TU WIEN Informatics

Evaluating the Arm TrustZone as an Environment for Rootkits

Analyzing the Impact of a Compromised Secure World

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering und Internet Computing

eingereicht von

Daniel Marth, BSc
Matrikelnummer 01227235

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Thomas Grechenig
Mitwirkung: Florian Fankhauser

Wien, 11. März 2021

Unterschrift Verfasser

Unterschrift Betreuung

TU WIEN Informatics

Evaluating the Arm TrustZone as an Environment for Rootkits

Analyzing the Impact of a Compromised Secure World

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering and Internet Computing

by

Daniel Marth, BSc
Registration Number 01227235

to the Faculty of Informatics

at the TU Wien

Advisor: Thomas Grechenig
Assistance: Florian Fankhauser

Vienna, 11th March, 2021

Signature Author

Signature Advisor

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Evaluating the Arm TrustZone as an Environment for Rootkits

Analyzing the Impact of a Compromised Secure World

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering and Internet Computing

by

Daniel Marth, BSc
Registration Number 01227235

ausgeführt am
Institut für Information Systems Engineering
Forschungsbereich Business Informatics
Forschungsgruppe Industrielle Software
der Fakultät für Informatik der Technischen Universität Wien

Advisor: Thomas Grechenig

Wien, 11th March, 2021

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Daniel Marth, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 11. März 2021

Daniel Marth

Acknowledgements

At this point, I would like to express my gratitude to everyone who supported me in working on this thesis. While this includes many people, I would like to highlight my family and friends but especially my wife. Without her patience and motivation I could not have done it. Thank you for everything!

Finally, I would like to thank the Establishing Security (ESSE) team of the INSO research group for supervising and supporting this work. Special thanks go to Clemens Hlauschek for constantly providing valuable technical and formal feedback.

Kurzfassung

Mobile Geräte wie Smartphones verarbeiten eine hohe Menge an persönlichen und vertraulichen Daten. Um sensible Dienste vor Schadsoftware zu schützen, teilt die Arm TrustZone das Gerät in zwei sogenannte Welten ("Worlds"). Kritische Dienste werden in einer isolierten Laufzeitumgebung mit eigenem Betriebssystem ausgeführt, die "Secure World" genannt wird. Das reguläre Betriebssystem sowie dessen Anwendungen befinden sich in der "Normal World", der Dienste der Secure World zur Verfügung stehen.

Während Speicher der Secure World vor der Normal World geschützt ist, hat die Secure World vollen Zugriff auf den Normal World Speicher. Implementierungen der Arm TrustZone sind herstellerspezifisch und proprietär auf allen für Endnutzer relevanten Geräten. Gleichzeitig wurden Schwachstellen in bedeutenden Implementierungen gefunden.

Zusammenfassend ist die Arm TrustZone isoliert, proprietär, privilegiert, verwundbar und weit verbreitet. Diese Eigenschaften schaffen perfekte Bedingungen für fortgeschrittene Schadsoftware wie Rootkits. Die mögliche Verwendung der Arm TrustZone als Umgebung für Rootkits wurde bereits 2013 vorgeschlagen. Soweit wir wissen wurden seither keine Publikationen oder Implementierungen zu Rootkits, die die Arm TrustZone nutzen, veröffentlicht. Größte Herausforderung für ein Secure World Rootkit ist die fehlende semantische Interpretation des Normal World Speichers. Umsetzung von Rootkit-Funktionen erfordert das Reverse Engineering von Datenstrukturen des Kernels zur Laufzeit. Invarianten werden genutzt, um kompilationsabhängige oder randomisierte Symbol-Adressen zu rekonstruieren.

Diese Arbeit liefert die folgenden Beiträge. 1) Design einer Secure World Rootkit Architektur. 2) Prototypische Implementierung von Rootkit-Funktionen, die mit mehreren aktuellen Versionen von Linux als Normal World Betriebssystem kompatibel sind und grundlegende Sicherheitsmaßnahmen umgehen können. 3) Diskussion über Schutzmaßnahmen, um die Normal World vor Schadsoftware in der Secure World zu beschützen.

Die Rekonstruktion von internen Strukturen des Kernels hängt von der zugrundeliegenden Implementierung ab. Linux ist ein aktiv entwickeltes Projekt, daher können sich Strukturen des Kernels im Laufe der Zeit verändern. Kleinere Änderungen des Quellcodes können vom Rootkit kompensiert werden. Stabilität des Rootkits wird experimentell durch Tests mit verschiedenen Versionen des Linux-Kernels bewiesen.

Keywords: Arm TrustZone, Rootkit, Reverse Engineering, Speichermanipulation

Abstract

Mobile devices such as smartphones carry an increased amount of personal and confidential data. In order to protect sensitive services from malware, the Arm TrustZone logically divides the device into two so-called “worlds”. Critical services are running in an isolated execution environment called “secure world” which has its own operating system (OS). The regular OS and its applications are located in the “normal world” and can use services provided by the secure world.

While the secure world memory is protected from the normal world, the secure world has full access to the normal world memory. Implementations of the Arm TrustZone are specific to the vendor and proprietary on currently relevant consumer devices. At the same time, security vulnerabilities have been discovered in all major implementations.

Summarizing, the Arm TrustZone is isolated, proprietary, privileged, vulnerable and widespread. These properties are perfect preconditions for hosting advanced malware such as rootkits. Usage of the Arm TrustZone as an environment for rootkits has been suggested already back in 2013. Since then, no publications or implementations of rootkits utilizing the Arm TrustZone were presented to the best of our knowledge. Major challenge for a secure world rootkit is that there is no semantic interpretation of the normal world memory available. Reverse engineering of kernel data structures at runtime is required to implement rootkit features. Invariants are used to reconstruct compilation-dependent or randomized symbol addresses.

This work makes the following contributions. 1) Design of a rootkit architecture utilizing the secure world. 2) Proof-of-concept implementation of rootkit functions supporting multiple recent Linux kernel versions as normal world OS and circumventing basic protection mechanisms. 3) Discussion of defensive techniques protecting the normal world from malware running in the secure world.

Reconstructing the internal structures of the kernel depends on the underlying implementation. Linux is an actively developed project, thus kernel structures potentially change over time. Minor changes in the source code are compensated by the rootkit implementation. Stability of the developed rootkit is proven experimentally by testing it on various versions of the Linux kernel.

Keywords: Arm TrustZone, rootkit, reverse engineering, memory manipulation

Contents

CHAPTER 1

Introduction

Digitization rapidly changed our everyday lives over the past years. Computers became omnipresent and found their way into many professional and private fields. With the rise of smartphones and Internet of Things (IoT) devices, this trend is continued and even accelerated [132]. Social networks and instant messaging applications foster the revelation of private information [86, 87]. Being connected to the physical world via cameras, microphones and other sensors, mobile devices are able to handle not only digital but also physical information [60]. Thus the digital and physical worlds converge with smartphones as their interface [79].

Concluding from the statements above, smartphones carry a growing amount of personal and confidential data. Smartphones are valuable devices that need to be protected from increasingly widespread malicious software [120]. Additionally, some companies allow their employees to bring along and connect their private devices to their work place. This so-called Bring Your Own Device (BYOD) policy makes smartphones an even more attractive target for attackers. Conventional security mechanisms enforced by companies such as firewalls can be bypassed by infecting phones in the private context. Once an infected device enters the company network again, the attack is continued without being affected by security mechanisms blocking attacks from the outside of the network [105].

Attackers usually attempt to gain full control over the infected device to camouflage the installed malware and avoid its removal. Up to date Operating Systems (OSs) allow multiple user accounts on the same hardware [34]. Users can be assigned varying permissions defining their abilities within the OS. For example, a common scenario is an installation containing a single administrative and one or more regular user accounts. The administrative account is allowed to install software and modify all files on the computer. In contrast, a regular user account can only use the already installed applications and modify files belonging to that specific account. Assuming an attack scenario, malicious regular user accounts can make use of bugs and misconfigurations to elevate the effective privileges and execute actions with the permissions of the administrative user. So-called

"rootkits" are a type of malicious software that is installed on the computer of a victim and tries to maintain administrative privileges. Based on a privilege escalation, rootkits provide long-lasting access to the infected machine and hide their presence from the legitimate user. Common techniques used by rootkits to hide themselves include the modification of OS structures and the redirection of execution flow ("hooking") [83].

To effectively protect crucial system components against rootkits, these components are commonly isolated from the conventional OS by making use of specific hardware features [165]. Hardware-assisted Isolated Execution Environments (HIEEs) act on a low hardware level and are equipped with high privileges on the machine. In case of a compromised vendor or a vulnerability in the implementation of the environment, they provide excellent preconditions for the deployment of rootkits [127, 165].

For example, the Intel System Management Mode (SMM) is used to implement platform-specific system control functions such as power management on Intel processors [165]. Proof-of-concept malware has already shown that it is possible to fully compromise the OS once an attacker has the possibility to execute code in the scope of the Intel SMM [55, 84].

On a different abstraction layer, the situation is similar. The Intel Management Engine (ME) is a platform for running secure services independently of the conventional OS on a separate physical processor. Since the introduction of the Intel ME in 2007, several vulnerabilities within the Intel ME itself as well as its services such as the Intel Advanced Management Technology (AMT) have been published [109, 127, 165]. Additionally, a proof-of-concept rootkit running in the Intel ME has been developed [144].

Intel SMM, Intel ME and Intel AMT were presented as examples for vulnerable HIEEs. As technologies of the chip manufacturer Intel, they are pervasive on desktop computers, laptops and servers. However, smartphones have different requirements and processors conforming to the Arm architecture are reported to account for 90% of all mobile application processors [10].

Similar to the Intel ME, modern Arm processors commonly support an isolation concept called "TrustZone" to improve security. Next to the normal execution environment the user controls ("normal world"), the TrustZone provides a protected execution environment ("secure world"). Although both worlds are running on the same physical processor, interactions and switches between them are strictly controlled by the hardware [125].

A processor supporting Arm TrustZone is able to run a minimal OS in the secure world that offers services to the normal world. These services are called "Trusted Applications (TAs)" and represent the sole way of accessing the secure world from the normal world. Thus, confidential data and algorithms may be utilized without being directly accessible to the normal world potentially running malicious software. TAs are, for example, used to handle sensitive user data such as passwords or support Digital Rights Management (DRM) [51].

According to the specification of the Armv8-A processor architecture, the secure world

has access to the normal world address space [9]. Existing publications pointed out that this property may be abused by malicious software, but described the resulting possibilities only rudimentarily [101, 124]. As already noted, comparable technologies were proven to support the installation of powerful rootkits.

Several security mechanisms are in place to protect the TrustZone from running unauthenticated OSs or TAs [65]. Still, a compromised vendor or an actively exploited security vulnerability could enable malicious software to be executed in the secure world. Existing reports showed that it is possible to reverse engineer proprietary implementations of the TrustZone and exploit vulnerabilities to execute arbitrary code in the context of the secure world OS on real-world devices [28, 29, 42, 59, 119]. Due to that, the capabilities of the Arm TrustZone need to be scrutinized to evaluate the impact of a compromised secure world.

1.1 Expected Results

In this thesis, we will provide an implementation of a novel state-of-the-art Arm TrustZone rootkit to show the risk for the normal world if malicious software is running in the secure world. Latest research in the field of rootkits [20, 39, 50, 74, 156, 161, 165, 166, 167, 168, 169, 172] is the basis for this new kind of malicious software. QEMU [24], an open-source hardware emulator, is the initial target environment for the rootkit. The normal world environment chosen for the development is powered by a Linux-based OS.

Proof-of-concept implementations provided by this thesis show mechanisms for code resting in the secure world to interact with the normal world via memory manipulation. Especially the inspection and modification of the normal world OS and its protection mechanisms is relevant. Typical rootkit functionalities based on these interaction primitives are implemented in the secure world. Basic randomization protections of the normal world kernel are evaded. Compatibility across multiple versions of the Linux kernel is ensured without having the respective source code available.

Defensive mechanisms protecting from Arm TrustZone rootkits are identified and their effectiveness is discussed. There do not exist any protective measures specifically for malware running in the Arm TrustZone to the best of our knowledge. Instead, general mechanisms protecting the normal world kernel are analyzed.

Summarizing, the following research questions will be answered by this thesis:

- What possibilities open up to malicious software in the secure world by manipulating normal world memory?
- Which invariants are necessary to support targeted memory manipulations without relying on the exact version and compilation configuration of the normal world OS?
- How can a rootkit tailored for the Arm TrustZone be structured?

- How effective are existing defensive mechanisms against a rootkit running in the secure world?

1.2 Methodological Approach

To build the thesis on top of a solid theoretical foundation, an academic literature research is conducted first. Related work and existing academic projects are analyzed to identify relevant concepts for the implementations. Modern rootkit design and defensive mechanisms are central points of this research step. After the academic background is established, publicly available industrial work is surveyed.

