software updates, use of secure execution environments, or remote attestation capabilities. Specific use cases, such as IoT applications, can then be flexibly built on top of the secure platform offering a flexible security infrastructure. Examples for ongoing work are Azure Sphere [Hun; Micb] and the Android Things [Goob] platform.

Potential future hardened platforms can also make use of our secure virtualization architecture. Its further development and integration into other use cases than presented is a promising step leading towards a full-fledged implementation for its later use as a product. Regarding the realization for smartphones, the virtualization architecture can also be further developed to comply with more recent mobile kernel versions and to support more recent hardware devices. Only when working on recent hardware and offering containers with recent OS versions, the architecture can find acceptance from end users. The proposed concepts for the provisioning, enrollment and lifecycle of the platform in untrusted environments like the IDS are suitable for being adopted to further productive use cases in the future.

The design of hardened platforms should also specifically address the threat of cold boot and DMA attacks. An open topic is to investigate the vulnerability of further systems regarding the cold boot attack to emphasize the need to resolve possible design weaknesses. In this case, the task is to at least employ standard methods against cold boot attacks, such as clearing the main memory on system start, or making the memory inaccessible to attackers. For memory attacks in general, our proposed memory encryption architectures can be further developed and employed in practice. In general, our proposed architectures can all be leveraged as building blocks to protect various systems and deployments. Further, our proposed architectures can be combined with hardware extensions in order to not only achieve confidentiality of main memory, but also to enforce its integrity regarding attackers not only reading, but also modifying memory contents to manipulate a system.

Bibliography

[Adva] Advanced Micro Devices: *AMD-V Nested Paging*. http://developer.amd.com/wordpress/media/2012/10/NPT-WP-1%201-final-TM.pdf. *Last access: September 4, 2019* (cit. on p. 139).

[Advb] Advanced Micro Devices: *AMDESE/AMDSEV: AMD Secure Encrypted Virtualization*. https://github.com/AMDESE/AMDSEV. *Last access: September 4, 2019* (cit. on p. 143).

[Advc] Advanced Micro Devices: *Secure Encrypted Virtualization API Version 0.16*. http://developer.amd.com/wordpress/media/2017/11/55766_SEV-KM-API_Specification.pdf. *Last access: September 4, 2019* (cit. on pp. 135, 139, 148).

[Aga15] Agarwal, Ritu and Suvarna Kothari: 'Review of Digital Forensic Investigation Frameworks'. *Information Science and Applications*. Vol. 339. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015: pp. 561–571. isbn: 978-3-662-46577-6 (cit. on pp. 74, 76).

[Alm14] Almohri, Hussain M.J., Danfeng (Daphne) Yao, and Dennis Kafura: 'DroidBarrier: Know What is Executing on Your Android'. *Proceedings of the 4th ACM Conference on Data and Application Security and Privacy*. CODASPY '14. San Antonio, Texas, USA: ACM, 2014: pp. 257–264. isbn: 978-1-4503-2278-2. doi: 10.1145/2557547.2557571. url: http://doi.acm.org/10.1145/2557547.2557571 (cit. on pp. 1, 26, 30).

[And] Android: *Security-Enhanced Linux in Android*. https://source.android.com/security/selinux. *Last access: September 4, 2019* (cit. on p. 29).

[And11] Andrus, Jeremy, Christoffer Dall, Alexander Van't Hof, Oren Laadan, and Jason Nieh: 'Cells: A Virtual Mobile Smartphone Architecture'. *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. SOSP '11. Cascais, Portugal: ACM, 2011: pp. 173–187. isbn: 978-1-4503-0977-6. doi: 10.1145/2043556.2043574. url: http://doi.acm.org/10.1145/2043556.2043574 (cit. on pp. 28–30, 38, 44).

[AnT] AnTuTu Developers: *AnTuTu Benchmark - Know Your Android Better*. http://www.antutu.com/en/index.shtml. *Last access: September 4, 2019* (cit. on p. 132).

[Apo13] Apostolopoulos, Dimitris, Giannis Marinakis, Christoforos Ntantogian, and Christos Xenakis: 'Discovering Authentication Credentials in Volatile Memory of Android Mobile Devices'. *Collaborative, Trusted and Privacy-Aware e/m-Services*. Ed. by Douligeris, Christos, Nineta Polemi, Athanasios Karantjias, and Winfried Lamersdorf. Vol. 399. IFIP Advances in Information and Communication Technology. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013: pp. 178–185. isbn: 978-3-642-37437-1 (cit. on pp. 3, 76).

[Arb97] Arbaugh, W. A., D. J. Farber, and J. M. Smith: 'A Secure and Reliable Bootstrap Architecture'. *Proceedings of the 1997 IEEE Symposium on Security and Privacy*. SP '97. Washington, DC, USA: IEEE Computer Society, 1997: pp. 65–. url: http://dl.acm.org/citation.cfm?id=882493.884371 (cit. on p. 2).

[ARM09] ARM Limited: *ARM Security Technology - Building a Secure System using TrustZone Technology*. ARM Technical White Paper. http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf. *Last access: September 4, 2019*. 2009 (cit. on pp. 2, 93, 97).

[Ash16] Ashfield, Bruce: *Linux Containers - Where Enterprise Meets Embedded Operating Environments*. Tech. rep. http://events.windriver.com/wrcd01/wrcm/2016/10/WP-Pulsar-Linux-Containers.pdf. *Last access: September 4, 2019*. Wind River Systems, 2016 (cit. on p. 28).

[Bac14] Backes, Michael, Sebastian Gerling, Christian Hammer, Matteo Maffei, and Philipp Styp-Rekowsky: 'AppGuard – Fine-Grained Policy Enforcement for Untrusted Android Applications'. *Revised Selected Papers of the 8th International Workshop on Data Privacy Management and Autonomous Spontaneous Security - Volume 8247*. Berlin, Heidelberg: Springer-Verlag, 2014: pp. 213–231. isbn: 978-3-642-54567-2. doi: 10.1007/978-3-642-54568-9_14. url: http://dx.doi.org/10.1007/978-3-642-54568-9_14 (cit. on pp. 1, 26, 30).

[Bal] Balena: *Introduction – What is balenaOS?* https://www.balena.io/os/docs. *Last access: September 4, 2019* (cit. on p. 28).

[Bar03] Barham, Paul, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield: 'Xen and the Art of Virtualization'. *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*. SOSP '03. Bolton Landing, NY, USA: ACM, 2003: pp. 164–177. isbn: 1-58113-757-5. doi: 10.1145/945445.945462. url: http://doi.acm.org/10.1145/945445.945462 (cit. on pp. 26, 32).

[Bau16] Bauer, Johannes, Michael Gruhn, and Felix C. Freiling: 'Lest we forget: Cold-boot attacks on scrambled DDR3 memory'. *Digital Investigation* (2016), vol. 16. DFRWS 2016 Europe: S65–S74. issn: 1742-2876. doi: 10.1016/j.diin.

2016.01.009. URL: http://www.sciencedirect.com/science/article/pii/S1742287616300032 (cit. on p. 77).

[Bec05] Becher, Michael, Maximillian Dornseif, and Christian N Klein: *FireWire: All Your Memory Are Belong To Us*. Proceedings of CanSecWest. https://cansecwest.com/core05/2005-firewire-cansecwest.pdf. *Last access: September 4, 2019*. 2005 (cit. on pp. 3, 76, 106, 136).

[Bec11] Becher, Michael, Felix C. Freiling, Johannes Hoffmann, Thorsten Holz, Sebastian Uellenbeck, and Christopher Wolf: 'Mobile Security Catching Up? Revealing the Nuts and Bolts of the Security of Mobile Devices'. *Proceedings of the 2011 IEEE Symposium on Security and Privacy*. SP '11. Washington, DC, USA: IEEE Computer Society, 2011: pp. 96–111. ISBN: 978-0-7695-4402-1. DOI: 10.1109/SP.2011.29. URL: http://dx.doi.org/10.1109/SP.2011.29 (cit. on pp. 1, 30).

[Ber14] Bernstein, David: 'Containers and Cloud: From LXC to Docker to Kubernetes'. *IEEE Cloud Computing* (Sept. 2014), vol. 1(3): pp. 81–84. ISSN: 2325-6095. DOI: 10.1109/MCC.2014.51 (cit. on p. 27).

[Bil16] Bilzhause, Arne, Manuel Huber, Henrich C Pöhls, and Kai Samelin: 'Cryptographically Enforced Four-Eyes Principle'. *11th International Conference on Availability, Reliability and Security (ARES)*. IEEE, Aug. 2016: pp. 760–767. DOI: 10.1109/ARES.2016.28 (cit. on p. 13).

[Bla12] Blass, Erik-Oliver and William Robertson: 'TRESOR-HUNT: Attacking CPU-bound Encryption'. *Proceedings of the 28th Annual Computer Security Applications Conference*. ACSAC '12. Orlando, Florida, USA: ACM, 2012: pp. 71–78. ISBN: 978-1-4503-1312-4. DOI: 10.1145/2420950.2420961. URL: http://doi.acm.org/10.1145/2420950.2420961 (cit. on p. 76).

[Boi06] Boileau, Adam: *Hit by a Bus: Physical Access Attacks with Firewire*. Presentation, Ruxcon. 2006 (cit. on pp. 3, 106, 136).

[Bra08] Brakensiek, Jörg, Axel Dröge, Martin Botteck, Hermann Härtig, and Adam Lackorzynski: 'Virtualization As an Enabler for Security in Mobile Devices'. *Proceedings of the 1st Workshop on Isolation and Integration in Embedded Systems*. IIES '08. Glasgow, Scotland: ACM, 2008: pp. 17–22. ISBN: 978-1-60558-126-2. DOI: 10.1145/1435458.1435462. URL: http://doi.acm.org/10.1145/1435458.1435462 (cit. on p. 30).