As there is no semantic interpretation of the normal world memory content available to the secure world, internals of the normal world Linux kernel at runtime need to be reverse engineered. Concepts such as page tables and processes need to be detected and interpreted without support from the kernel. Invariants identified based on the implementation of the Linux kernel serve as the foundation for this process.

Candidates for invariants include the verification of references by simulation of address resolution, limitations on feasible values and observation of intentionally triggered state modifications. Stability of these methods is to be evaluated during this work by analysis of the respective kernel source code and the frequency of changes applied to it.

Reverse engineering of memory at runtime is tightly coupled to the specific kernel implementation, which naturally changes over time. To ensure assumptions about kernel internals are chosen general enough to endure minor implementation changes, experiments with recent kernel releases are performed. For each kernel release to be tested, the implemented functionality is tested and the result of the execution interpreted.

Before starting with the actual practical research, the development environment needs to be set up. OP-TEE [112], a portable and open implementation of a secure world OS, is the basis for all further developments within this thesis. Like a normal world OS, it abstracts low-level hardware details and provides a stable runtime environment. Using OP-TEE as a simplification layer helps to focus on the relevant parts of this work.

Development of a proof-of-concept rootkit happens on top of a local copy of the existing OP-TEE code. The rootkit is able to make use of all functionality the OP-TEE secure world OS contains. Instead of using Arm TrustZone hardware, QEMU serves as a virtual environment to explore the possibilities of the TrustZone. Inspection of arbitrary physical addresses is possible with QEMU. A custom normal world kernel module printing internal information helps to understand kernel structure and verify the correctness of invariants during development of the rootkit. As the C programming language was the primary technology used to create OP-TEE and also the Linux kernel, it is the language of choice for the rootkit created in the scope of this thesis.

Protective measures against Arm TrustZone rootkits are derived from well-known rootkit defense mechanisms. Major goal of the protection techniques is to diagnose anomalies in

the execution of instructions or integrity violations of the memory content due to the presence of a rootkit in the TrustZone. Additionally, randomization techniques applied by the kernel increase the complexity of reverse engineering kernel structures. Once the implementation of the rootkit is finished, the effectiveness of the identified defensive techniques and possible improvements are discussed theoretically.

CHAPTER 2

Related Work

All of the academic publications below address topics relevant for the implementation of rootkits and the discussion of defensive measures against them. However, no implementation of a rootkit on the basis of the Arm TrustZone was published to the best of our knowledge.

2.1 Weaknesses of Hardware-assisted Isolated Execution Environments

To defend the normal world from a compromised secure world, it is essential to know weaknesses of HIEEs like the Arm TrustZone.

Vulnerabilities in secure world OSs and TAs have been published by various sources. Cerdeira et al. [42] analyzed vulnerability reports of all major commercial Trusted Execution Environments (TEEs). Protection mechanisms like Address Space Layout Randomization (ASLR) and stack canaries taken for granted in the normal world were found to be implemented insufficiently or missing in most secure world implementations. Defenses were suggested that help to mitigate the identified architectural issues. “Bits, Please!” [26] is an online blog covering the topics of reverse engineering and exploiting Qualcomm’s TrustZone implementation [27, 28, 29]. Furthermore, an attack on the normal world Linux kernel is demonstrated [30]. Rosenberg [123] exploited an integer overflow vulnerability on Qualcomm-based devices to write to arbitrary locations in the secure memory. Sanfelix [129] pointed out insufficient security measures. Multiple exploits for vulnerabilities in TAs were described. Shen [135] developed two exploits to execute arbitrary code in the context of the Huawei TEE and ultimately read images from a smartphone fingerprint sensor. Komaromy [91] created a blog series about reverse engineering and exploiting Samsung’s TrustZone implementation.

Zhang et al. [168] accomplished a timing-based cache side-channel attack on the Arm TrustZone in 2016. Secret information from the secure world was recovered by exploiting cache contention between the normal world and the secure world. Attacks from the normal world kernel as well as an Android application were proposed.

In 2018, Zhang et al. [169] showed how features of the x86 architecture can be misused to sabotage memory forensics. The developed prototype manipulates the physical address space to conceal data. Another feature presented is the protection of malicious code by running it inside an Intel Software Guard Extensions (SGX) enclave.

Ryan [128] published a caching side-channel attack on the TrustZone. Unknown Elliptic Curve Digital Signature Algorithm (ECDSA) private keys stored in Qualcomm's TrustZone keystore can be extracted by a normal world kernel module.

Machiry et al. [101] introduced a vulnerability class called "BOOMERANG" that abuses the capabilities of the Arm TrustZone to read and write arbitrary memory locations. BOOMERANG leverages the Arm TrustZone to allow untrusted applications to steal sensitive data from other applications, bypass security checks or gain full control of the normal world OS. Several implementations by different vendors were evaluated and found to be vulnerable.

Fleischner et al. [59] evaluated the exploitability of memory-safety violations inside TEEs. OP-TEE was used as basis for their case study, extended with vulnerable examples inspired by real-world exploits seen in the wild.

2.2 Rootkit Implementations

Concerning the implementation of the rootkit, proof-of-concept models for other environments than the Arm TrustZone exist.

2.2.1 Arm Rootkits

Rootkits targeting Arm Central Processing Units (CPUs) without relying on the TrustZone are available.

David et al. [49] developed "Cloaker", a non-persistent rootkit for Arm processors. Hardware state modifications are used for concealment and operation. Additionally, a framework for checking the integrity of the state of hardware devices was designed and implemented.

Buhren et al. [39] proposed a hypervisor-based Arm rootkit that moves the victim OS into a Virtual Machine (VM). Because the hypervisor is running on a higher privilege level than the OS, their rootkit is difficult to detect and remove.

Zhang et al. [171] designed and implemented a technique for gaining administrative permissions on Android devices using features of the hypervisor. In contrast to the work of Buhren et al. [39], this work requires the hypervisor to be initially absent.

Bickford et al. [25] demonstrated the possibilities of smartphone rootkits. A Neo Freerunner smartphone with an Arm processor [108] was used for this research.

Zhang et al. [166] presented a way to evade introspection from the secure world using the processor cache and implemented "CacheKit" as a proof-of-concept rootkit.

None of the above rootkits make use of the TrustZone, in contrast to this thesis.

2.2.2 Rootkits Targeting Non-TrustZone Hardware-assisted Isolated Execution Environments

In contrast to the Arm TrustZone, comparable mechanisms of the x86 architecture have been target of further research.

Embleton et al. [55, 56] implemented a rootkit based on the Intel SMM. Proof-of-concept implementations for a chipset level keylogger and network backdoor directly interfacing with the network card were provided.

Schiffman and Kaplan [133] presented an approach to hijack Universal Serial Bus (USB) host controllers by running malware in x86's SMM. A respective USB keylogger was created as proof-of-concept.

King et al. [88] made use of virtualization for their "SubVirt" rootkit. By manipulating the boot sequence, the rootkit runs before the legitimate OS and hoists it to a VM.

Zhang et al. [169] used the Intel SGX technology to protect the secret key of a custom ransomware implementation.

Schwarz et al. [134] implemented an Intel SGX enclave malware which fully and stealthily impersonates its host application.

Canella et al. [41] found a way to detect physically-backed kernel addresses and break Kernel Address Space Layout Randomization (KASLR).

2.3 Rootkit Detection and Prevention

No rootkit implementations for the Arm TrustZone are publicly known at the time of writing. Consequentially, there are also no defensive techniques specific to this type of rootkit. Techniques to detect and prevent rootkits relying on other technologies have been proposed in the past and serve as an inspiration for the discussion of potential protection mechanisms targeting rootkits utilizing the Arm TrustZone.

2.3.1 TrustZone-based Dynamic Rootkit Detection and Prevention

The Arm TrustZone has already found applications in forensics and the protection against normal world rootkits. Due to the control these approaches have over the normal world, they serve as an inspiration for the implementation of a rootkit residing in the TrustZone.

Sun et al. [145] showed that it is possible to use the Arm TrustZone for conducting forensic operations on the normal world memory and registers even if the normal world OS crashes or is compromised.

"CacheKit", a rootkit already described in Section 2.2.1, was later defeated by "CacheLight" developed by Gutteerenz et al. [74]. A secure world service prevents malicious use of cache locking mechanisms.

Ge et al. [63] as well as Azab et al. [19] described mechanisms that leverage the Arm TrustZone for introspecting and protecting the normal world kernel from rootkits.

Guan et al. [73] implemented "TrustShadow", a TrustZone-based defensive system that shields applications from a compromised OS.

Zhang et al. [167] make use of the processor cache and encryption of application memory to prevent cold boot attacks on the TrustZone.

Jiang et al. [82] developed "LKRDet", a rootkit detection framework targeted at IoT devices running in the Arm TrustZone. Hardware Performance Counterss (HPCs) are utilized to spot deviations between executions in a clean and a compromised environment to find evidence for normal world kernel rootkits.

Brasser et al. [36] proposed the TrustZone-backed "SANCTUARY", which enables the creation of isolated compartments in the normal world.

2.3.2 Non-TrustZone Dynamic Rootkit Detection and Prevention

Non-TrustZone hardware and virtualization features have also been used to identify rootkits or for forensics in general. These publications present interesting ideas for defending the normal world against secure world rootkits.

Xiao et al. [161] implemented a VM introspection tool called "HyperLink" designed to do a partial reconstruction of the OS state without having the relevant source code available. Invariants are used to recover parts of the state from memory.

Zhang et al. [170] built an Intrusion Detection System (IDS) using an IBM secure coprocessor.

Wang and Karri [156] showed that hardware performance metrics can be used to detect and identify rootkits.

Grill et al. [71] developed a framework for detecting, analyzing and preventing bootkits based on virtualization.

Dawson et al. [50] use power supply voltage measurements as a side-channel that can not be tampered with or spoofed.

Zhou and Makris [172] proposed a hardware-assisted rootkit detection system which can not be compromised through software attacks. Machine learning helps to identify malicious process behavior.

Wang et al. [155] showed how to use Intel SMM to securely acquire and transmit the full state of a protected machine to a remote server where its integrity can be verified.

Xiong et al. [162] presented a hypervisor-based integrity protection system that confines the behavior of untrusted kernel extensions.

Pendergrass and McGill [116] verify consistency of critical kernel data structures at runtime.

Gruss et al. [72] proposed Kernel Address Isolation to have Side channels Efficiently Removed (KAISER) (meanwhile called Kernel Page-Table Isolation (KPTI)), a system that eliminates microarchitectural side-channel attacks on kernel address information.

2.3.3 Static Rootkit Detection

Previous sections listed attempts to dynamically detect rootkits at runtime. Another option is to analyze binaries statically.

Kruegel et al. [92] presented a technique that identifies instruction sequences that are an indication of rootkits. Symbolic execution is used to simulate the execution of kernel modules.

Musavi and Kharrazi [107] try to classify drivers by looking at various metrics of the drivers' disassembled code. Obfuscation of drivers is emphasized to be a valuable indicator for the presence of a rootkit.

CHAPTER 3

Computer Security Basics

A general understanding of computer security basics is essential to understand the background and implications of this thesis. This chapter provides the necessary definitions and connections to establish this foundation. Well-known standards and definitions are used during the explanations.

3.1 Definition of Computer Security

As a starting point, the term "computer security" itself needs to be defined. Several definitions are available, but this work will rely only on the National Institute of Standards and Technology (NIST) as a source.

The definition contained in the "Glossary of key information security terms, NISTIR 7298 Rev. 2" is included below.

> Measures and controls that ensure confidentiality, integrity, and availability of information system assets including hardware, software, firmware, and information being processed, stored, and communicated. (Definition of "computer security" provided by the NIST [89])

Within this definition, the security attributes confidentiality, integrity and availability are introduced, which are commonly summarized as the "CIA triad" [69, 142]. While the triad is a well-known concept and widely used in the field of computer security, it is often extended by additional properties. Most notably authenticity and accountability complement the triad as security attributes [142]. Figure 3.1 visualizes the essential computer security attributes considered in this work.

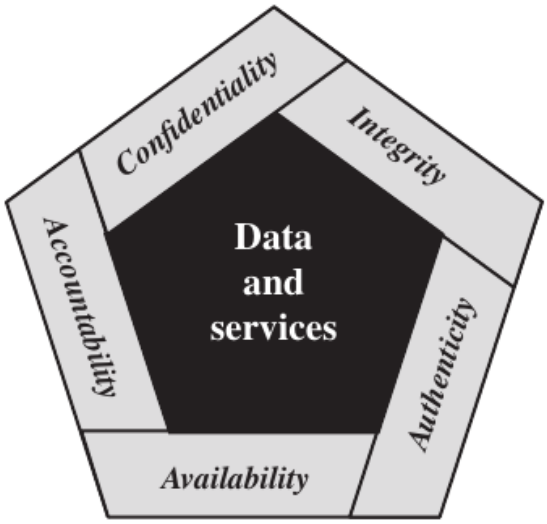

Figure 3.1: Essential computer security attributes by Stallings and Brown [142].

3.2 Security Attribute Triad

Section 3.1 provided a definition for the term "computer security". Further explanations of the security attributes confidentiality, integrity and availability are given in this section.

3.2.1 Confidentiality

Confidentiality is a property that forbids the unauthorized disclosure of information [69, 89].

A computer system needs to implement proper access controls to verify whether an entity is allowed to access information. To do so, there must be a possibility to identify an entity and its corresponding permissions [69].

Encryption is another powerful technique to ensure confidentiality of information. During the encryption process, data is transformed using a secret key. This key is only known by authorized entities. Unauthorized entities do not have access to the key and thus can not access the data [69].

An example for an attack on confidentiality is an eavesdropper between two communicating parties ("man-in-the-middle attack") [142].

3.2.2 Integrity

Integrity is a property that forbids the unauthorized modification or destruction of information [89].

Cryptographic functions such as checksums and digital signatures are one possibility to provide integrity [69].

In case a man-in-the-middle actively alters communication between two parties, data integrity is violated [142].

3.2.3 Availability

While the previous attributes confidentiality and integrity forbid unauthorized entities to perform actions on information, availability requires information to be accessible and modifiable by authorized entities in a reliable and timely fashion [89].

Redundancy of software as well as hardware within the system improves availability [69].

One technique attackers use to cut availability of computer systems are Distributed Denial of Service (DDoS) attacks [142].

3.3 Further Security Attributes

Authenticity and accountability are common additions to the triad of security attributes [4]. Below follows an explanation of these properties.

3.3.1 Authenticity

Authenticity is a property of being genuine and to be able to be verified [89].

3.3.2 Accountability

Accountability is a property that requires actions to be traced back to the executing entity [89].

CHAPTER 4

Fundamentals of our Arm TrustZone Rootkit

Complementary to the general basics contained in Chapter 3, this chapter explains technical concepts and affected technologies.

4.1 Arm Processor Architecture

Arm (previously written "ARM") processors follow specifications developed by Arm Holdings [6] that are sold as Intellectual Property (IP). Chip manufacturers need to buy the respective licenses from Arm Holdings in order to produce Arm processors. This section gives technical details about the Arm architecture including the concept of the Arm TrustZone.

4.1.1 Instruction Set Architecture

Mobile devices are required to use the available resources efficiently in order to achieve adequate performance and a long battery life at the same time. While desktop computers nowadays mainly use processors based on the x86 and AMD64 Complex Instruction Set Computer (CISC) Instruction Set Architectures (ISAs), energy efficient Arm Reduced Instruction Set Computer (RISC) processors are reported to power 90% of all mobile application processors [10].

RISC processors use simple fixed length instructions that take only a single CPU cycle to execute. In contrast, CISC processors support complex instructions (e.g., cryptographic functions) of variable length that take multiple CPU cycles to execute [31].

As of the time of writing, Armv8 is the newest available version of the Arm ISA. It differs significantly from its predecessors by supporting a 64-bit architecture called "AArch64".

However, it is compatible with the 32-bit architecture used in previous versions which is now referred to as "AArch32" [9].

Note that the Linux kernel refers to "AArch32" simply as "ARM", whereas "AArch64" is called "ARM64".

4.1.2 Architecture Profiles

Depending on the intended use case, there are varying requirements for a processor. Sometimes it is necessary to consume as little power as possible, in other cases completely deterministic behavior is desired. Arm architectures have three so-called profiles that share the same instruction set but are backed by specific implementations [8, 45]:

- Application (A)
- Real-Time (R)
- Microcontroller (M)

Figure 4.1 shows the fundamental differences and common use cases of the architecture profiles.

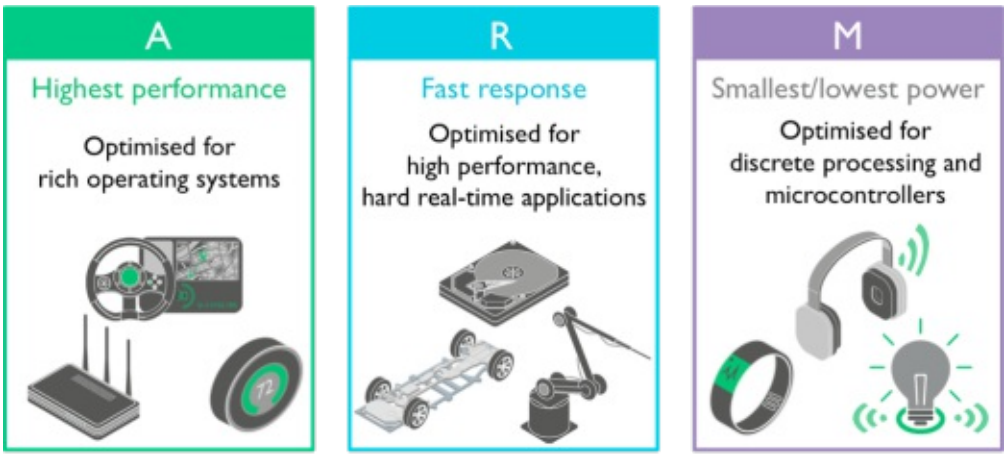

Figure 4.1: Fundamental differences between the architecture profiles [81].

To refer to a profile of a specific architecture, the profile identifier is appended to the architecture name. For example, the application profile of Armv8 is called Armv8-A.

Within the scope of this work, we will focus on Armv8-A without considering a specific microarchitecture.

4.1.3 Exception Levels

Processors conforming to the Armv8 specification running the AArch64 instruction set support different execution modes called Exception Levels (ELs). Processes can move execution to another EL by triggering an exception. There are different instructions to trigger exceptions, depending on the current and target EL. Exceptions can be handled on the same or on a higher EL, but not on a lower one. Exception Vector Tables (EVTs) for each EL contain the instructions of the exception handlers. The exception handler returns to the previous EL via the Exception Return (ERET) instruction [9].

User applications are always executed in EL0, the processor mode with the least privileges. Supervisor Calls (SVCs) allow the user applications running in EL0 to communicate with the OS running in EL1 [9].

EL2 can be used by hypervisors managing VMs. Hypervisor Calls (HVCs) allow the OS running in EL1 to communicate with the hypervisor in EL2 [9].

EL3 contains a component called the "secure monitor". Its task is to handle Secure Monitor Calls (SMCs). Further details about this component are given in Section 4.1.5.

4.1.4 Virtual Memory Management

Modern OSs make use of a mechanism called "virtual memory". Instead of accessing memory locations directly by their physical addresses, each process is simulated to have the whole theoretical address space for itself. Virtual addresses are used to address memory in the virtual memory. The Memory Management Unit (MMU) is responsible for translating memory accesses from virtual addresses to physical addresses [157].

Memory regions are managed in units of pages. Pages have a fixed size corresponding to the memory translation granule size. 4KB, 16KB and 64KB are sizes supported by Armv8 [7, 113].

Mapping between the two address spaces is accomplished with translation tables. Each table has the size of a page. Starting from a single initial page table, multiple levels of translation tables are used to convert virtual addresses to physical ones [113].

Blocks are larger page sizes supported on specific translation levels. Considering a translation granule of 4KB, memory blocks have the following size depending on the translation level [7, 113]:

- Level 1: 1GB
- Level 2: 2MB

Armv8-A effectively uses up to 48 bits for addressing[1]. Bit sections of virtual addresses are actually indices for translation tables and the corresponding page address offset [7].

[1]The Armv8.2-A extension optionally increases this limit to 52 bits [7]. Within the scope of this work this possibility is neglected.

Figure 4.2 shows the structure of the 48 virtual address bits when using the 4KB memory translation granule size.

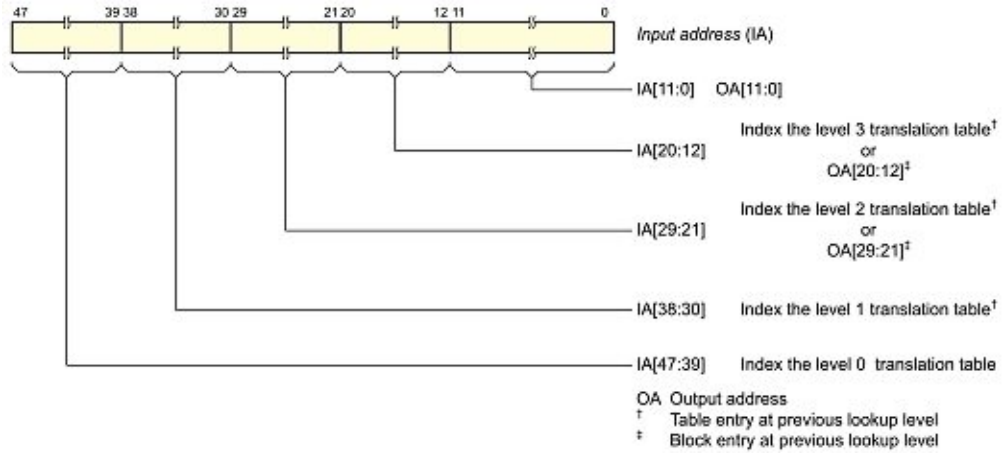

Figure 4.2: Virtual address structure when using the 4KB translation granule size according to the Armv8-A architecture reference manual [7].

Combining the information given above, the overall translation process for the 4KB translation granule is described in Figure 4.3.

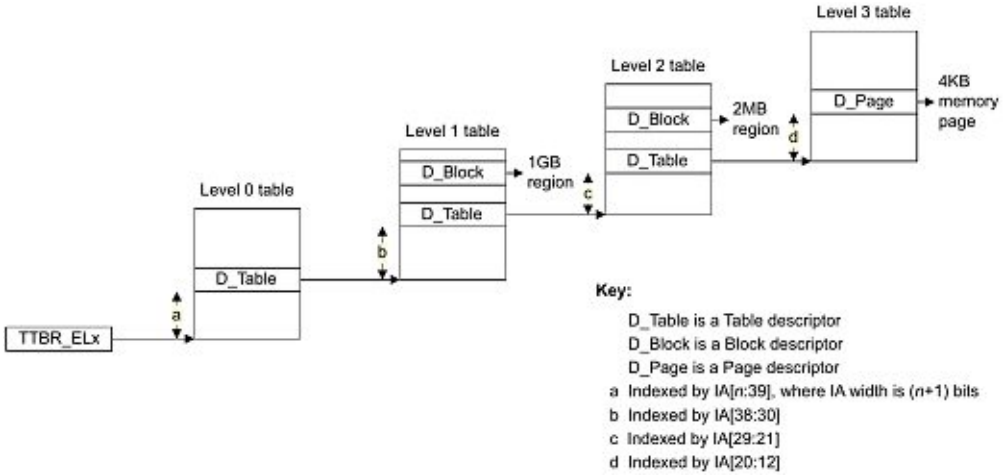

Figure 4.3: Multi-level address translation according to the Armv8-A architecture reference manual. `TTBR_ELx` represents the Translation Table Base Register (TTBR) (base address of the translation table) of ELx [7].

4.1.5 Arm TrustZone Architecture

The Arm TrustZone is a security extension introduced with Armv6 [11]. Below follows an explanation of the most important properties of the Arm TrustZone.

Splitting the Device into Secure World and Normal World

With processors running the Arm TrustZone, the physical machine is split into two sections.

The conventional OS that is generally assumed to run untrusted software is called "normal world" or Rich Execution Environment (REE). Theoretically, the normal world can just ignore the splitting of the physical machine and continue working without any change.

Services that process sensitive data are an attractive goal for attackers. To protect these services, they are run in the so-called "secure world" or TEE where they are managed by a trusted OS in the form of TAs. This trusted OS should offer a minimal attack surface and is running independently of the OS in the normal world.

Analogous to the concept described in Section 4.1.3, the secure world is split into three ELs. Note that EL2, which can run hypervisors in the normal world, is not used by the Arm TrustZone unless the Armv8.4-SecEL2 extension is implemented [7]. Figure 4.4 shows the layout of the system considering ELs and worlds.

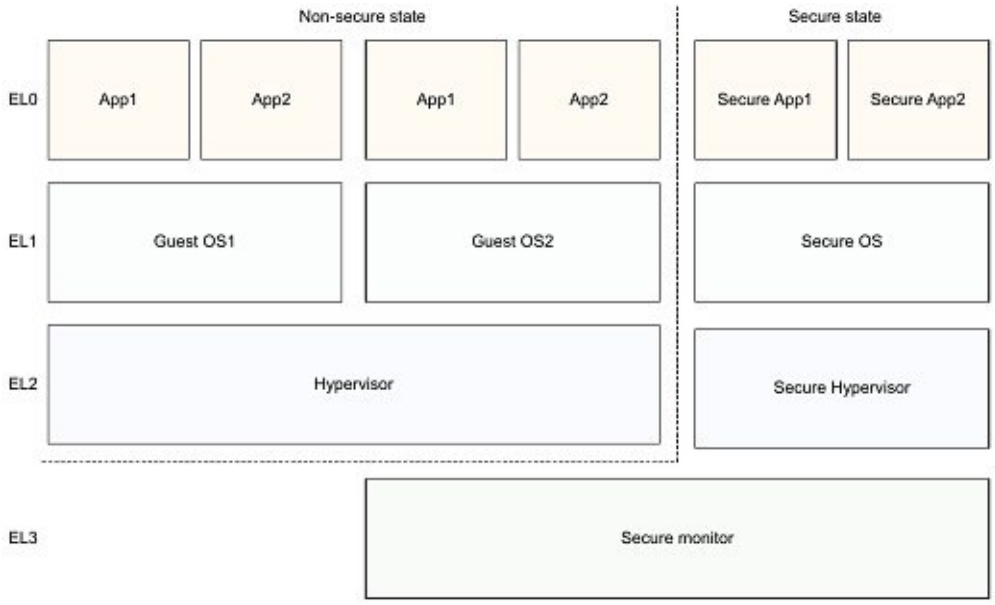

Figure 4.4: Armv8 ELs and their components [7].