[Bro18] Brost, Gerd S., Manuel Huber, Michael Weiss, Mykolai Protsenko, Julian Schütte, and Sascha Wessel: 'An Ecosystem and IoT Device Architecture for Building Trust in the Industrial Data Space'. *Proceedings of the 4th ACM Workshop on Cyber-Physical System Security*. CPSS '18. Incheon, Republic of Korea: ACM, 2018: pp. 39–50. ISBN: 978-1-4503-5755-5. DOI: 10.1145/3198458.3198459. URL: https://doi.org/10.1145/3198458.3198459 (cit. on pp. 7, 26, 66, 69, 159, 160).

[Bug11] Bugiel, Sven, Lucas Davi, Alexandra Dmitrienko, Stephan Heuser, Ahmad-Reza Sadeghi, and Bhargava Shastry: 'Practical and Lightweight Domain Isolation on Android'. *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*. SPSM '11. Chicago, Illinois, USA: ACM, 2011: pp. 51–62. isbn: 978-1-4503-1000-0. doi: 10.1145/2046614.2046624. url: http://doi.acm.org/10.1145/2046614.2046624 (cit. on p. 26).

[Bug13] Bugiel, Sven, Stephan Heuser, and Ahmad-Reza Sadeghi: 'Flexible and Fine-grained Mandatory Access Control on Android for Diverse Security and Privacy Policies'. *Proceedings of the 22nd USENIX Conference on Security*. SEC'13. Washington, D.C.: USENIX Association, 2013: pp. 131–146. isbn: 978-1-931971-03-4. url: http://dl.acm.org/citation.cfm?id=2534766.2534778 (cit. on p. 26).

[Buh17] Buhren, Robert, Shay Gueron, Jan Nordholz, Jean-Pierre Seifert, and Julian Vetter: 'Fault Attacks on Encrypted General Purpose Compute Platforms'. *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*. CODASPY '17. Scottsdale, Arizona, USA: ACM, 2017: pp. 197–204. isbn: 978-1-4503-4523-1. doi: 10.1145/3029806.3029836. url: http://doi.acm.org/10.1145/3029806.3029836 (cit. on p. 96).

[Car11] Carbone, Richard, C Bean, and M Salois: *An In-Depth Analysis of the Cold Boot Attack: Can It Be Used for Sound Forensic Memory Acquisition?* Tech. rep. http://www.dtic.mil/docs/citations/ADA545078. *Last access: September 4, 2019*. DTIC Document, 2011: p. 118 (cit. on p. 76).

[Cel16] Celesti, Antonio, Davide Mulfari, Maria Fazio, Massimo Villari, and Antonio Puliafito: 'Exploring Container Virtualization in IoT Clouds'. *2016 IEEE International Conference on Smart Computing (SMARTCOMP)*. IEEE, 2016: pp. 1–6. doi: 10.1109/SMARTCOMP.2016.7501691 (cit. on p. 28).

[Cha08] Chan, Ellick M., Jeffrey C. Carlyle, Francis M. David, Reza Farivar, and Roy H. Campbell: 'BootJacker: Compromising Computers Using Forced Restarts'. *Proceedings of the 15th ACM Conference on Computer and Communications Security*. CCS '08. Alexandria, Virginia, USA: ACM, 2008: pp. 555–564. isbn: 978-1-59593-810-7. doi: 10.1145/1455770.1455840. url: http://doi.acm.org/10.1145/1455770.1455840 (cit. on p. 75).

[Che01] Chen, Peter M. and Brian D. Noble: 'When Virtual Is Better Than Real'. *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*. HOTOS '01. Washington, DC, USA: IEEE Computer Society, 2001: pp. 133–138. url: http://dl.acm.org/citation.cfm?id=874075.876409 (cit. on p. 77).

[Che15] Chen, Wenzhi, Lei Xu, Guoxi Li, and Yang Xiang: 'A Lightweight Virtualization Solution for Android Devices'. *IEEE Transactions on Computers*. Vol. 64. 10. Washington, DC, USA: IEEE Computer Society, Oct. 2015: pp. 2741–2751. doi: 10.1109/TC.2015.2389791. url: http://dx.doi.org/10.1109/TC.2015.2389791 (cit. on pp. 28, 30).

[Che08] Chen, Xi, Robert P. Dick, and Alok Choudhary: 'Operating System Controlled Processor-memory Bus Encryption'. *Proceedings of the Conference on Design, Automation and Test in Europe*. DATE '08. Munich, Germany: ACM, 2008: pp. 1154–1159. isbn: 978-3-9810801-3-1. doi: 10.1145/1403375.1403657. url: http://doi.acm.org/10.1145/1403375.1403657 (cit. on pp. 96, 98).

[Chi11] Chin, Erika, Adrienne Porter Felt, Kate Greenwood, and David Wagner: 'Analyzing Inter-application Communication in Android'. *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*. MobiSys '11. Bethesda, Maryland, USA: ACM, 2011: pp. 239–252. isbn: 978-1-4503-0643-0. doi: 10.1145/1999995.2000018. url: http://doi.acm.org/10.1145/1999995.2000018 (cit. on pp. 1, 26, 30).

[Cho05] Chow, Jim, Ben Pfaff, Tal Garfinkel, and Mendel Rosenblum: 'Shredding Your Garbage: Reducing Data Lifetime Through Secure Deallocation'. *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*. SSYM'05. Baltimore, MD: USENIX Association, 2005: pp. 22–22. url: http://dl.acm.org/citation.cfm?id=1251398.1251420 (cit. on p. 103).

[Col15] Colp, Patrick, 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'. *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '15. Istanbul, Turkey: ACM, 2015: pp. 177–189. isbn: 978-1-4503-2835-7. doi: 10.1145/2694344.2694380. url: http://doi.acm.org/10.1145/2694344.2694380 (cit. on pp. 76, 98, 103).

[Cor03] Corner, Mark D. and Brian D. Noble: 'Protecting Applications with Transient Authentication'. *Proceedings of the 1st International Conference on Mobile Systems, Applications and Services*. MobiSys '03. San Francisco, California: ACM, 2003: pp. 57–70. doi: 10.1145/1066116.1066117. url: http://doi.acm.org/10.1145/1066116.1066117 (cit. on p. 100).

[Cor02] Corner, Mark D. and Brian D. Noble: 'Zero-interaction Authentication'. *Proceedings of the 8th Annual International Conference on Mobile Computing and Networking*. MobiCom '02. Atlanta, Georgia, USA: ACM, 2002: pp. 1–11. isbn: 1-58113-486-X. doi: 10.1145/570645.570647. url: http://doi.acm.org/10.1145/570645.570647 (cit. on p. 100).

[Cos16] Costan, Victor and Srinivas Devadas: *Intel SGX Explained*. Cryptology ePrint Archive, Report 2016/086. https://eprint.iacr.org/2016/086. 2016 (cit. on p. 2).

[CTS18] CTS Labs: *Severe Security Advisory on AMD Processors*. Tech. rep. https://safefirmware.com/amdflaws_whitepaper.pdf. *Last access: September 4, 2019*. 2018 (cit. on p. 136).

[Cya] CyanogenMod: *Github – CyanogenMod/android_kernel_samsung_jf*. https://github.com/CyanogenMod/android_kernel_samsung_jf. *Last access: September 4, 2019* (cit. on pp. 80, 82).

[Dal14] Dall, Christoffer and Jason Nieh: 'KVM/ARM: The Design and Implementation of the Linux ARM Hypervisor'. *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '14. Salt Lake City, Utah, USA: ACM, 2014: pp. 333–348. ISBN: 978-1-4503-2305-5. DOI: 10.1145/2541940.2541946. URL: http://doi.acm.org/10.1145/2541940.2541946 (cit. on p. 26).

[Dav09] Davi, Lucas, Ahmad-Reza Sadeghi, and Marcel Winandy: 'Dynamic Integrity Measurement and Attestation: Towards Defense Against Return-oriented Programming Attacks'. *Proceedings of the 2009 ACM Workshop on Scalable Trusted Computing*. STC '09. Chicago, Illinois, USA: ACM, 2009: pp. 49–54. ISBN: 978-1-60558-788-2. DOI: 10.1145/1655108.1655117. URL: http://doi.acm.org/10.1145/1655108.1655117 (cit. on p. 2).

[Dev09] Devine, Christophe and Guillaume Vissian: *Compromission Physique par le Bus PCI*. Proceedings of SSTIC, Thales Security Systems. 2009 (cit. on pp. 3, 76, 106, 136).

[Dol81] Dolev, Danny and Andrew Yao: 'On the Security of Public Key Protocols'. *Proceedings of the 22Nd Annual Symposium on Foundations of Computer Science*. SFCS '81. Washington, DC, USA: IEEE Computer Society, 1981: pp. 350–357. DOI: 10.1109/SFCS.1981.32. URL: https://doi.org/10.1109/SFCS.1981.32 (cit. on p. 41).

[Du17] Du, Zhao-Hui, Zhiwei Ying, Zhenke Ma, Yufei Mai, Phoebe Wang, Jesse Liu, and Jesse Fang: *Secure Encrypted Virtualization is Unsecure*. Dec. 2017. arXiv: 1712.05090 [cs.CR]. URL: https://arxiv.org/abs/1712.05090 (cit. on p. 137).

[Duc06] Duc, Guillaume and Ronan Keryell: 'CryptoPage: An Efficient Secure Architecture with Memory Encryption, Integrity and Information Leakage Protection'. *Proceedings of the 22Nd Annual Computer Security Applications Conference*. ACSAC '06. Washington, DC, USA: IEEE Computer Society, 2006: pp. 483–492. ISBN: 0-7695-2716-7. DOI: 10.1109/ACSAC.2006.21. URL: https://doi.org/10.1109/ACSAC.2006.21 (cit. on p. 96).

[Emba] Embedded Microprocessor Benchmark Consortium: *CPU Benchmark - MCU Benchmark - CoreMark*. http://www.eembc.org/coremark/. *Last access: September 4, 2019* (cit. on p. 132).

[Embb] Embedi: *Remotely Compromise Devices by using Bugs in Marvell Avastar Wi-Fi: From Zero Knowledge to Zero-click RCE*. https://embedi.org/blog/remotely-compromise-devices-by-using-bugs-in-marvell-avastar-wi-fi-from-zero-knowledge-to-zero-click-rce/. *Last access: September 4, 2019* (cit. on p. 21).

[Enc10] Enck, William, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth: 'TaintDroid: An Information-flow Tracking System for Realtime Privacy Monitoring on Smartphones'. *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*. OSDI'10. Vancouver, BC, Canada: Berkeley, CA, USA:USENIX Association, 2010: pp. 393–407. URL: http://dl.acm.org/citation.cfm?id=1924943.1924971 (cit. on p. 26).

[Enc09] Enck, William, Machigar Ongtang, and Patrick McDaniel: 'On Lightweight Mobile Phone Application Certification'. *Proceedings of the 16th ACM Conference on Computer and Communications Security*. CCS '09. Chicago, Illinois, USA: ACM, 2009: pp. 235–245. ISBN: 978-1-60558-894-0. DOI: 10.1145/1653662.1653691. URL: http://doi.acm.org/10.1145/1653662.1653691 (cit. on pp. 1, 26, 30).

[Eng03] England, Paul, Butler Lampson, John Manferdelli, Marcus Peinado, and Bryan Willman: 'A Trusted Open Platform'. *Computer* (July 2003), vol. 36(7): pp. 55–62. ISSN: 0018-9162. DOI: 10.1109/MC.2003.1212691. URL: http://dx.doi.org/10.1109/MC.2003.1212691 (cit. on p. 2).

[EPN] EPN Solutions: *Test & Measurement Probes and Adapters*. http://www.epnsolutions.net/ddr.html. *Last access: September 4, 2019* (cit. on p. 21).

[Exo] Exodus Intel VRT: *Broadpwn: Remotely Compromising Android and iOS via a Bug in Broadcom's Wi-Fi Chipsets*. https://blog.exodusintel.com/2017/07/26/broadpwn/. *Last access: September 4, 2019* (cit. on p. 21).

[Faz13] Fazio, Maria, Antonio Celesti, and Massimo Villari: 'Design of a Message-Oriented Middleware for Cooperating Clouds'. *European Conference on Service-Oriented and Cloud Computing*. Ed. by Canal, Carlos and Massimo Villari. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013: pp. 25–36. ISBN: 978-3-642-45364-9 (cit. on p. 28).

[Fei15] Feizollah, Ali, Nor Badrul Anuar, Rosli Salleh, and Ainuddin Wahid Abdul Wahab: 'A Review on Feature Selection in Mobile Malware Detection'. *Digital Investigation: The International Journal of Digital Forensics & Incident Response*. Vol. 13. C. Amsterdam, The Netherlands: Elsevier Science Publishers B. V., June 2015: pp. 22–37. DOI: 10.1016/j.diin.2015.02.001. URL: http://dx.doi.org/10.1016/j.diin.2015.02.001 (cit. on pp. 1, 30).

[Fel11] Felt, Adrienne Porter, Matthew Finifter, Erika Chin, Steve Hanna, and David Wagner: 'A Survey of Mobile Malware in the Wild'. *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*. SPSM '11. Chicago, Illinois, USA: ACM, 2011: pp. 3–14. ISBN: 978-1-4503-1000-0. DOI: 10.1145/2046614.2046618. URL: http://doi.acm.org/10.1145/2046614.2046618 (cit. on pp. 1, 26, 30).

[Fel15] Felter, Wes, Alexandre Ferreira, Ram Rajamony, and Juan Rubio: 'An Updated Performance Comparison of Virtual Machines and Linux Containers'. *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium On*. IEEE, 2015: pp. 171–172. doi: 10.1109/ISPASS.2015.7095802 (cit. on p. 27).

[Fraa] Fraunhofer AISEC: *trustm3 is trust/me – The Home of Project trust/me (Trusted Mobile Equipment)*. https://github.com/trustm3. *Last access: September 4, 2019* (cit. on pp. 6, 27, 30, 44, 53, 69, 72).

[Frab] Fraunhofer AISEC: *What is trust/me*. https://trustm3.github.io/. *Last access: September 4, 2019* (cit. on pp. 27, 72).

[Fut] FuturePlus Systems: *Bus Analysis Tools*. https://www.futureplus.com. *Last access: September 4, 2019* (cit. on p. 21).

[Gar03] Garfinkel, Tal, Mendel Rosenblum, and Dan Boneh: 'Flexible OS Support and Applications for Trusted Computing'. *Proceedings of the 9th Conference on Hot Topics in Operating Systems - Volume 9*. HOTOS'03. Lihue, Hawaii: USENIX Association, 2003: pp. 25–25. url: http://dl.acm.org/citation.cfm?id=1251054.1251079 (cit. on p. 2).

[Ger14] Gerlach, Wolfgang, Wei Tang, Kevin Keegan, Travis Harrison, Andreas Wilke, Jared Bischof, Mark D'Souza, Scott Devoid, Daniel Murphy-Olson, Narayan Desai, and Folker Meyer: 'Skyport: Container-based Execution Environment Management for Multi-cloud Scientific Workflows'. *Proceedings of the 5th International Workshop on Data-Intensive Computing in the Clouds*. DataCloud '14. New Orleans, Louisiana: IEEE Press, 2014: pp. 25–32. isbn: 978-1-4799-7034-6. doi: 10.1109/DataCloud.2014.6. url: https://doi.org/10.1109/DataCloud.2014.6 (cit. on p. 27).

[Gla] Glass, Echidna: *Heimdall*. https://glassechidna.com.au/heimdall. *Last access: September 4, 2019* (cit. on p. 84).

[Gol] Goldman, Ken: *IBM's TPM 2.0 TSS*. https://sourceforge.net/projects/ibmtpm20tss. *Last access: September 4, 2019* (cit. on p. 70).

[Gooa] Google: *ADB Fastboot Install – A script to install ADB & Fastboot on Mac OS X and/or Linux*. https://code.google.com/archive/p/adb-fastboot-install/. *Last access: September 4, 2019* (cit. on p. 85).

[Goob] Google: *Android Developers: Android Things*. https://developer.android.com/things. *Last access: September 4, 2019* (cit. on p. 164).

[Gooc] Google: *kernel/msm – Kernel Tree for Qualcomm Chipsets*. https://android.googlesource.com/kernel/msm. *Last access: September 4, 2019* (cit. on p. 44).

[Good] Google Cloud Security Team: *AMD-PSP: fTPM Remote Code Execution via Crafted EK Certificate*. http://seclists.org/fulldisclosure/2018/Jan/12. *Last access: September 4, 2019* (cit. on p. 164).

[Gooe] Google Developers: *Protocol Buffers - Google's Data Interchange format*. `https://github.com/google/protobuf`. *Last access: September 4, 2019* (cit. on pp. 44, 71).

[Göt16a] Götzfried, Johannes, Nico Dörr, Ralph Palutke, and Tilo Müller: 'HyperCrypt: Hypervisor-Based Encryption of Kernel and User Space'. *11th International Conference on Availability, Reliability and Security (ARES)*. IEEE, Aug. 2016: pp. 79–87. doi: 10.1109/ARES.2016.13 (cit. on p. 97).

[Göt13] Götzfried, Johannes and Tilo Müller: 'ARMORED: CPU-Bound Encryption for Android-Driven ARM Devices'. *Proceedings of the 2013 International Conference on Availability, Reliability and Security*. ARES '13. Washington, DC, USA: IEEE Computer Society, 2013: pp. 161–168. isbn: 978-0-7695-5008-4. doi: 10.1109/ARES.2013.23. url: http://dx.doi.org/10.1109/ARES.2013.23 (cit. on pp. 76, 96).

[Göt16b] Götzfried, Johannes, Tilo Müller, Gabor Drescher, Stefan Nürnberger, and Michael Backes: 'RamCrypt: Kernel-based Address Space Encryption for User-mode Processes'. *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*. ASIA CCS '16. Xi'an, China: ACM, 2016: pp. 919–924. isbn: 978-1-4503-4233-9. doi: 10.1145/2897845.2897924. url: http://doi.acm.org/10.1145/2897845.2897924 (cit. on p. 98).

[Gre12] Greene, James: *Intel Trusted Execution Technology*. Intel Technology White Paper. `https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/trusted-execution-technology-security-paper.pdf`. *Last access: September 4, 2019*. 2012 (cit. on p. 99).

[Gru13] Gruhn, Michael and Tilo Müller: 'On the Practicability of Cold Boot Attacks'. *Proceedings of the 2013 International Conference on Availability, Reliability and Security*. ARES '13. Washington, DC, USA: IEEE Computer Society, 2013: pp. 390–397. isbn: 978-0-7695-5008-4. doi: 10.1109/ARES.2013.52. url: https://doi.org/10.1109/ARES.2013.52 (cit. on pp. 3, 73, 76).

[Gru16] Gruss, Daniel, Clémentine Maurice, and Stefan Mangard: 'Rowhammer.Js: A Remote Software-Induced Fault Attack in JavaScript'. *Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment - Volume 9721*. DIMVA 2016. San Sebastian, Spain: Springer-Verlag, 2016: pp. 300–321. isbn: 978-3-319-40666-4. doi: 10.1007/978-3-319-40667-1_15. url: https://doi.org/10.1007/978-3-319-40667-1_15 (cit. on p. 2).

[Gua14] Guan, Le, Jingqiang Lin, Bo Luo, and Jiwu Jing: 'Copker: Computing with Private Keys without RAM'. *NDSS*. 2014: pp. 23–26 (cit. on p. 96).

[Gua15] Guan, Le, Jingqiang Lin, Bo Luo, Jiwu Jing, and Jing Wang: 'Protecting Private Keys Against Memory Disclosure Attacks Using Hardware Transactional Memory'. *Proceedings of the 2015 IEEE Symposium on Security and Privacy*. SP '15. Washington, DC, USA: IEEE Computer Society, 2015: pp. 3–19. ISBN: 978-1-4673-6949-7. DOI: 10.1109/SP.2015.8. URL: https://doi.org/10.1109/SP.2015.8 (cit. on p. 96).