Both worlds share the same physical processor. A flag called `NS` ("non-secure") in the Secure Configuration Register (SCR) `SCR_EL3` indicates the current world of the

processor [7]. However, the way the normal world can interact with the secure world is strictly defined by the processor specification.

GlobalPlatform [68], a nonprofit organization that standardizes secure chip technology, additionally administrates a specification about the Arm TrustZone architecture and its internal Application Programming Interface (API) [117]. Though this specification is not enforced by the chip design, it is followed by several implementations as described in the following section.

According to the Arm specification, the Random Access Memory (RAM) regions of both worlds do not need to be physically separated. Different translation tables are used by the MMU to prevent the normal world from accessing the secure world memory pages. As shown in Figure 4.5, code running in the secure world can also add insecure memory pages to its translation tables [9, 11].

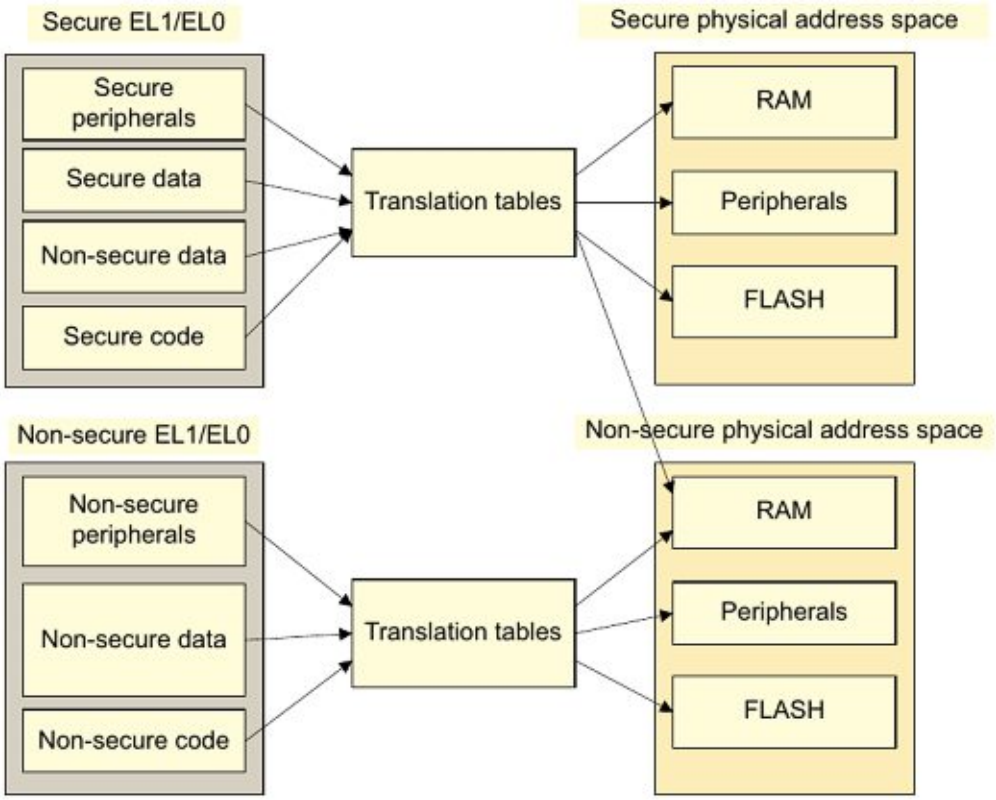

Figure 4.5: Physical address space access restrictions [9].

The secure monitor is the only part of the secure world that is accessible by the normal world and represents the interface between these two worlds. Main task of the monitor is to intercept SMCs, manage the context switch between the worlds and notify the secure world OS appropriately about the call. Apart from hardware Interrupt Requests (IRQs)

and Fast Interrupt Requests (FIQs), the SMC instruction is the only way to invoke the secure world after handing over control to the normal world during the boot process [117].

Overview of Trusted Execution Environments

Various implementations of TEEs exist, a few of them are presented in this section.

OP-TEE [112] is a TEE primarily developed and maintained by Linaro [2, 94] conforming to the GlobalPlatform specification. Although OP-TEE was started as a proprietary project, its code was released as open-source in 2014 [2, 110]. Primary contribution of the OP-TEE project is the secure world OS, but it provides a complete configuration to build and run a usable test system. Several target devices are supported, including a virtual device for Armv8. SierraTEE is an alternative supporting Arm TrustZone as well as the Microprocessor without Interlocked Pipelined Stages (MIPS) architecture [136]. Like OP-TEE, SierraTEE follows the specification from GlobalPlatform. Trusty TEE is a free open source alternative for Android initiated by Google [154].

Considering smartphones, only proprietary TEEs with limited public documentation are of practical relevance. System on a Chip (SoC) manufacturer Qualcomm [160] uses its own implementation called Qualcomm Secure Execution Environment (QSEE). Trustonic [153] develops a TEE called “Kinibi”. Android phone manufacturer Huawei [77] is using its “TrustedCore” as a TEE. Due to the lack of information about proprietary TEEs, their robustness and benevolence can not be independently investigated without a significant reverse engineering effort.

4.2 The Linux Kernel

The Linux kernel (henceforth also referred to as “Linux”) is an open-source Unix-like OS kernel initially developed by Linus Torvalds in 1991 [38]. Due to its openness, Linux is a popular target system for academic research and proof-of-concept implementations [39, 49, 74, 82, 166]. Because it constitutes the normal world OS in this work, knowledge about the internals of Linux is crucial for understanding the developed rootkit. Therefore, an introduction to the relevant components and mechanisms is provided.

4.2.1 Booting Linux on TrustZone-enabled Systems

Before an OS can be started, several steps are executed as part of the boot process. Details vary between architectures, focus of this section is on the boot process of systems with Arm TrustZone. Specifically, the OP-TEE configuration on the Armv8 architecture running in QEMU which is used for this work is described. Figure 4.6 gives a high-level overview of the typical boot procedure on systems using Arm TrustZone.

A reference implementation of the secure world firmware for the application profile is available as an open-source project called “Trusted Firmware-A” [12]. Significant part of that project are the various boot loaders. Acting on multiple sequential stages, these

boot loaders are responsible for initializing the system and preparing the environment for the respective next stage. Exact functionality of the boot loaders is out of scope of this section, but documentation is provided by the Trusted Firmware-A project [58].

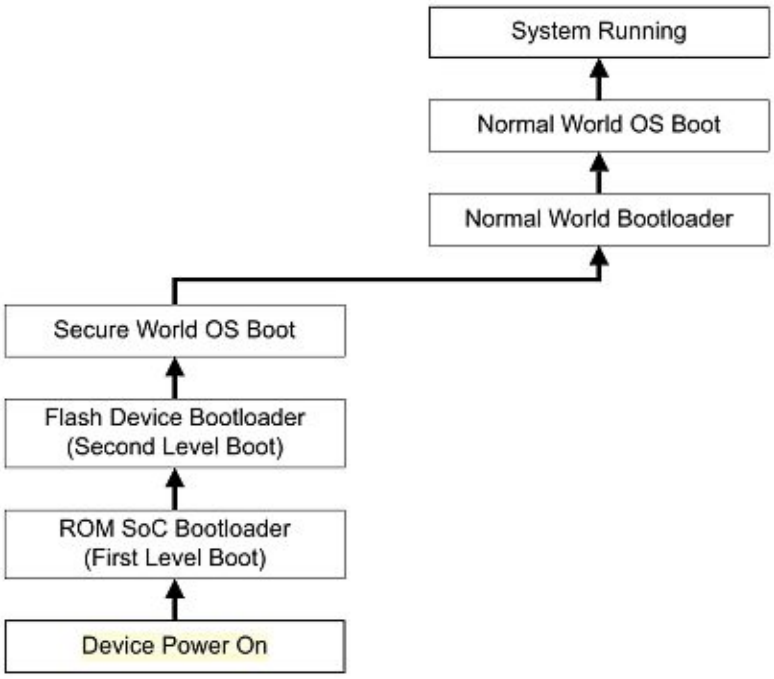

Figure 4.6: Typical boot sequence on a system using Arm TrustZone as described by the Arm documentation [11].

Upon start, the CPU executes in the secure world. By being started before the normal world, it is ensured that the secure world can first run in an environment untampered by the normal world. Once the secure world OS (e.g., OP-TEE) finished its initialization, execution is continued in the normal world [11].

EFI Developer Kit II (EDK II) [151], an open-source implementation of the Unified Extensible Firmware Interface (UEFI) specification, is used to generate the normal world firmware [54]. As normal world boot loader, it passes execution to the normal world OS [11].

Final component in the boot process is the normal world OS. Normal world boot loaders expect to launch a single binary file generated from the kernel source code and a corresponding configuration. This compiled file is called the image of the kernel [11]. Figure 4.7 by Stallings and Brown further demonstrates this concept.

In general kernel images might be compressed to save storage and speed up the loading procedure. Linux currently does not provide a decompressor in its ARM64 build. Thus, either the boot loader needs to take care of the decompression or an uncompressed image

must be used [33]. Uncompressed UEFI images follow the Portable Executable (PE) / Common Object File Format (COFF) format [115]. With respect to this specification, the presence of the image can be verified by checking the initial "MZ" magic bytes [5].

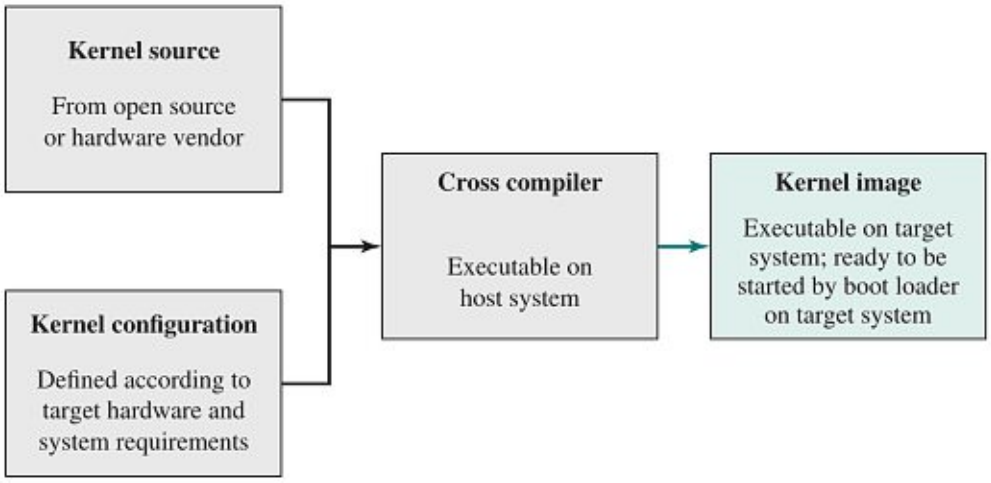

Figure 4.7: Generation of the kernel image as visualized by Stallings and Brown [141].

4.2.2 Linux Process Management

A process or task is a running program that is managed by the kernel [157]. Internally, the kernel uses the `task_struct` structure to keep track of tasks. Many fields are contained in this structure, a few of them shall be described shortly.

- `comm`: Name of the associated executable (limited to 16 characters).
- `pid`: Unix-like systems traditionally assign each process a unique Process Identifier (PID). Linux assigns PIDs starting at 0 and increments them for each new process by one.
- `state`: Current status of the process (e.g., running, dead). Bits of this field represent multiple properties.
- `cred` and `real_cred`: References to credentials of the process determining its permissions.

Table 4.1 lists the state options of the Linux kernel relevant for this work.

Hexadecimal State	Symbolic State
0x0000	TASK_RUNNING
0x0001	TASK_INTERRUPTIBLE
0x0002	TASK_UNINTERRUPTIBLE
0x0020	EXIT_ZOMBIE
0x0080	TASK_DEAD
0x0400	TASK_NOLOAD

Table 4.1: Process state values in their numerical and symbolic form [131].

State options listed in Table 4.1 are a small excerpt of the available values. Furthermore, these options can be combined with bit operations to provide a more detailed description of the task state.

Over the lifetime of a task, it will be assigned different states by the kernel. Transitions between task states are shown in a simplified way in Figure 4.8.

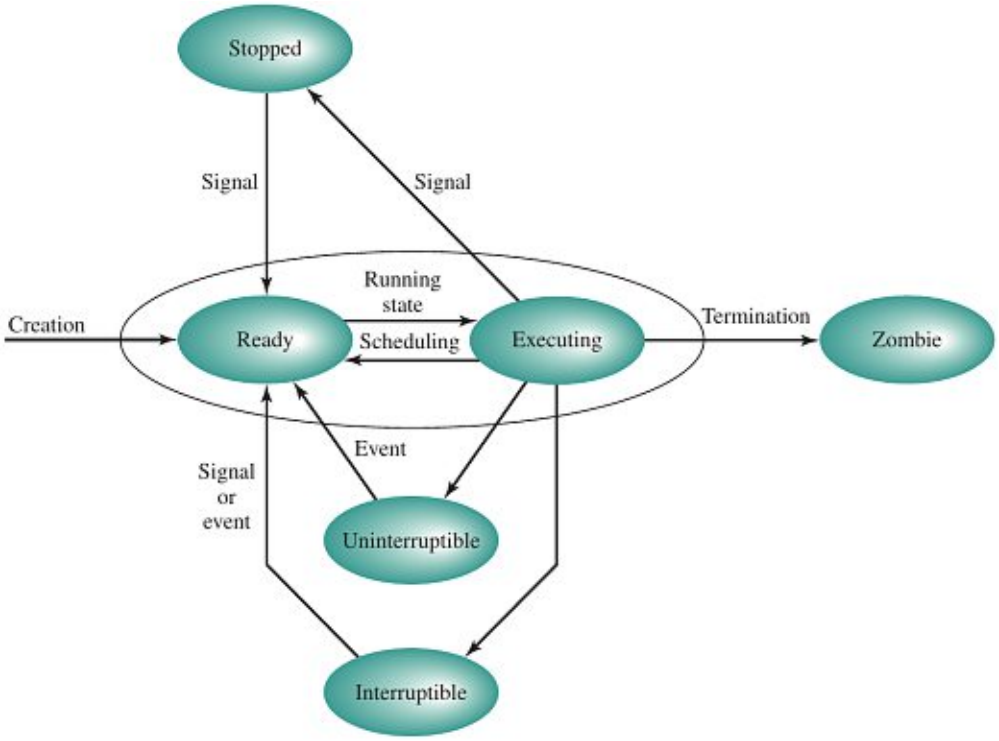

Figure 4.8: Linux process states and transitions illustrated by Stallings and Brown [141].

Multiple tasks running simultaneously on the same machine compete for resources such as CPU time. Execution order is determined by the OS's scheduler component. In case a

process is not scheduled indefinitely although it would be ready for execution, it is starved of CPU time [141]. Figure 4.9 visualizes the general concept of CPU time starvation.

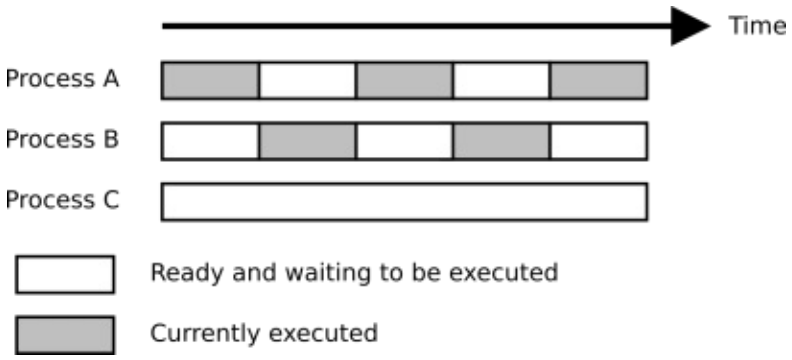

Figure 4.9: Processes A and B are alternately executed. Although ready, process C is not executed at all and starved of CPU time.

The credential fields `cred` and `real_cred` apply the Read-Copy-Update (RCU) synchronization mechanism. Through this pattern, each field can be updated by a single source while still being available for consistent reading operations without further synchronization mechanisms such as locks [104, 149, 158]. Reference counting is used to efficiently handle the allocation of instances [57].

Tasks are managed in a cyclic doubly linked list of `task_struct` instances. A doubly linked list is a data structure that is characterized by distinct elements having a reference to its predecessor and successor [138].

Figure 4.10 visualizes how implementations used in applications usually refer to the beginning of the following element.

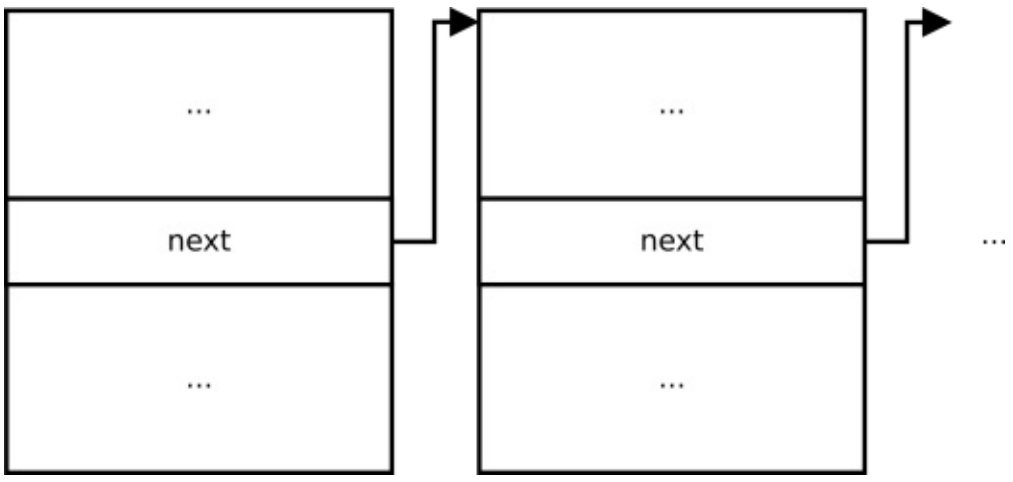

Figure 4.10: Successor reference in lists commonly used in applications.

The implementation of the list data structure within the Linux kernel differs from what is commonly used for regular applications.

Actual list fields in the Linux kernel contain only the pointers to the previous and next list elements and the actual data structure is wrapped around this field. Instead of pointing to the beginning of the referenced element, each element points to the respective list field [43, 161]. Figure 4.11 highlights this important implementation detail for the successor.

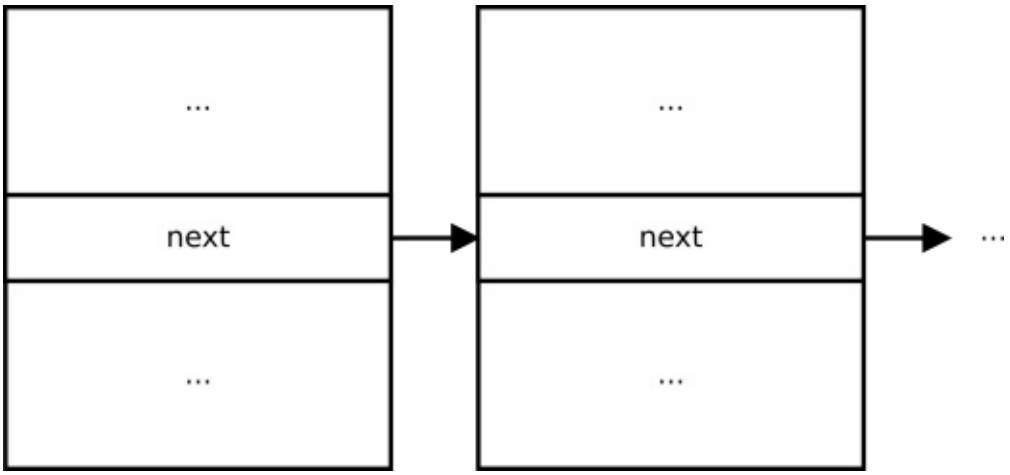

Figure 4.11: Successor reference in lists within the Linux kernel.

Linux kernel lists are cyclic, i.e., successor of the last element in the list is the first element and the predecessor of the first element is the last element in the list [43]. Figure 4.12 visualizes the cyclic property of Linux kernel lists in the forward direction.

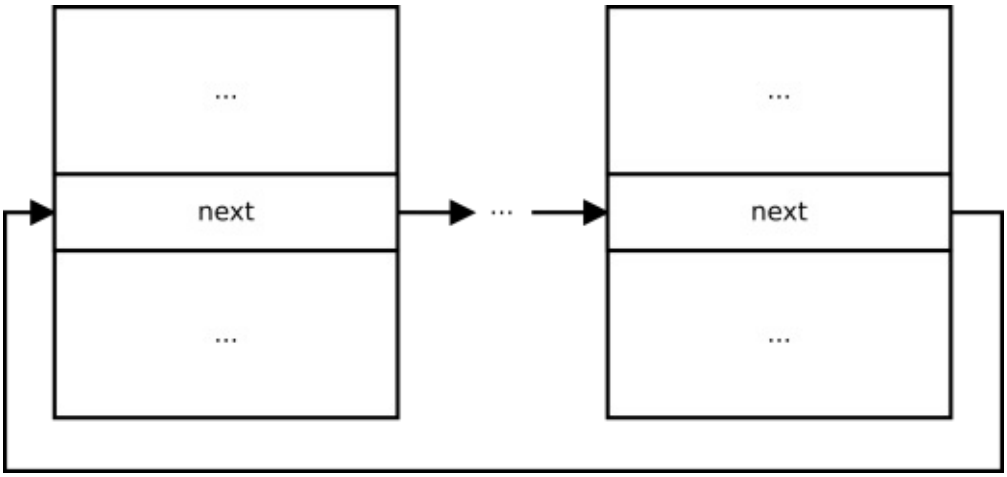

Figure 4.12: Cyclic successor reference in lists within the Linux kernel.

`task_struct` forms a cyclic doubly linked list as described above via its `tasks` field. First task in the list is named "swapper/0" according to its `comm` field, followed by "init" as second task [161].

4.3 Introduction to Rootkits

Rootkits are an advanced form of malicious software. Once they are installed with elevated system privileges, their goal is to hide their presence from legitimate users and provide long-lasting privileged access to the attacker [4].

The term "rootkit" is a combination of the terms "root" and "kit". "root" traditionally is the name of the administrative account on Unix-like OSs, "kit" refers to the actual implementation of the software components [4].

Use-cases of rootkits span a wide range. Law enforcement, industrial espionage, political espionage and cybercrime are some example applications for rootkits [32].

4.3.1 Historical Overview

Rootkits per definition try to hide themselves. Therefore it is challenging to comprehensively describe the historic evolution of rootkits. Below follow significant publicly known events in rootkit history.

First generation of rootkits masqueraded as Unix system programs [126]. Davis and Dake are attributed the first ever rootkit in 1990 for the Sun OS [37].

Hoglund presented the first rootkit for the Windows NT platform called "NTRootkit" in 1999. His patch to remove all security restrictions from objects within the Windows NT domain had a size of only four bytes [76].

Sony BMG included a rootkit in its copy protection software in 2005. Although the rootkit's task was to enforce copy protection and it had no malicious purpose, its functionality was soon exploited by malware developers [69].

Stuxnet is a computer worm targeting Supervisory Control and Data Acquisition (SCADA) systems that contains a rootkit component [102].

4.3.2 Types of Rootkits

Following sections classify rootkits according to their execution environment [4].

In general, the higher the privilege level the rootkit is running on, the harder it is to detect. This fact caused a "layer-below" race between the rootkit authors and the architects trying to defend their systems [142].

User-mode Rootkits

User-mode rootkits run on the same privilege level as applications. Due to the low privilege level, applications are the only available target for rootkits in this environment [4].

Kernel-mode Rootkits

Kernel-mode rootkits run on the same privilege level as the OS [4].

User-mode / Kernel-mode Hybrid Rootkits

The previous sections introduced user-mode and kernel-mode rootkits. Hybrid rootkits combine the properties of both [4].

Bootkits

Bootkits are a special type of rootkit, that interfere with the boot process of a system. Protection mechanisms such as the Windows driver signing policy, kernel patch protection and regular antivirus software can be circumvented by using bootkits in the early stages of system initialization [4, 71].

Hypervisor Rootkits

Hypervisor rootkits move the OS into a VM to avoid detection. Misusing the features of the hypervisor, a rootkit can intercept hardware calls made by the original OS [4].

Firmware / Hardware Rootkits

Firmware rootkits run directly in the firmware of the hardware. Replacement of the hardware is often the only option to fully remove a firmware rootkit [4].

4.4 Architecture of State-of-the-Art Rootkits

Section 4.3 introduced the concept of rootkits. To evaluate the Arm TrustZone as an environment for rootkits, it is necessary to analyze the architecture of existing state-of-the-art rootkits.