[Gua] Guangrong, Xiao: *[PATCH v3 00/11] KVM: x86: Track Guest Page Access*. http://www.mail-archive.com/linux-kernel@vger.kernel.org/msg1076006.html. *Last access: September 4, 2019* (cit. on pp. 143, 150).

[Gut01] Gutmann, Peter: 'Data Remanence in Semiconductor Devices'. *Proceedings of the 10th Conference on USENIX Security Symposium*. Vol. 10. SSYM'01. Washington, D.C.: USENIX Association, 2001. URL: http://dl.acm.org/citation.cfm?id=1251327.1251331 (cit. on pp. 3, 22, 73, 106).

[Gut99] Gutmann, Peter: 'The Design of a Cryptographic Security Architecture'. *Proceedings of the 8th Conference on USENIX Security Symposium - Volume 8*. SSYM'99. Washington, D.C.: USENIX Association, 1999: pp. 13–13. URL: http://dl.acm.org/citation.cfm?id=1251421.1251434 (cit. on p. 96).

[Hal09] Halderman, J. Alex, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten: 'Lest We Remember: Cold-boot Attacks on Encryption Keys'. *Communications of the ACM*. Vol. 52. 5. New York, NY, USA: ACM, May 2009: pp. 91–98. DOI: 10.1145/1506409.1506429. URL: http://doi.acm.org/10.1145/1506409.1506429 (cit. on pp. 3, 75, 86, 106).

[Het17] Hetzelt, Felicitas and Robert Buhren: 'Security Analysis of Encrypted Virtual Machines'. *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. VEE '17. Xi'an, China: ACM, 2017: pp. 129–142. ISBN: 978-1-4503-4948-2. DOI: 10.1145/3050748.3050763. URL: http://doi.acm.org/10.1145/3050748.3050763 (cit. on pp. 96, 137, 140).

[Hew] Hewitt, Joey: *Search an Android RAM Dump for Linux dm-crypt (incl. LUKS/cryptsetup) Keys*. https://github.com/scintill/keysearch. *Last access: September 4, 2019* (cit. on pp. 73, 80, 81).

[Hil14] Hilgers, Christian, Holger Macht, Tilo Müller, and Michael Spreitzenbarth: 'Post-Mortem Memory Analysis of Cold-Booted Android Devices'. *Proceedings of the 2014 Eighth International Conference on IT Security Incident Management & IT Forensics*. IMF '14. Washington, DC, USA: IEEE Computer Society, 2014: pp. 62–75. ISBN: 978-1-4799-4328-9. DOI: 10.1109/IMF.2014.8. URL: http://dx.doi.org/10.1109/IMF.2014.8 (cit. on pp. 76, 81).

[Hoo11] Hoog, Andrew: *Android Forensics: Investigation, Analysis and Mobile Security for Google Android*. Syngress Publishing, 2011. ISBN: 9781597496520 (cit. on p. 74).

[Hor17] Horsch, Julian, Manuel Huber, and Sascha Wessel: 'TransCrypt: Transparent Main Memory Encryption Using a Minimal ARM Hypervisor'. *Proceedings of the 16th International Conference on Trust, Security and Privacy in Computing and Communications*. TrustCom '17. Sydney, Australia: IEEE, Aug. 2017: pp. 408–417. doi: 10.1109/Trustcom/BigDataSE/ICESS.2017.232 (cit. on pp. 10, 95, 96, 131, 132, 162).

[Hub17a] Huber, Manuel, Julian Horsch, Junaid Ali, and Sascha Wessel: 'Freeze & Crypt: Linux Kernel Support for Main Memory Encryption'. *14th International Conference on Security and Cryptography*. SECRYPT 2017. Madrid, Spain: ScitePress, 2017: pp. 17–30. isbn: 978-989-758-259-2. doi: 10.5220/0006378400170030 (cit. on pp. 12, 95).

[Hub18] Huber, Manuel, Julian Horsch, Junaid Ali, and Sascha Wessel: 'Freeze and Crypt: Linux Kernel Support for Main Memory Encryption'. *Computers & Security* (2018), vol. 86: pp. 420–436. issn: 0167-4048. doi: 10.1016/j.cose.2018.08.011. url: http://www.sciencedirect.com/science/article/pii/S0167404818310435 (cit. on pp. 9, 12, 95, 161).

[Hub16a] Huber, Manuel, Julian Horsch, Michael Velten, Michael Weiss, and Sascha Wessel: 'A Secure Architecture for Operating System-Level Virtualization on Mobile Devices'. *Revised Selected Papers of the 11th International Conference on Information Security and Cryptology - Volume 9589*. Inscrypt 2015. Beijing, China: Springer-Verlag New York, Inc., 2016: pp. 430–450. isbn: 978-3-319-38897-7. doi: 10.1007/978-3-319-38898-4_25. url: http://dx.doi.org/10.1007/978-3-319-38898-4_25 (cit. on pp. 6, 7, 26, 159).

[Hub17b] Huber, Manuel, Julian Horsch, and Sascha Wessel: 'Protecting Suspended Devices from Memory Attacks'. *Proceedings of the 10th European Workshop on Systems Security*. EuroSec'17. Belgrade, Serbia: ACM, 2017: 10:1–10:6. isbn: 978-1-4503-4935-2. doi: 10.1145/3065913.3065914. url: http://doi.acm.org/10.1145/3065913.3065914 (cit. on pp. 8, 95, 161).

[Hub16b] Huber, Manuel, Benjamin Taubmann, Sascha Wessel, Hans P. Reiser, and Georg Sigl: 'A Flexible Framework for Mobile Device Forensics Based on Cold Boot Attacks'. *EURASIP Journal on Information Security*. Vol. 2016. 1. New York, NY, United States: Hindawi Publishing Corp., Dec. 2016: 41:1–41:13. doi: 10.1186/s13635-016-0041-4. url: https://doi.org/10.1186/s13635-016-0041-4 (cit. on pp. 8, 12, 74, 160).

[Hun] Hunt, Galen: *Introducing Microsoft Azure Sphere: Secure and power the intelligent edge*. https://azure.microsoft.com/en-us/blog/introducing-microsoft-azure-sphere-secure-and-power-the-intelligent-edge. *Last access: September 4, 2019* (cit. on p. 164).

[Hwa08] Hwang, Joo-Young, Sang-Bum Suh, Sung-Kwan Heo, Chan-Ju Park, Jae-Min Ryu, Seong-Yeol Park, and Chul-Ryun Kim: 'Xen on ARM: System Virtualization Using Xen Hypervisor for ARM-Based Secure Mobile Phones'. *5th IEEE Consumer Communications and Networking Conference*. IEEE, 2008: pp. 257–261. DOI: 10.1109/ccnc08.2007.64 (cit. on p. 26).

[Ind] Industrial Data Space Association: *IoT edge platform "Trusted Connector"*. https://github.com/industrial-data-space/trusted-connector. *Last access: September 4, 2019* (cit. on pp. 53, 66).

[Int] Intel Corporation: *Intel Architecture Memory Encryption Technologies Specification*. https://software.intel.com/sites/default/files/managed/a5/16/Multi-Key-Total-Memory-Encryption-Spec.pdf. *Last access: September 4, 2019* (cit. on pp. 97, 138, 163).

[Kam00] Kamp, Poul-Henning and Robert N. M. Watson: 'Jails: Confining the Omnipotent Root'. *Proceedings of the 2nd International SANE Conference*. Vol. 43. 2000: p. 116 (cit. on p. 27).

[Kap17] Kaplan, David: *Protecting VM Register State with SEV-ES*. Advanced Micro Devices. https://www.amd.com/system/files/TechDocs/Protecting%20VM%20Register%20State%20with%20SEV-ES.pdf. *Last access: September 4, 2019*. Feb. 2017 (cit. on p. 135).

[Kap16] Kaplan, David, Jeremy Powell, and Tom Woller: *AMD Memory Encryption*. Tech. rep. https://developer.amd.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf. *Last access: September 4, 2019*. Advanced Micro Devices, 2016 (cit. on pp. 96, 136).

[Kau17] Kaur, Kuljeet, Tanya Dhand, Neeraj Kumar, and Sherali Zeadally: 'Container-as-a-Service at the Edge: Trade-off between Energy Efficiency and Service Availability at Fog Nano Data Centers'. *IEEE wireless communications* (June 2017), vol. 24(3): pp. 48–56. ISSN: 1536-1284. DOI: 10.1109/MWC.2017.1600427 (cit. on p. 27).

[Kil09] Kil, Chongkyung, Emre C. Sezer, Ahmed M. Azab, Peng Ning, and Xiaolan Zhang: 'Remote Attestation to Dynamic System Properties: Towards Providing Complete System Integrity Evidence'. *IEEE/IFIP International Conference on Dependable Systems Networks*. IEEE, June 2009: pp. 115–124. DOI: 10.1109/DSN.2009.5270348 (cit. on p. 2).

[Kim14] Kim, Yoongu, 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'. *Proceeding of the 41st Annual International Symposium on Computer Architecuture*. ISCA '14. Minneapolis, Minnesota, USA: IEEE Press, 2014: pp. 361–372. ISBN: 978-1-4799-4394-4. URL: http://dl.acm.org/citation.cfm?id=2665671.2665726 (cit. on p. 2).

[Koc18] Kocher, Paul, 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 (cit. on p. 2).

[Li19] Li, Mengyuan, Yinqian Zhang, Zhiqiang Lin, and Yan Solihin: 'Exploiting Unprotected I/O Operations in AMD's Secure Encrypted Virtualization'. *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, 2019. URL: https://www.usenix.org/conference/usenixsecurity19/presentation/li-mengyuan (cit. on pp. 137, 157).

[Lib] LibVMI Project: *LibVMI Virtual Machine Introspection*. http://libvmi.com. *Last access: September 4, 2019* (cit. on p. 136).

[Lie03] Lie, David, John Mitchell, Chandramohan A. Thekkath, and Mark Horowitz: 'Specifying and Verifying Hardware for Tamper-Resistant Software'. *Proceedings of the 2003 IEEE Symposium on Security and Privacy*. SP '03. Washington, DC, USA: IEEE Computer Society, 2003: pp. 166–177. ISBN: 0-7695-1940-7. URL: http://dl.acm.org/citation.cfm?id=829515.830564 (cit. on p. 96).

[Lie00] Lie, David, Chandramohan Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John Mitchell, and Mark Horowitz: 'Architectural Support for Copy and Tamper Resistant Software'. *ACM SIGPLAN Notices*. Vol. 35. 11. New York, NY, USA: ACM, Nov. 2000: pp. 168–177. DOI: 10.1145/356989.357005. URL: http://doi.acm.org/10.1145/356989.357005 (cit. on p. 96).

[Lin15] Lindenlauf, Simon, Hans Höfken, and Marko Schuba: 'Cold Boot Attacks on DDR2 and DDR3 SDRAM'. *Proceedings of the 2015 10th International Conference on Availability, Reliability and Security*. ARES '15. Washington, DC, USA: IEEE Computer Society, 2015: pp. 287–292. ISBN: 978-1-4673-6590-1. DOI: 10.1109/ARES.2015.28. URL: http://dx.doi.org/10.1109/ARES.2015.28 (cit. on pp. 76, 77, 86).

[Lip18a] Lipp, Moritz, 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 (cit. on p. 2).

[Lip18b] Lipp, Moritz, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg: 'Meltdown: Reading Kernel Memory from User Space'. *Proceedings of the 27th USENIX Conference on Security Symposium*. SEC'18. Baltimore, MD, USA: USENIX Association, 2018: pp. 973–990. ISBN: 978-1-931971-46-1. URL: http://dl.acm.org/citation.cfm?id=3277203.3277276 (cit. on p. 2).

[Lit] LITTLEKERNEL: *Introduction – Littlekernel/LK Wiki*. https://github.com/littlekernel/lk/wiki/Introduction. *Last access: September 4, 2019* (cit. on p. 92).

[Maa12] MAARTMANN-MOE, CARSTEN: *Adventures with Daisy in Thunderbolt-DMA-Land: Hacking Macs through the Thunderbolt Interface*. Break & Enter. 2012 (cit. on pp. 3, 76).

[Mau84] MAUDE, TIM and DERWENT MAUDE: 'Hardware Protection Against Software Piracy'. *Communications of the ACM*. Vol. 27. 9. New York, NY, USA: ACM, Sept. 1984: pp. 950–959. DOI: 10.1145/358234.358271. URL: http://doi.acm.org/10.1145/358234.358271 (cit. on p. 96).

[McC10] MCCUNE, JONATHAN M., YANLIN LI, NING QU, ZONGWEI ZHOU, ANUPAM DATTA, VIRGIL GLIGOR, and ADRIAN PERRIG: 'TrustVisor: Efficient TCB Reduction and Attestation'. *Proceedings of the 2010 IEEE Symposium on Security and Privacy*. SP '10. Washington, DC, USA: IEEE Computer Society, 2010: pp. 143–158. ISBN: 978-0-7695-4035-1. DOI: 10.1109/SP.2010.17. URL: http://dx.doi.org/10.1109/SP.2010.17 (cit. on p. 2).

[McC08] MCCUNE, JONATHAN M., BRYAN J. PARNO, ADRIAN PERRIG, MICHAEL K. REITER, and HIROSHI ISOZAKI: 'Flicker: An Execution Infrastructure for TCB Minimization'. *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*. Eurosys '08. Glasgow, Scotland UK: ACM, 2008: pp. 315–328. ISBN: 978-1-60558-013-5. DOI: 10.1145/1352592.1352625. URL: http://doi.acm.org/10.1145/1352592.1352625 (cit. on p. 2).

[McK13] MCKEEN, FRANK, ILYA ALEXANDROVICH, ALEX BERENZON, CARLOS V. ROZAS, HISHAM SHAFI, VEDVYAS SHANBHOGUE, and UDAY R. SAVAGAONKAR: 'Innovative Instructions and Software Model for Isolated Execution'. *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*. HASP '13. Tel-Aviv, Israel: ACM, 2013: 10:1–10:1. ISBN: 978-1-4503-2118-1. DOI: 10.1145/2487726.2488368. URL: http://doi.acm.org/10.1145/2487726.2488368 (cit. on pp. 2, 97).

[Mer14] MERKEL, DIRK: 'Docker: Lightweight Linux Containers for Consistent Development and Deployment'. *Linux Journal*. Vol. 2014. 239. Houston, TX: Belltown Media, Mar. 2014. URL: http://dl.acm.org/citation.cfm?id=2600239.2600241 (cit. on p. 27).

[Mica] MICROCHIP: *BlueBorne Attack Vector*. https://www.microchip.com/design-centers/wireless-connectivity/bluetooth/blueborne-attack-vector. *Last access: September 4, 2019* (cit. on p. 21).

[Micb] MICROSOFT: *Azure Sphere Documentation*. https://docs.microsoft.com/en-us/azure-sphere/. *Last access: September 4, 2019* (cit. on p. 164).

[Micc] Microsoft: *Microsoft Security Bulletin MS17-008 - Critical*. https://docs.microsoft.com/en-us/security-updates/securitybulletins/2017/ms17-008. *Last access: September 4, 2019* (cit. on p. 135).

[Mor19] Morbitzer, Mathias, Manuel Huber, and Julian Horsch: 'Extracting Secrets from Encrypted Virtual Machines'. *Proceedings of the Ninth ACM on Conference on Data and Application Security and Privacy*. CODASPY '19. Richardson, Texas, USA: ACM, 2019: p. 10. isbn: 978-1-4503-6099-9. doi: 10.1145/3292006.3300022. url: https://doi.org/10.1145/3292006.3300022 (cit. on pp. 11, 135–138, 163).

[Mor18] Morbitzer, Mathias, Manuel Huber, Julian Horsch, and Sascha Wessel: 'SEVered: Subverting AMD's Virtual Machine Encryption'. *Proceedings of the 11th European Workshop on Systems Security*. EuroSec'18. Porto, Portugal: ACM, 2018: 1:1–1:6. isbn: 978-1-4503-5652-7. doi: 10.1145/3193111.3193112. url: http://doi.acm.org/10.1145/3193111.3193112 (cit. on pp. 10, 96, 135–138, 141–143, 149, 150, 155, 156, 162).

[Mul16] Mulfari, Davide, Maria Fazio, Antonio Celesti, Massimo Villari, and Antonio Puliafito: 'Design of an IoT Cloud System for Container Virtualization on Smart Objects'. *European Conference on Service-Oriented and Cloud Computing*. Ed. by Celesti, Antonio and Philipp Leitner. Cham: Springer International Publishing, 2016: pp. 33–47. isbn: 978-3-319-33313-7 (cit. on p. 28).

[Mül10] Müller, Tilo, Andreas Dewald, and Felix C. Freiling: 'AESSE: A Cold-boot Resistant Implementation of AES'. *Proceedings of the Third European Workshop on System Security*. EUROSEC '10. Paris, France: ACM, 2010: pp. 42–47. isbn: 978-1-4503-0059-9. doi: 10.1145/1752046.1752053. url: http://doi.acm.org/10.1145/1752046.1752053 (cit. on p. 96).

[Mül11] Müller, Tilo, Felix C. Freiling, and Andreas Dewald: 'TRESOR Runs Encryption Securely Outside RAM'. *Proceedings of the 20th USENIX Conference on Security*. SEC'11. San Francisco, CA: USENIX Association, 2011: pp. 17–17. url: http://dl.acm.org/citation.cfm?id=2028067.2028084 (cit. on pp. 76, 96, 103).

[Mül13] Müller, Tilo and Michael Spreitzenbarth: 'FROST: Forensic Recovery of Scrambled Telephones'. *Proceedings of the 11th International Conference on Applied Cryptography and Network Security*. ACNS'13. Banff, AB, Canada: Springer-Verlag, 2013: pp. 373–388. isbn: 978-3-642-38979-5. doi: 10.1007/978-3-642-38980-1_23. url: http://dx.doi.org/10.1007/978-3-642-38980-1_23 (cit. on pp. 3, 73–75, 86, 106).

[Mül12] Müller, Tilo, Benjamin Taubmann, and Felix C. Freiling: 'TreVisor: OS-independent Software-based Full Disk Encryption Secure Against Main Memory Attacks'. *Proceedings of the 10th International Conference on Applied Cryptography and Network Security*. ACNS'12. Singapore: Springer-Verlag, 2012: pp. 66–83. isbn: 978-3-642-31283-0. doi: 10.1007/978-3-642-31284-7_5.

URL: http://dx.doi.org/10.1007/978-3-642-31284-7_5 (cit. on pp. 76, 96).

[Nta14] Ntantogian, Christoforos, Dimitris Apostolopoulos, Giannis Marinakis, and Christos Xenakis: 'Evaluating the Privacy of Android Mobile Applications under Forensic Analysis'. *Computers & Security* (2014), vol. 42: pp. 66–76. ISSN: 0167-4048. DOI: 10.1016/j.cose.2014.01.004. URL: http://www.sciencedirect.com/science/article/pii/S0167404814000157 (cit. on pp. 3, 73, 76).

[Ong09] Ongtang, Machigar, Stephen McLaughlin, William Enck, and Patrick McDaniel: 'Semantically Rich Application-Centric Security in Android'. *Proceedings of the 2009 Annual Computer Security Applications Conference*. ACSAC '09. Washington, DC, USA: IEEE Computer Society, 2009: pp. 340–349. ISBN: 978-0-7695-3919-5. DOI: 10.1109/ACSAC.2009.39. URL: http://dx.doi.org/10.1109/ACSAC.2009.39 (cit. on pp. 1, 26, 30).