4.4.1 Infection Techniques

Usually rootkits support larger malware concepts like trojans, viruses or worms. By piggybacking on trusted software, they find their way into computer systems [4]. Droppers or downloaders are used to install and execute the malware. Anti-debugging and anti-emulation checks are usually executed before the payload deployed [103].

4.4.2 Concealment Techniques

It is a major goal of rootkits to stay concealed from legitimate users and detection mechanisms. Below follows an explanation of common concealment techniques used by rootkits [126].

Malicious System Files on Disk

Disk persistence allows a rootkit to survive reboots of the victim machine. Early rootkits masqueraded as system applications such as `ls` or `top` to stay hidden. However, as explained in Section 4.5.1, this approach is easy to detect. Thus, it is not a popular procedure among modern rootkits although it is still in use by some of them [126]. Newer techniques relying on disk persistence tend to avoid direct usage of the file system. FragFS was presented as a way to hide information in the file system meta information space [32]. Alternatively, regions of the physical disk that are not part of the file system such as inter-partition gaps can be used to hide a rootkit [71].

Hooking and In-Memory Redirection of Code Execution

Another important concept applied by rootkits is the modification of process memory in order to redirect execution to malicious code. A common term for this practice is "hooking". References to legitimate functions are hijacked by the rootkit to manipulate results returned to the caller and evade its detection. For example, a function that lists the content of a directory could be modified to not return files belonging to the rootkit. Hooking can be achieved in user-mode, kernel-mode, or a hybrid of both. Various techniques can be used to redirect the execution, for example via modification of the Interrupt Descriptor Table (IDT) or inline function patching. Because hooking is also used by legitimate applications, user-mode hooking is easy to detect but difficult to classify as malicious. Kernel-mode hooking is harder to detect and implement. Hybrid hooking is more complicated to detect and implement than kernel-mode hooking [126, 147].

Direct Kernel Object Manipulation

Direct Kernel Object Manipulation (DKOM) attacks the integrity of the kernel runtime state. Kernel structures are manipulated directly in memory to hide the presence of the rootkit. A typical example is the removal of a specific entry in the process list to make it invisible to user space tools. This technique relies on the fact that accounting utilities such as `ps` use a different list than the scheduler which keeps the processes running [21]. While hooking primarily targets static components such as the IDT, DKOM aims to subvert the integrity of the system by targeting dynamic kernel data structures responsible for bookkeeping operations [126, 137].

Code Mutation

In order to evade direct analysis, rootkits encrypt parts of their code. The encryption algorithm is modified with each generation to avoid emulation and runtime analysis [126]. Techniques listed above aim at hiding the presence of a rootkit. Code mutation protects the malicious software from being identified as such and complicates further inspection.

4.5 Rootkit Detection Mechanisms

Next, various rootkit detection mechanisms are presented.

4.5.1 File Comparison

Manipulation of system files by rootkits was described in Section 4.4.2. They can be detected by comparing the manipulated binary file to a clean copy. For this approach to work, clean copies of relevant system binaries need to be kept available and up to date [93].

4.5.2 Signature Analysis

Signature-based rootkit detection attempts to recognize code fragments known to be part of malicious software [126].

4.5.3 Behavioral / Heuristic Analysis

Additionally to the recognition of code fragments explained in the previous paragraph, there are other heuristics to detect rootkits. For example, specific repetitive system call sequences indicate the presence of a rootkit [126]. Static and dynamic analysis techniques can be combined for improved behavioral analysis [75].

Rootkits are often part of larger malware systems. By hiding network operations on the victim host, an attacker can load additional software components or communicate with other hosts without the user noticing. However, it is not possible to hide the actual network traffic once it leaves the network interface. IDSs can intercept the traffic and conduct an analysis [126].

4.5.4 Detecting Hooks

Section 4.4.2 explained the concept of hooking. Several techniques can be used to detect hooks installed by malicious software of which a few shall be mentioned. Memory scanning techniques periodically verify the values of common hooking targets. Hooking functions utilized by attackers for hooking can help to prevent attacks instead of only detecting them. Comparing in-memory code sections with the binary file on disk can reveal abnormal modifications [126].

4.5.5 Cross-View Detection

OSs provide in many cases multiple ways to accomplish the same task. Traversing of the file system is for example possible via the file API and direct queries to the disk controller. Having these options, two views on the same piece of information are available. Maintaining integrity of a complete system across all possible views is not considered feasible for a rootkit due to the complexity of modern OSs. Furthermore, rootkits themselves rely on working functionality of their victim systems that therefore can not be manipulated. Cross-view detection uses these weaknesses for the detection of rootkits. If there are different outcomes between views, a tampered view can be concluded and the presence of a rootkit is likely [126].

4.5.6 Invariant Specification

Aspects of the kernel that should not change with an uninfected OS can be periodically monitored. Violation of an invariant is evidence for the presence of a rootkit [126].

4.5.7 Hardware Solutions

A well-designed rootkit is difficult to detect by protection mechanisms running on the same host. Peripheral Component Interconnect (PCI) devices with Direct Memory Access (DMA) can be used to get an untampered view on the memory of a machine [126].

4.5.8 Virtualization Techniques

Similar to the concept of hardware-based DMA mentioned in Section 4.5.7, a hypervisor is able to inspect the memory of a VM. Concealment techniques modifying the guest OS can be discovered by software running outside the VM [126].

4.6 The Machine Emulator QEMU

QEMU [24] is an open-source machine emulator that has been extensively used for security-related research prototypes [22, 23, 106, 139, 143, 146, 148, 164]. One major feature of QEMU is that the CPU architecture of the host system running QEMU and the CPU architecture of the target system emulated by QEMU may differ. For example, a system based on the x86 architecture is able to run software compiled for the Arm architecture. This translation is performed dynamically at runtime [24].

Additionally to the dynamic translation of CPU instructions, QEMU emulates external devices such as keyboards and network cards. A debugger enables the detailed inspection of the emulated system [24].

Winter et al. [159] extended QEMU to support the features of the Arm TrustZone. Communication with both worlds is enabled through virtual serial ports. As OP-TEE is compatible with this implementation [121], QEMU is a flexible and convenient initial target for this work.

CHAPTER 5

Design and Implementation of a Secure World Rootkit

In the scope of this thesis, a proof-of-concept rootkit utilizing the Arm TrustZone was developed. Its purpose is to demonstrate the inherent practical capabilities gained by running in the secure world. Details about the technical design and implementation of the rootkit are provided in the remaining chapter.

5.1 Runtime Environment

Section 4.1.5 introduced the general concept of the Arm TrustZone world separation. This architecture has direct consequences on the setup of the runtime environment which is described in this section.

The basis of the runtime environment used in this work is provided by the OP-TEE project [112]. Previous research successfully built upon OP-TEE before [36, 59, 101]. Although OP-TEE focuses on development of an open-source secure world OS, it comes with a complete configuration that covers the normal world as well. Mechanisms for updating, building and running software for both worlds are included.

5.1.1 Normal World

Drivers for the communication with the Arm TrustZone are a requirement for the normal world OS. Linaro maintains a fork of the official Linux repository which is used in the OP-TEE default configuration [95].

User space of the normal world is kept minimalistic with only a few general purpose applications and the client applications that are part of OP-TEE. By default, the normal world configuration shipped with OP-TEE sets up two users intended for interactive sessions:

- A privileged user named “root”.
- A non-privileged user named “test”.

5.1.2 Secure World

OP-TEE is chosen as OS for the secure world. A major challenge is the deployment of the secure world code. Regular consumer devices apply authentication checks on the secure world images on startup. Only files cryptographically signed by the respective vendor can legitimately be loaded. Developers and researchers not affiliated with a vendor are left with the following possibilities to deploy custom code [124]:

- Emulate a device.
- Use a development board.
- Find a way to bypass the security restrictions.

Emulation via QEMU [24] was chosen for this experimental implementation. Advantages over the other possibilities are that emulation is easily accessible, trivial to set up, free of costs and enables convenient debugging features. OP-TEE comes with support for running in QEMU for the Armv8 [7] architecture, which eases the setup of a working environment significantly. While OP-TEE officially supports a selection of physical devices [118], deployment of the rootkit to these is out of scope for this work.

5.1.3 Communicating with both Worlds

Upon start of the OP-TEE setup, QEMU launches two additional terminal windows. Figure 5.1 shows the terminal windows when running the OP-TEE environment in QEMU. Left window belongs to the QEMU monitor, followed by the secure world and the normal world on the right.

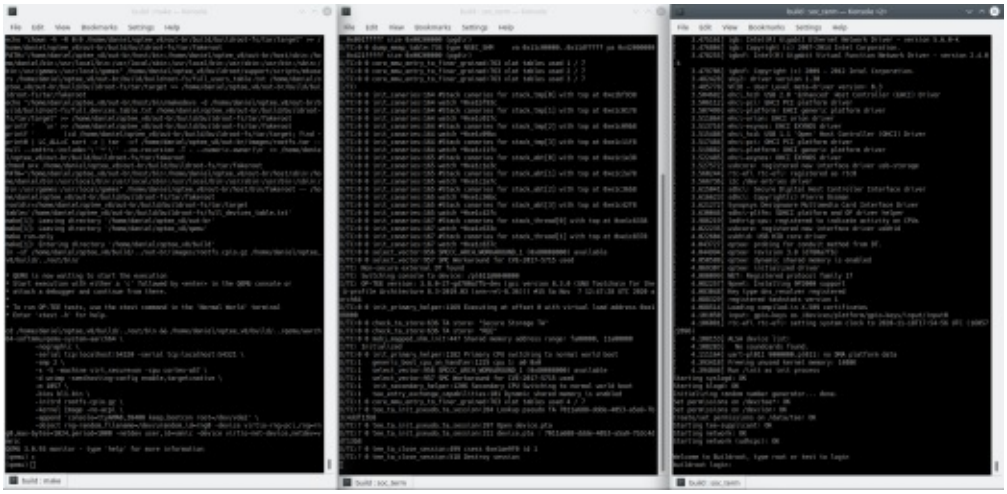

Figure 5.1: QEMU monitor, secure world and normal world terminals (from left to right).

Task of the initially started QEMU monitor terminal is to manage the emulator. A convenient feature of the monitor commonly used during development of the rootkit is the inspection of arbitrary memory locations. Next is a read-only connection to the log of the secure world. Finally, the last terminal is interactive and attached to the normal world OS. This terminal acts as interface for the user to log in and run applications.

5.2 Compilation Setup

The rootkit is compiled as a so-called "pseudo TA". As such, the rootkit conforms to the API structure of a TA but is technically part of the secure world OS itself. Major advantage of this approach is that the rootkit is able to make use of the available APIs [152].

On the one hand there is the external API, which is exposed to the normal world. Regular normal world applications use this API to call secure world TAs via the secure monitor. OP-TEE follows a standard by GlobalPlatform on how the API is structured [66].

Besides the external API, there is an OS-internal API. Functions to accomplish common tasks such as mapping normal world pages or logging debug messages are provided.

C is used as programming language for the complete implementation. Source files which are part of the rootkit are integrated into the build setup provided by the OP-TEE project.

5.3 Rootkit Architecture

Considering the properties of the Arm TrustZone described in Section 4.1.5, its permissions can be used for the development of a rootkit. Specifically, the TrustZone's ability to

access the physical memory is investigated in this work.

In the scope of this research scenario, the exact structure of the normal world kernel is assumed to be unknown. No access to the normal world kernel source code or the compiled binary on disk is possible. Locations of symbols such as functions and data structures are not available to the rootkit. Field offsets within kernel structures might vary between kernel versions due to added or removed fields. Partially the order of fields within data structures is randomized as a security measure during the compilation process.

OP-TEE uses a memory region shared between the normal world and the secure world for data transfer [44]. The secure world module of the rootkit abuses the low-level implementation of this feature to achieve full access to the physical memory. Normal world memory pages can be mapped to the shared memory region by knowing their physical addresses. Once mapped, the memory pages can be accessed via secure world virtual addresses in the same way as regular secure world memory. Memory manipulations provide the respective rootkit functionality. Due to limits enforced by OP-TEE, unused shared memory is freed again by the rootkit as soon as it is not needed anymore. Convenient functions to map normal world pages and free them again are part of the internal OP-TEE API accessible to the rootkit pseudo TA.

Being located at EL1 of the secure world, this module is hidden from conventional normal world rootkit detection mechanisms. Invariants are employed by this module to gather information and reconstruct parts of the internal state of the kernel. A similar approach was used successfully by Xiao et al. [161] for the HyperLink tool on the x86 architecture.

An unprivileged normal world client application communicates with the secure world module using the standardized API. Without elevated privileges, the normal world application does not have access to kernel-internal information which could be passed to the secure world. Nevertheless, system calls can be utilized by the normal world client to trigger actions within the kernel. Subsequent changes of the kernel state in memory can then be observed and interpreted by the secure world module.

Summarizing, the implemented rootkit consists of the following two parts:

- An unprivileged normal world application (normal world EL0).
- An secure world OP-TEE pseudo TA (secure world EL1).

Figure 5.2 visualizes compromised components in their respective world and ELs.

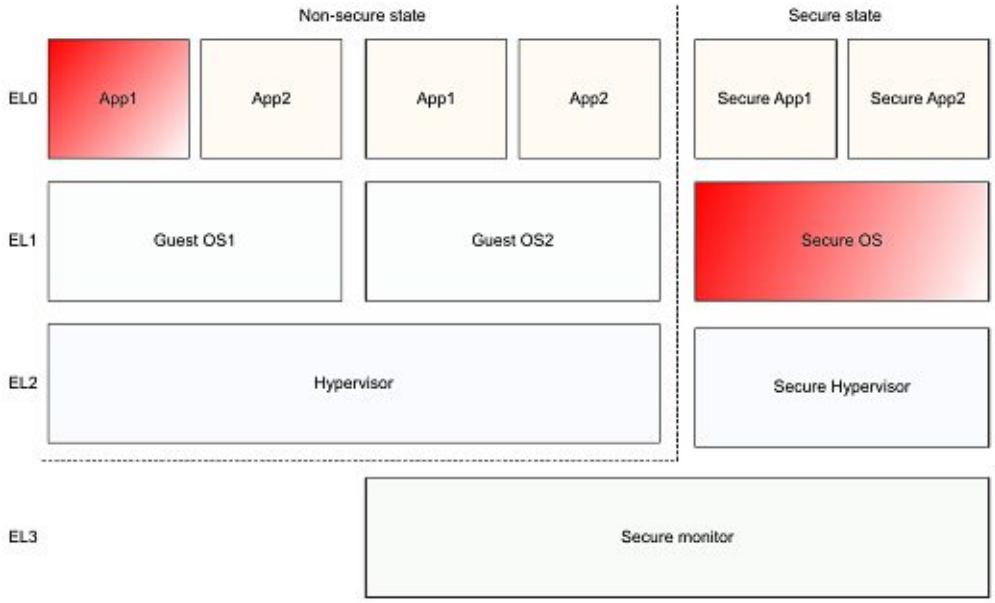

Figure 5.2: Components compromised by the rootkit are highlighted [7].

During development, a custom driver in the Linux kernel (normal world EL1) was used to manually verify values extracted by the rootkit. It is not part of the actual rootkit and therefore neglected in this description.

5.4 Implemented Rootkit Functionality

An API-oriented architecture relying on the design presented in Section 5.3 is used for the implementation. Three malware features are fully functional, but the modular design allows trivial extension of the rootkit.

For simplicity, the rootkit does not keep an internal state across calls. Each invocation of a rootkit function takes care of its prerequisites by itself.

Following sections describe the implemented functionalities in detail.

5.4.1 Memory Carving

The first implemented rootkit functionality is data extraction. Normal world memory is carved for data structures containing static byte sequences. Based on a given leading byte sequence header and trailing byte sequence footer, memory regions spanning across both sequences are identified. Listing 5.1 explains the abstract structure of a memory region to identify.

```
[Header]
[Content]
[Footer]
```

Listing 5.1: Structure of a memory region to identify.

Corresponding byte sequences are passed by the normal world client to the secure world component. Thus, the secure world implementation is generic and can be applied to arbitrary data formats having static headers and footers.

Specifically, carving for private keys of the Rivest–Shamir–Adleman (RSA) [122] public-key cryptosystem conforming to the Privacy-Enhanced Mail (PEM) [97] format was implemented as a demonstration of the rootkit. This type of keys can be trivially identified by its characteristic header and footer. Listing 5.2 shows the general structure of such a key.

```
-----BEGIN RSA PRIVATE KEY-----
[Encoded private key]
-----END RSA PRIVATE KEY-----
```

Listing 5.2: Structure of an RSA key conforming to the PEM format [97].

To detect the memory regions structured as in Listing 5.2, the normal world client passes a header parameter as shown in Listing 5.3 to the secure world component.

```
-----BEGIN RSA PRIVATE KEY-----
```

Listing 5.3: Header parameter passed by the normal world client to detect RSA private keys.

Listing 5.4 shows the respective footer parameter.

```
-----END RSA PRIVATE KEY-----
```

Listing 5.4: Footer parameter passed by the normal world client to detect RSA private keys.

Major difficulty for this feature is to narrow down the relevant memory space to inspect. Azab et al. [19] presented a way to trap translation table updates by instrumenting the normal world kernel source code. However, Section 5.3 restricted the adversary model to only consider access to the runtime memory. Instead, the following approach is implemented. Delimitation of the relevant memory regions requires knowledge about the location of memory pages and memory blocks as explained in Section 4.1.4. Physical addresses of these memory units can be calculated by a recursive scheme that starts at the initial page table. Given the content of the kernel image, the location of the initial page table can be deduced.

Details about the secure world side of the implementation are provided next. Figure 5.3 visualizes the complete procedure.

Finding the Kernel Image

First, the kernel image needs to be found. A bruteforce search is the most primitive way of finding specific memory regions. Every reasonable location in the theoretical address space is checked whether it is mapped by the normal world kernel and its content matches a specified sequence of bytes.

In the default configuration of OP-TEE version 3.11.0, the normal world kernel image is loaded at a randomized address via KASLR [17, 53]. Section 4.2.1 described the UEFI header as distinctive start of the kernel image. Iterating the complete theoretical address space when using 48 address bits is a significant computational effort, especially on low-powered mobile devices. Given current devices, only a fraction of the theoretically addressable memory is available. Alignment restrictions can be used to decrease the amount of addresses to check and speed up the bruteforce search.

Starting from the normal world memory base at physical address `0x40000000` [13], memory pages are mapped to the secure world one by one. The kernel image and its leading UEFI header is aligned to a 64KB (2^{16}) boundary. Considering this limitation, the theoretical number of addresses to check can be reduced from 2^{48} to 2^{32}. At the time of writing, the UEFI header of the ARM64 Linux image is generated via the opcodes of a specific assembly instruction right at the beginning of the image. Leading bytes of each page are checked for the static value `0x91005a4d` which corresponds to the UEFI header assembly instruction in Listing 5.5. Checking only for the existence of the "MZ" bytes instead of the complete instruction opcode would be a more general but also less reliable approach.

```
add x13, x18, #0x16
```

Listing 5.5: Assembly instruction to generate the "MZ" UEFI header.

If the value is found, the start of the kernel image was identified with high probability.

Finding the Initial Page Table

A page table walk can be used as optimization of a bruteforce search over the full theoretical memory address space. Iterating over the page tables requires knowledge about the location of the initial page table (called `swapper_pg_dir` on Linux). Given knowledge of the address of the UEFI header, the steps to identify the initial page table are as follows.

While parts of the kernel are at runtime located at randomized or compilation-dependent locations, there exist important cornerstones mapped at absolute addresses or relative offsets that can be considered stable across different versions of the kernel. Instructions of the kernel are resident in memory after the system booted up. Parsing those instructions can reveal additional information about the compilation-dependent properties of the kernel. Directly after the `add` instruction forming the UEFI header, execution jumps to the primary entrypoint represented by the symbol `primary_entry` via the unconditional

branch instruction `b`. Listing 5.6 shows the relevant instructions in the ARM64-specific Linux source code [14].

```
add x13, x18, #0x16
b primary_entry
```

Listing 5.6: First instructions of the ARM64 Linux image.

Compilation adds the relative address of the `primary_entry` symbol to the `b` instruction. Opcodes of the jump instruction are parsed to calculate the target address [7]. According to the ARM64 Linux linker script, the initial page table is located directly before the `primary_entry` symbol [18]. Memory space is traversed backwards until a non-zero page is encountered. The first non-zero page before `primary_entry` is the `swapper_pg_dir` symbol.

Page Table Walk

Section 4.1.4 explained the fundamentals of the Armv8 virtual memory system. Once the initial page table is identified, the rootkit maps it into the secure world memory. Each 64-bit value of the page table is inspected separately and interpreted according to the architecture reference manual as listed below [7].

1. If the Least Significant Bit (LSB) (bit 0) of the page table entry is not 1, it is an invalid entry that is skipped.
2. If bit 1 is set, the entry refers to either of the following:
   a) On translation level 0, 1 and 2, the entry refers to a translation table on the next translation level.
   b) On translation level 3, the entry refers to a memory page.
3. If bit 1 is not set on translation level 1 or 2, it refers to a memory block.

This scheme is applied recursively to cover all translation levels. Memory blocks and pages contain the actual data and compose the memory regions to be searched. Addresses used by this scheme to refer to pages, tables and blocks are physical addresses, i.e., no further processing is required to map them into the secure world address space [7].

Memory blocks have a fixed size that depends on the translation granule size. While the proof-of-concept rootkit was extensively tested with 4KB pages, the implementation can be adjusted trivially to work with other granule configurations.

Pattern-based Matching

Previous sections explained the identification of memory regions to be considered. Finally, these memory pages and blocks are searched for the passed patterns. Although more efficient algorithms exist, a naive byte-wise comparison with algorithmic complexity `O(mn)` is used for simplicity.

First, the passed data header value is searched for. If the header is found, this memory location is saved as the beginning of the memory region to be detected. Starting from the location of the header, now the data footer value is searched for. In case also this search is successful, the location of the footer marks the end of a valid match.

Found matches can then be further processed in the secure world (e.g., transmitted over the network via the GlobalPlatform sockets API [67]) or returned to the normal world client.

Summary

Summarizing the feature implementation to carve memory described in this section, the steps are as follows:

- The normal world client calls the secure world with the patterns as parameters.
- Next, the secure world searches for the UEFI header.
- Once the UEFI header is found, the address of the initial page table can be calculated.
- Memory pages and blocks are searched recursively for the patterns passed by the normal world.
- Matches can then be processed arbitrarily.

Figure 5.3 visualizes the overall process.

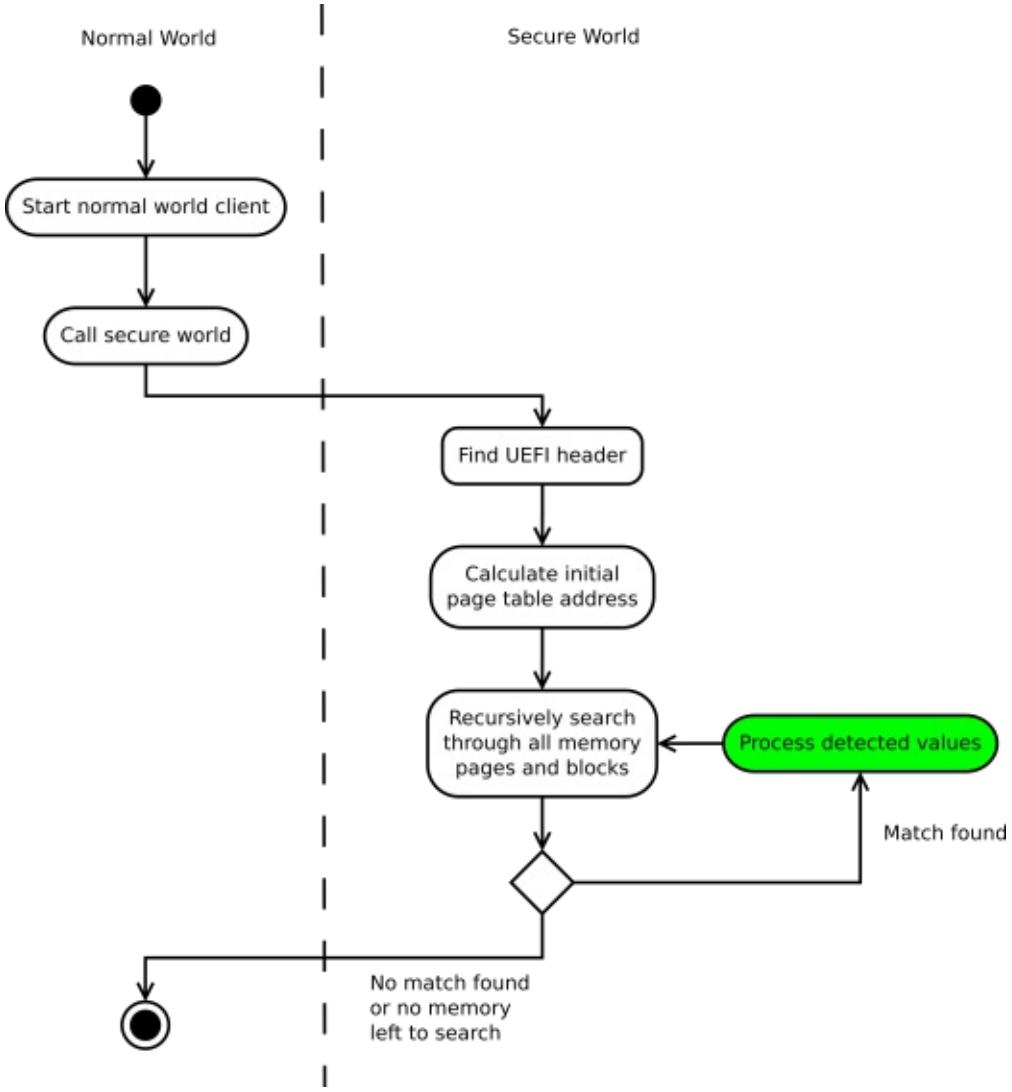

Figure 5.3: Steps to find static patterns in the allocated memory.