[Ope] OpenFog Consortium: *OpenFog Reference Architecture for Fog Computing*. https://www.iiconsortium.org/pdf/OpenFog_Reference_Architecture_2_09_17.pdf. *Last access: September 4, 2019* (cit. on p. 53).

[OSG] OSGi Alliance: *Architecture - OSGi Alliance*. https://www.osgi.org/developer/architecture. *Last access: September 4, 2019* (cit. on p. 66).

[Ott16] Otto, Boris, Sören Auer, Jan Cirullies, Jan Jürjens, Nadja Menz, Jochen Schon, and Sven Wenzel: *Industrial Data Space: Digital Sovereignity over Data*. Fraunhofer-Gesellschaft. Munich, Germany, 2016. DOI: 10.13140/RG.2.1.2673.0649 (cit. on pp. 53, 55).

[Ott17] Otto, Boris, Steffen Lohmann, Sören Auer, Gerd Brost, Jan Cirullies, Andreas Eitel, Thilo Ernst, Christian Haas, Manuel Huber, Christian Jung, et al.: *Reference Architecture Model for the Industrial Data Space*. Fraunhofer-Gesellschaft. Munich, Germany, 2017 (cit. on p. 53).

[Pah15] Pahl, Claus and Brian Lee: 'Containers and Clusters for Edge Cloud Architectures – A Technology Review'. *Proceedings of the 2015 3rd International Conference on Future Internet of Things and Cloud*. Washington, DC, USA: IEEE Computer Society, 2015: pp. 379–386. DOI: 10.1109/FiCloud.2015.35. URL: https://doi.org/10.1109/FiCloud.2015.35 (cit. on p. 27).

[Pap17] Papadopoulos, Panagiotis, Giorgos Vasiliadis, Giorgos Christou, Evangelos Markatos, and Sotiris Ioannidis: 'No Sugar but All the Taste! Memory Encryption Without Architectural Support'. *Computer Security – ESORICS 2017*. Ed. by Foley, Simon N., Dieter Gollmann, and Einar Snekkenes. Cham: Springer International Publishing, 2017: pp. 362–380. ISBN: 978-3-319-66399-9 (cit. on p. 98).

[Pay] Payer, Mathias: *AMD SEV Attack Surface: A Tale of too much Trust*. https://nebelwelt.net/blog/20160922-AMD-SEV-attack-surface.html. *Last access: September 4, 2019* (cit. on pp. 136, 140).

[Pei16] Peinl, René, Florian Holzschuher, and Florian Pfitzer: 'Docker Cluster Management for the Cloud - Survey Results and Own Solution'. *J. Grid Comput.* (June 2016), vol. 14(2): pp. 265–282. ISSN: 1570-7873. DOI: 10.1007/s10723-016-9366-y. URL: http://dx.doi.org/10.1007/s10723-016-9366-y (cit. on p. 27).

[Pen14] Peng, Sancheng, Shui Yu, and Aimin Yang: 'Smartphone Malware and Its Propagation Modeling: A Survey'. *IEEE Communications Surveys & Tutorials* (2014), vol. 16(2): pp. 925–941. ISSN: 1553-877X. DOI: 10.1109/SURV.2013.070813.00214 (cit. on pp. 1, 26, 30).

[Pet10] Peterson, Peter A. H.: 'Cryptkeeper: Improving security with encrypted RAM'. *2010 IEEE International Conference on Technologies for Homeland Security (HST)*. IEEE, Nov. 2010: pp. 120–126. DOI: 10.1109/THS.2010.5655081 (cit. on p. 97).

[Pet07] Pettersson, Torbjörn: *Cryptographic Key Recovery from Linux Memory Dumps*. Chaos Communication Camp. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.87.7761&rep=rep1&type=pdf. *Last access: September 4, 2019*. 2007 (cit. on pp. 3, 73, 80).

[Pló16] Plósz, Sándor, Csaba Hegedűs, and Pál Varga: 'Advanced Security Considerations in the Arrowhead Framework'. *International Conference on Computer Safety, Reliability, and Security*. Ed. by Skavhaug, Amund, Jérémie Guiochet, Erwin Schoitsch, and Friedemann Bitsch. Cham: Springer International Publishing, 2016: pp. 234–245. ISBN: 978-3-319-45480-1. DOI: 10.1007/978-3-319-45480-1_19 (cit. on p. 53).

[Poe14] Poeplau, Sebastian, Yanick Fratantonio, Antonio Bianchi, Christopher Kruegel, and Giovanni Vigna: 'Execute This! Analyzing Unsafe and Malicious Dynamic Code Loading in Android Applications'. *Proceedings of the 20th Annual Network & Distributed System Security Symposium (NDSS)*. Vol. 14. 2014: pp. 23–26. ISBN: 1-891562-35-5. DOI: 10.14722/ndss.2014.23328 (cit. on pp. 1, 26, 30).

[Pri] Princeton University: *Memory Research Project Source Code - Center for Information Technology Policy*. https://citp.princeton.edu/topics/memory/code. *Last access: September 4, 2019* (cit. on pp. 73, 75, 80, 149).

[Rek] Rekall Forensics: *Rekall*. http://www.rekall-forensic.com. *Last access: September 4, 2019* (cit. on p. 136).

[Res14] Reshetova, Elena, Janne Karhunen, Thomas Nyman, and N. Asokan: 'Security of OS-Level Virtualization Technologies'. *Secure IT Systems*. Ed. by Bernsmed, Karin and Simone Fischer-Hübner. Vol. 8788. Cham: Springer International Publishing, 2014: pp. 77–93. ISBN: 978-3-319-11598-6. DOI: 10.1007/978-3-319-11599-3_5 (cit. on p. 27).

[RIF] RIFFBox: *Rocker Team Flashing Interface: RIFF Box*. http://www.riffbox.org. *Last access: September 4, 2019* (cit. on p. 76).

[Ros] Rosenberg, Dan: *Azimuth Security: Re-visiting the Exynos Memory Mapping Bug*. http://blog.azimuthsecurity.com/2013/02/re-visiting-exynos-memory-mapping-bug.html. *Last access: September 4, 2019* (cit. on p. 76).

[Ros14] Rosenberg, Dan: *Reflections on Trusting TrustZone*. Black Hat USA. https://blackhat.com/docs/us-14/materials/us-14-Rosenberg-Reflections-on-Trusting-TrustZone.pdf. *Last access: September 4, 2019*. 2014 (cit. on p. 164).

[Ros12] Rossier, Daniel: *EmbeddedXEN: A Revisited Architecture of the XEN Hypervisor to support ARM-based Embedded Virtualization*. Reconfigurable Embedded Digital Systems Institute, School of Business and Engineering. Yverdon-les-Bains, Switzerland, 2012 (cit. on p. 26).

[Rus12] Russello, Giovanni, Mauro Conti, Bruno Crispo, and Earlence Fernandes: 'MOSES: Supporting Operation Modes on Smartphones'. *Proceedings of the 17th ACM Symposium on Access Control Models and Technologies*. SACMAT '12. Newark, New Jersey, USA: ACM, 2012: pp. 3–12. isbn: 978-1-4503-1295-0. doi: 10.1145/2295136.2295140. url: http://doi.acm.org/10.1145/2295136.2295140 (cit. on p. 26).

[Sai04] Sailer, Reiner, Xiaolan Zhang, Trent Jaeger, and Leendert van Doorn: 'Design and Implementation of a TCG-based Integrity Measurement Architecture'. *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*. SSYM'04. San Diego, CA: USENIX Association, 2004: pp. 16–16. url: http://dl.acm.org/citation.cfm?id=1251375.1251391 (cit. on p. 2).

[Sal14] Salvador, Otavio and Daiane Angolini: *Embedded Linux Development with Yocto Project*. Packt Publishing Ltd, 2014. isbn: 9781783282333 (cit. on p. 70).

[Sch] Schaufler, Casey: *LSM: Generalize Existing Module Stacking*. https://lwn.net/Articles/617372/. *Last access: September 4, 2019* (cit. on p. 33).

[Sch16] Schütte, Julian and Gerd Stefan Brost: 'A Data Usage Control System Using Dynamic Taint Tracking'. *2016 IEEE 30th International Conference on Advanced Information Networking and Applications (AINA)*. IEEE, 2016: pp. 909–916. doi: 10.1109/AINA.2016.127 (cit. on p. 69).

[Seo14] Seo, Kyoung-Taek, Hyun-Seo Hwang, Il-Young Moon, Oh-Young Kwon, and Byeong-Jun Kim: 'Performance Comparison Analysis of Linux Container and Virtual Machine for Building Cloud'. *Advanced Science and Technology Letters* (2014), vol. 66(105-111): p. 2 (cit. on p. 27).

[Sev13] Sevinsky, Russ: *Funderbolt: Adventures in Thunderbolt DMA Attacks*. Black Hat USA. https://media.blackhat.com/us-13/US-13-Sevinsky-Funderbolt-Adventures-in-Thunderbolt-DMA-Attacks-Slides.pdf. *Last access: September 4, 2019*. 2013 (cit. on pp. 3, 76).

[Sim11] Simmons, Patrick: 'Security Through Amnesia: A Software-based Solution to the Cold Boot Attack on Disk Encryption'. *Proceedings of the 27th Annual Computer Security Applications Conference*. ACSAC '11. Orlando, Florida, USA: ACM, 2011: pp. 73–82. isbn: 978-1-4503-0672-0. doi: 10.1145/2076732.2076743. url: http://doi.acm.org/10.1145/2076732.2076743 (cit. on p. 96).

[Ski13] Skillen, Adam, David Barrera, and Paul C. van Oorschot: 'Deadbolt: Locking Down Android Disk Encryption'. *Proceedings of the Third ACM Workshop on Security and Privacy in Smartphones & Mobile Devices*. SPSM '13. Berlin, Germany: ACM, 2013: pp. 3–14. isbn: 978-1-4503-2491-5. doi: 10.1145/2516760.2516771. url: http://doi.acm.org/10.1145/2516760.2516771 (cit. on p. 76).

[Ste13] Stewin, Patrick and Iurii Bystrov: 'Understanding DMA Malware'. *Proceedings of the 9th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. DIMVA'12. Heraklion, Crete, Greece: Springer-Verlag, 2013: pp. 21–41. isbn: 978-3-642-37299-5. doi: 10.1007/978-3-642-37300-8_2. url: http://dx.doi.org/10.1007/978-3-642-37300-8_2 (cit. on pp. 3, 106).

[Su09] Su, Lifeng, Stephan Courcambeck, Pierre Guillemin, Chr. Schwarz, and Renaud Pacalet: 'SecBus: Operating System Controlled Hierarchical Page-based Memory Bus Protection'. *Proceedings of the Conference on Design, Automation and Test in Europe*. DATE '09. Nice, France: European Design and Automation Association, 2009: pp. 570–573. isbn: 978-3-9810801-5-5. url: http://dl.acm.org/citation.cfm?id=1874620.1874759 (cit. on p. 96).

[Suh03] Suh, G. Edward, Dwaine Clarke, Blaise Gassend, Marten van Dijk, and Srinivas Devadas: 'Efficient Memory Integrity Verification and Encryption for Secure Processors'. *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO 36. Washington, DC, USA: IEEE Computer Society, 2003: pp. 339–350. isbn: 0-7695-2043-X. url: http://dl.acm.org/citation.cfm?id=956417.956575 (cit. on p. 96).

[Suh07] Suh, G. Edward, Charles W. O'Donnell, and Srinivas Devadas: 'Aegis: A Single-Chip Secure Processor'. *IEEE Design & Test*. Vol. 24. 6. Los Alamitos, CA, USA: IEEE Computer Society Press, Nov. 2007: pp. 570–580. doi: 10.1109/MDT.2007.179. url: http://dx.doi.org/10.1109/MDT.2007.179 (cit. on p. 96).

[Suh05] Suh, G. Edward, Charles W. O'Donnell, Ishan Sachdev, and Srinivas Devadas: 'Design and Implementation of the AEGIS Single-Chip Secure Processor Using Physical Random Functions'. *Proceedings of the 32nd Annual International Symposium on Computer Architecture*. ISCA '05. Washington, DC, USA: IEEE Computer Society, 2005: pp. 25–36. isbn: 0-7695-2270-X. doi: 10.1109/ISCA.2005.22. url: https://doi.org/10.1109/ISCA.2005.22 (cit. on p. 96).

[Syl12a] Sylve, Joe: *LiME-Linux Memory Extractor*. Proceedings of the 7th ShmooCon conference. https://github.com/504ensicslabs/lime. *Last access: September 4, 2019*. 2012 (cit. on pp. 75, 76).

[Syl12b] Sylve, Joe, Andrew Case, Lodovico Marziale, and Golden G Richard: 'Acquisition and Analysis of Volatile Memory from Android Devices'. *Digital Investigation* (2012), vol. 8(3): pp. 175–184. ISSN: 1742-2876. DOI: 10.1016/j.diin.2011.10.003. URL: http://www.sciencedirect.com/science/article/pii/S1742287611000879 (cit. on pp. 76, 81).

[Tan12] Tang, Yang, Phillip Ames, Sravan Bhamidipati, Ashish Bijlani, Roxana Geambasu, and Nikhil Sarda: 'CleanOS: Limiting Mobile Data Exposure with Idle Eviction'. *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*. OSDI'12. Hollywood, CA, USA: USENIX Association, 2012: pp. 77–91. ISBN: 978-1-931971-96-6. URL: http://dl.acm.org/citation.cfm?id=2387880.2387888 (cit. on pp. 3, 73, 99).

[Tat18] Tatar, Andrei, Radhesh Krishnan Konoth, Elias Athanasopoulos, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi: 'Throwhammer: Rowhammer Attacks over the Network and Defenses'. *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*. USENIX ATC '18. Boston, MA, USA: USENIX Association, 2018: pp. 213–225. ISBN: 978-1-931971-44-7. URL: http://dl.acm.org/citation.cfm?id=3277355.3277377 (cit. on p. 2).

[Tau15] Taubmann, Benjamin, Manuel Huber, Sascha Wessel, Lukas Heim, Hans Peter Reiser, and Georg Sigl: 'A Lightweight Framework for Cold Boot Based Forensics on Mobile Devices'. *Proceedings of the 2015 10th International Conference on Availability, Reliability and Security*. ARES '15. Washington, DC, USA: IEEE Computer Society, 2015: pp. 120–128. ISBN: 978-1-4673-6590-1. DOI: 10.1109/ARES.2015.47. URL: https://doi.org/10.1109/ARES.2015.47 (cit. on pp. 11, 74).

[Thea] The Linux Kernel Organization: *The Definitive KVM (Kernel-based Virtual Machine) API Documentation*. https://www.kernel.org/doc/Documentation/virtual/kvm/api.txt. *Last access: September 4, 2019* (cit. on p. 150).

[Theb] The Volatility Foundation: *Open Source Memory Forensics*. http://www.volatilityfoundation.org/releases. *Last access: September 4, 2019* (cit. on pp. 77, 136).

[Thec] The Volatility Foundation: *volatilityfoundation/volatility – Linux*. https://github.com/volatilityfoundation/volatility/wiki/Linux. *Last access: September 4, 2019* (cit. on p. 77).

[Thi10] Thing, Vrizlynn L. L., Kian-Yong Ng, and Ee-Chien Chang: 'Live Memory Forensics of Mobile Phones'. *Digital Investigation*. Vol. 7. Amsterdam, The Netherlands: Elsevier Science Publishers B. V., Aug. 2010: pp. 74–82. doi: 10.1016/j.diin.2010.05.010. url: http://dx.doi.org/10.1016/j.diin.2010.05.010 (cit. on p. 76).

[Tru] Trusted Computing Group and others: *TPM Main Specification Version 1.2 Rev. 116*. https://trustedcomputinggroup.org/resource/tpm-main-specification. *Last access: September 4, 2019* (cit. on p. 2).

[Ubu] Ubuntu: *AppArmor*. https://help.ubuntu.com/community/AppArmor. *Last access: September 4, 2019* (cit. on p. 29).

[UL a] UL Benchmarks: *3DMark Android Benchmark*. https://benchmarks.ul.com/3dmark-android. *Last access: September 4, 2019* (cit. on p. 45).

[UL b] UL Benchmarks: *PCMark for Android - A Better Benchmark for Smartphones and Tablets*. https://benchmarks.ul.com/pcmark-android. *Last access: September 4, 2019* (cit. on p. 45).

[Vel17] Velten, Michael: 'Hardware-based Integrity Protection combined with Continuous User Verification in Virtualized Systems'. https://mediatum.ub.tum.de/1359952. Dissertation. Technical University of Munich, 2017 (cit. on p. 2).

[VMw] VMware: *VMSA-2017-0006: VMware ESXi, Workstation and Fusion Updates Address Critical and Moderate Security Issues*. https://www.vmware.com/security/advisories/VMSA-2017-0006.html. *Last access: September 4, 2019* (cit. on p. 135).

[Wag18] Wagner, Steffen: 'Implicit Remote Attestation of Microkernel-based Embedded Systems'. https://mediatum.ub.tum.de/1400024. Dissertation. Technical University of Munich, 2018 (cit. on p. 2).

[Wag] Wagner, Steffen, Sergej Proskurin, and Tamas Bakos: *TPM 2.0 Simulator Extraction Script*. https://github.com/stwagnr/tpm2simulator. *Last access: September 4, 2019* (cit. on p. 70).

[Wei12] Weinmann, Ralf-Philipp: 'Baseband Attacks: Remote Exploitation of Memory Corruptions in Cellular Protocol Stacks'. *Proceedings of the 6th USENIX Conference on Offensive Technologies*. WOOT'12. Bellevue, WA: USENIX Association, 2012: pp. 12–21. url: http://dl.acm.org/citation.cfm?id=2372399.2372402 (cit. on pp. 3, 76, 106).

[Wei16] Weiss, Michael: 'System Architectures to Improve Trust, Integrity and Resilience of Embedded Systems'. https://mediatum.ub.tum.de/1295004. Dissertation. Technical University of Munich, 2016 (cit. on p. 2).

[Wer19] Werner, Jan, Joshua Mason, Manos Antonakakis, Michalis Polychronakis, and Fabian Monrose: 'The SEVerESt Of Them All: Inference Attacks Against Secure Virtual Enclaves'. *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security, AsiaCCS 2019, Auckland, New Zealand, July 09-12, 2019*. 2019: pp. 73–85. doi: 10.1145/3321705.3329820. url: https://doi.org/10.1145/3321705.3329820 (cit. on pp. 138, 157).

[Wes15] Wessel, Sascha, Manuel Huber, Frederic Stumpf, and Claudia Eckert: 'Improving Mobile Device Security with Operating System-level Virtualization'. *Computers & Security*. Vol. 52. C. Oxford, UK: Elsevier Advanced Technology Publications, July 2015: pp. 207–220. doi: 10.1016/j.cose.2015.02.005. url: https://doi.org/10.1016/j.cose.2015.02.005 (cit. on pp. 6, 7, 26, 160).

[Wes13] Wessel, Sascha, Frederic Stumpf, Ilja Herdt, and Claudia Eckert: 'Improving Mobile Device Security with Operating System-Level Virtualization'. *Security and Privacy Protection in Information Processing Systems: 28th IFIP TC 11 International Conference*. Ed. by Janczewski, Lech J., Henry B. Wolfe, and Sujeet Shenoi. Vol. 405. IFIP Advances in Information and Communication Technology. Berlin Heidelberg: Springer Berlin Heidelberg, 2013: pp. 148–161. isbn: 978-3-642-39217-7. doi: 10.1007/978-3-642-39218-4_12. url: http://dx.doi.org/10.1007/978-3-642-39218-4_12 (cit. on pp. 6, 26, 28, 30).

[Wu14] Wu, Chiachih, Yajin Zhou, Kunal Patel, Zhenkai Liang, and Xuxian Jiang: 'Airbag: Boosting Smartphone Resistance to Malware Infection'. *Proceedings of the Network and Distributed System Security Symposium*. 2014. isbn: 1-891562-35-5. doi: 10.14722/ndss.2014.23164 (cit. on p. 28).

[Wu18] Wu, Yuming, Yutao Liu, Ruifeng Liu, Haibo Chen, Binyu Zang, and Haibing Guan: 'Comprehensive VM Protection Against Untrusted Hypervisor Through Retrofitted AMD Memory Encryption'. *IEEE International Symposium on High Performance Computer Architecture (HPCA)*. Feb. 2018: pp. 441–453. doi: 10.1109/HPCA.2018.00045 (cit. on p. 138).

[Wür16] Würstlein, Alexander, Michael Gernoth, Johannes Götzfried, and Tilo Müller: 'Exzess: Hardware-Based RAM Encryption Against Physical Memory Disclosure'. *Proceedings of the 29th International Conference on Architecture of Computing Systems*. Vol. 9637. ARCS '16. New York, NY, USA: Springer-Verlag New York, Inc., 2016: pp. 60–71. isbn: 978-3-319-30694-0. doi: 10.1007/978-3-319-30695-7_5. url: http://dx.doi.org/10.1007/978-3-319-30695-7_5 (cit. on p. 96).

[Xav13] Xavier, Miguel G., Marcelo V. Neves, Fabio D. Rossi, Tiago C. Ferreto, Timoteo Lange, and Cesar A. F. De Rose: 'Performance Evaluation of Container-Based Virtualization for High Performance Computing Environments'. *Proceedings of the 2013 21st Euromicro International Conference*

on Parallel, Distributed, and Network-Based Processing. PDP '13. Washington, DC, USA: IEEE Computer Society, 2013: pp. 233–240. ISBN: 978-0-7695-4939-2. DOI: 10.1109/PDP.2013.41. URL: http://dx.doi.org/10.1109/PDP.2013.41 (cit. on p. 27).

[XDA] XDA Developers: *What is the Samsung Anyway Jig?* https://www.xda-developers.com/what-is-the-samsung-anyway-jig. *Last access: September 4, 2019* (cit. on p. 84).

[Xen] Xenproject.org Security Team: *x86: Broken Check in memory_exchange() Permits PV Guest Breakout*. https://xenbits.xen.org/xsa/advisory-212.html. *Last access: September 4, 2019* (cit. on p. 135).

[Xia15] Xia, Yubin, Yutao Liu, Cheng Tan, Mingyang Ma, Haibing Guan, Binyu Zang, and Haibo Chen: 'TinMan: Eliminating Confidential Mobile Data Exposure with Security Oriented Offloading'. *Proceedings of the Tenth European Conference on Computer Systems*. EuroSys '15. Bordeaux, France: ACM, 2015: 27:1–27:16. ISBN: 978-1-4503-3238-5. DOI: 10.1145/2741948.2741977. URL: http://doi.acm.org/10.1145/2741948.2741977 (cit. on p. 99).

[Xu19] Xu, Meng, Manuel Huber, Zhichuang Sun, Paul England, Marcus Peinado, Sangho Lee, Andrey Marochko, Dennis Mattoon, Rob Spiger, and Stefan Thom: 'Dominance as a New Trusted Computing Primitive for the Internet of Things'. *Proceedings of the 2019 IEEE Symposium on Security and Privacy*. SP '19. San Francisco, CA: IEEE Computer Society, 2019 (cit. on p. 12).

[Yan03] Yang, Jun, Youtao Zhang, and Lan Gao: 'Fast Secure Processor for Inhibiting Software Piracy and Tampering'. *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO 36. Washington, DC, USA: IEEE Computer Society, 2003: pp. 351–360. ISBN: 0-7695-2043-X. URL: http://dl.acm.org/citation.cfm?id=956417.956576 (cit. on p. 96).

[Yit17] Yitbarek, Salessawi Ferede, Misiker Tadesse Aga, Reetuparna Das, and Todd M. Austin: 'Cold Boot Attacks are Still Hot: Security Analysis of Memory Scramblers in Modern Processors'. *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. Feb. 2017: pp. 313–324. DOI: 10.1109/HPCA.2017.10 (cit. on pp. 76, 77).

[Yod] Yoder, Kent: *LUKS support for storing keys in TPM NVRAM*. https://github.com/shpedoikal/tpm-luks. *Last access: September 4, 2019* (cit. on p. 102).

[Zha15] Zhang, Ning, Kun Sun, Wenjing Lou, Y. Thomas Hou, and Sushil Jajodia: 'Now You See Me: Hide and Seek in Physical Address Space'. *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*. ASIA CCS '15. Singapore, Republic of Singapore: ACM, 2015: pp. 321–331. ISBN: 978-1-4503-3245-3. DOI: 10.1145/2714576.2714600. URL: http://doi.acm.org/10.1145/2714576.2714600 (cit. on p. 77).

[Zha16] Zhao, Lianying and Mohammad Mannan: 'Hypnoguard: Protecting Secrets Across Sleep-wake Cycles'. *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. CCS '16. Vienna, Austria: ACM, 2016: pp. 945–957. isbn: 978-1-4503-4139-4. doi: 10.1145/2976749.2978372. url: http://doi.acm.org/10.1145/2976749.2978372 (cit. on p. 99).

[Zho12] Zhou, Yajin and Xuxian Jiang: 'Dissecting Android Malware: Characterization and Evolution'. *Proceedings of the 2012 IEEE Symposium on Security and Privacy*. SP '12. Washington, DC, USA: IEEE Computer Society, 2012: pp. 95–109. isbn: 978-0-7695-4681-0. doi: 10.1109/SP.2012.16. url: http://dx.doi.org/10.1109/SP.2012.16 (cit. on pp. 1, 26, 30).

List of Acronyms

AS . . . . . Address Space
AWT . . . . . Authenticated Watchdog Timer

BMA . . . . . Bare-Metal Application

CA . . . . . Certificate Authority
cgroups . . . . . control groups
CM . . . . . Container Management
COM . . . . . Connector Operation Module
COW . . . . . Copy-On-Write
CPS . . . . . Cyber-Physical Systems
CRL . . . . . Certificate Revocation List
CRTM . . . . . Core Root of Trust for Measurement
CSR . . . . . Certificate Signing Request

DAT . . . . . Dynamic Attribute Token
DDMS . . . . . Dalvik Debugging Monitor Server
DMA . . . . . Direct Memory Access
DRAM . . . . . Dynamic Random Access Memory
DRM . . . . . Digital Rights Management
DTB . . . . . Device Tree Blob

ECDHE . . . . . Elliptic-Curve Diffie-Hellman Ephemeral
eMMC . . . . . embedded Multi-Media Controller

FDE . . . . . Full Disk Encryption
FROST . . . . . Forensic Recovery Of Scrambled Telephones

GCM . . . . . Google Cloud Messaging

GPA . . . . . Guest Physical Address
GPU . . . . . Graphics Processing Unit
GVA . . . . . Guest Virtual Address

HPA . . . . . Host Physical Address
HSM . . . . . Hardware Security Module
HV . . . . . Hypervisor

IDS . . . . . Industrial Data Space
IOMMU . . . . . I/O Memory Management Unit
IoT . . . . . Internet of Things
IPC . . . . . Inter-Process Communication
IV . . . . . Initialization Vector

JTAG . . . . . Joint Test Action Group

KASLR . . . . . Kernel Address Space Layout Randomization
KSM . . . . . Kernel Samepage Merging
KVM . . . . . Kernel-based Virtual Machine

LiME . . . . . Linux Memory Extractor
LSM . . . . . Linux Security Modules
LUKS . . . . . Linux Unified Key Setup

MAD . . . . . Median Absolute Deviation
MDM . . . . . Mobile Device Management
MFA . . . . . Multi-Factor Authentication
MITM . . . . . Man-In-The-Middle
MK . . . . . Multi-Key
MKTME . . . . . Multi-Key TME
MMU . . . . . Memory Management Unit

NFC . . . . . Near Field Communication

OCSP . . . . . Online Certificate Status Protocol
OSGi . . . . . Open Service Gateway Initiative

PBKDF2 . . . . . Password-Based Key Derivation Function 2
PCR . . . . . Platform Configuration Register
PGD . . . . . Page Global Directory
PKI . . . . . Public Key Infrastructure
PMD . . . . . Page Middle Directory
PTE . . . . . Page Table Entry
PUD . . . . . Page Upper Directory
PXE . . . . . Preboot eXecution Environment

RIL . . . . . Radio Interface Layer

SDO . . . . . Sensitive Data Object
SE . . . . . Secure Element
SELinux . . . . . Security-Enhanced Linux
SEV . . . . . Secure Encrypted Virtualization
SEV-ES . . . . . SEV Encrypted State
SGX . . . . . Software Guard Extensions
SLAT . . . . . Second Level Address Translation
SM . . . . . Security Management
SME . . . . . Secure Memory Encryption
SoC . . . . . System on a Chip
SP . . . . . Secure Processor
SSH . . . . . Secure Shell
SSS . . . . . Sanitizable Signature Schemes
SVM . . . . . Secure Virtual Machine

TCB . . . . . Trusted Computing Base
TCG . . . . . Trusted Computing Group
TEE . . . . . Trusted Execution Environment
TESCP . . . . . Trust Ecosystem Secure Communication Protocol
TLB . . . . . Translation Lookaside Buffer
TLS . . . . . Transport Layer Security
TME . . . . . Total Memory Encryption
TPM . . . . . Trusted Platform Module
TSS . . . . . TCG Software Stack

TXT . . . . . Trusted Execution Technology

UART . . . . . Universal Asynchronous Receiver Transmitter

VM . . . . . Virtual Machine
VMA . . . . . Virtual Memory Area
VMCB . . . . . Virtual Machine Control Block
VMI . . . . . Virtual Machine Introspection
VPN . . . . . Virtual Private Network

List of Figures

List of Tables

List of Listings