5.4.2 Privilege Escalation

Another major functionality implemented in the scope of this thesis is the elevation of privileges. An unprivileged (non-root) process is modified to be capable of executing actions with elevated (root) permissions. Figure 5.4 shows the concept of a user “test” attacking a more privileged user “root”.

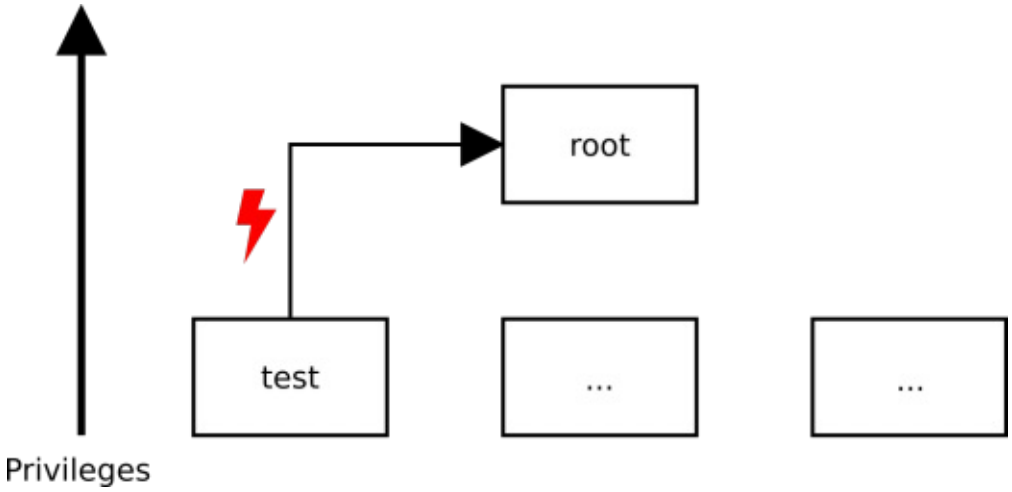

Figure 5.4: Schematic attack of the user “test” on the user “root” to gain privileges.

From an high-level API point of view the privilege escalation works as follows. The normal world client passes the PID of an arbitrary, but in general unprivileged, target process to the secure world. Next, the rootkit uses memory operations to identify and manipulate the kernel process structure to elevate the privileges of the selected target process (DKOM). Execution then continues in the normal world and the target process is able to launch actions with elevated privileges.

Remaining section lists the single steps of the privilege escalation in detail. A visualization of the steps is provided in Figure 5.6.

Finding the Initial Task Structure

Section 4.2.2 explained how processes are managed by the Linux kernel in a doubly linked list of instances of the `task_struct` type. Knowledge about the location of the list in memory is crucial for the goal of elevating privileges of a process. Additionally, randomization of field order in the structure can potentially be applied at compile time and needs to be considered for a stable implementation [78].

Each task has a name stored in the `comm` structure field. The first process to be started by the kernel (i.e., the first element of the task list) is called `init_task` with the process name “swapper” [161]. On Symmetrical Multiprocessing (SMP) systems there is an additional “/0” suffix for the initial task on the first CPU. “swapper/0” as a string is relatively easy to identify within arbitrary data, thus a bruteforce search starting from the UEFI header is applied.

Even with the location of the process name of `init_task` available, due to structure-internal randomization the beginning of the `task_struct` instance can still not be trivially concluded [78].

Randomization only concerns parts of the `task_struct`. Thread information fields at the beginning of the structure are explicitly excluded from the compile-time randomization [78, 150]. Typical bit patterns of these fields help to identify the beginning of the `task_struct` [130]. Implicitly, the identification of the structure beginning provides the offset of the `comm` field.

Calculating the Kernel Image Virtual Address Offset

Subsection "Finding the Kernel Image" of Section 5.4.1 and subsection "Finding the Initial Task Structure" of this section explained how to circumvent randomization of physical kernel image addresses by using bruteforce searches and considering data constraints. At runtime, data structure instances such as `init_task` exclusively work with virtual addresses. However, for OP-TEE only physical addresses are accessible. Further interpretation of memory references within the kernel image requires identification of the exact translation process and and the constant offset between physical and virtual kernel address mappings. Several properties of virtual addresses are checked for all candidates within the assumed range of the `init_task` instance.

First, a coarse filter verifying the expected format of the virtual address is applied. Kernel virtual addresses have all bits not used for the actual addressing set to 1 [15]. A formal explanation is given in Equation 5.1, which makes use of "`&`", "`~`" and "`<<`" as binary operators like in the C programming language[1]. `VA_BITS` is a constant that refers the number of bits used for the virtual addressing, e.g., 48 for the 4KB translation granule size.

```
candidate & (~0 << VA_BITS) == (~0 << VA_BITS)        (5.1)
```

Virtual addresses share the page offset bits with their corresponding physical address. Depending on the size of the pages, the offset consists of a different number of bits. 4KB pages which are used in the scope of this work use 12 offset bits [7].

As a final check, due to the semantics of its fields `init_task` must contain multiple references to itself [80]. Occurrences of the candidate are counted. If a minimum threshold for the number of occurrences is reached, the check is successful.

Given all the constraints listed above, the virtual address of `init_task` can be found reliably within the limited memory region.

Iterating Tasks

Finding and analyzing tasks requires a stable mechanism to iterate the task list. Starting from the initial task, the pointer to the next instance needs to be identified. The field which manages the doubly linked list is called `tasks`. First entry of the `tasks` field is

[1] For simplicity, all literals in this equation are assumed to be 64-bit in size to match the candidate address. Actual C code requires explicit typing by adding literal suffixes (e.g., `0ul`) to achieve this.

the pointer to the successor, therefore the offset of the `tasks` field is identical to the offset of the successor field it contains.

All virtual addresses in the assumed range of the `task_struct` instance are inspected and their translation is simulated. If the `comm` field of the second task has a value of "init", the successor field of the `tasks` attribute was successfully identified [161]. This invariant is used to verify a successful identification of the `tasks` field.

Section 4.2.2 explained how lists within the Linux kernel differ from commonly used user space implementations. Xiao et al. [161] presented that offsets within the structure are constant among all instances of the structure. Based on this statement, general formulae for arbitrary structure fields in the list elements can be provided.

Equation 5.2 shows the calculation of the beginning of the second task in the task list. The asterisk ("$*$") symbol is used to mark the access to the value at the given address (like in the C programming language).

$$*(\texttt{init\_task\_start} + \texttt{tasks\_field\_offset}) - \texttt{tasks\_field\_offset} \tag{5.2}$$

Equation 5.2 can be extended for arbitrary fields in the list. Calculation of the address of the `comm` field of the second entry in the task list is shown in Equation 5.3.

$$*(\texttt{init\_task\_start} + \texttt{tasks\_field\_offset}) - \texttt{tasks\_field\_offset} + \texttt{comm\_field\_offset} \tag{5.3}$$

Once the invariant concerning the name of the second task is fulfilled, the offset of the `tasks` field was found. Iterating over all tasks in the cyclic list requires to follow the value of the `tasks` field until it is equal to `init_task`.

During this stage, it is the first time virtual addresses need to be resolved to physical addresses. Different types of memory layouts need to be considered to improve compatibility across different kernel versions [16]. Relevant address translation implementations were taken directly from the Linux kernel source code. To discover the correct implementation for the currently running system, a bruteforce scheme of translation simulations is applied. Once a translation scheme fulfills the above invariant, it is chosen for every future address translations.

Identifying Processes

Further analysis and manipulation of tasks requires them to be identifiable. A data invariant is used to find PIDs. Equation 5.4 formalizes the invariant between two consecutive tasks.

```
*(task_start + pid_field_offset)
                                ==
            *(*(task_start + tasks_field_offset)
             - tasks_field_offset + pid_field_offset) - 1   (5.4)
```

Below follows a description of the implementation.

It is assumed that tasks started early by the kernel are running until the system is shut down. Therefore, the processes at the start of the task list are assumed to have PIDs starting at 0 and being strictly incremented by 1 without any interruption. Each process has an expected PID at an unknown offset. Offsets are tried starting from 0 and incremented by the size of a PID after each iteration. If the value at the current offset matches the expected PID, the next process is checked for its respective expected PID. The PID offset is found if an empirically determined number of `task_struct` instances at the start of the list have incrementing PIDs beginning with 0.

Identifying and Overwriting Credential Pointers

Permissions of a process are defined by the credential structures referenced by its corresponding `task_struct` instance. At the time of writing, there are two pointers to credential structures inside `task_struct`:

- `cred`
- `real_cred`

Both fields store the address of an instance of a structure type `cred`, which is assumed to be randomized internally at compile time. While the internals of the structure as well as the exact purpose of splitting the permissions into two fields is out of scope of this work, the state of the pointers in memory is examined in detail. Initially, both fields contain the same value, i.e., they refer to the same structure in memory. Changes to the credentials of a `task_struct` are applied by the kernel through the `override_creds` function.

One notable example where `override_creds` is used is within the `access` system call [3]. During a fraction of the execution of the `access` system call, the values of the `cred` and `real_cred` fields differ. Afterwards, the initial value is restored and the two fields contain identical values again. Figure 5.5 sketches this behavior. Credential values of the initial task “swapper/0” represent elevated privileges and are never modified.

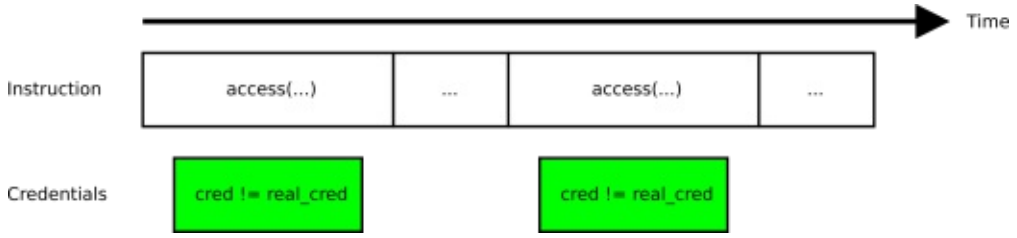

Figure 5.5: Temporary difference between `cred` and `real_cred` during execution of the `access` system call.

These structural properties of the actively developed Linux kernel can potentially be changed in future versions. They were first published as part of a stable Linux kernel release in version 2.6.29 in 2009 [46, 47]. Due to the age of this implementation, this behavior considered well-established and assumed a precondition for this rootkit implementation.

The implemented approach for a privilege escalation works as follows.

First, the normal world client process is forked. Sole purpose of the newly created child process is to repeatedly call the `access` system call and thereby cause the kernel to modify the credential pointers of the respective `task_struct` in memory. As soon as the parent process is done with its procedure, the child process serves no further purpose and is killed.

Meanwhile, the parent process calls the secure world rootkit component via the regular API and passes the PID of the child process along as parameter. After identifying the `task_struct` instance of the initial task in memory as described in Section 5.4.2, the memory range of the structure is searched for two identical 64-bit numbers which match the format of virtual addresses. Once two candidate offsets are identified, the child process of the client is observed. A heuristic is used to validate the candidates. In case the values at the offsets in the `task_struct` of the child process differ in some cases but are identical in others, the offsets of the `cred` and `real_cred` fields have been successfully calculated. Because the system call only modifies the pointers for a fraction of its execution, the check for differing values is repeated several times to get more reliable results.

Although mapping and modifying the credential structure instances would be possible at this stage, compile-time randomization within the structure renders this scheme highly complex. A trivial approach is to overwrite the credential addresses of the target process with those of the credentials of the initial task [57]. Through this action, all future child processes of the target process inherit the elevated privileges. Effective permissions of the target process itself are not modified. For this reason a shell which can then launch arbitrary processes with elevated privileges is a suitable choice for a target process.

Section 4.2.2 mentioned that credentials within `task_struct` are subject to a reference counting mechanism. Copying credential addresses to foreign tasks as presented above

bypasses the reference counting mechanism. Terminating the target process and thereby destroying its credentials causes a kernel fault, because the credentials are still referenced in the initial task.

Summary

Summarizing the feature implementation to achieve privilege escalation described in this section, the steps are as follows:

- The normal world client is forked.
- `access` is repeatedly called by the child process.
- In the parent process the secure world is invoked.
- The secure world component executes a text-based search for the initial task name.
- Having the offset of the task name available, the start address of the `task_struct` instance is calculated.
- Offsets for the task list, PID and credential pointers are detected.
- Credentials pointers of the target task are overwritten with those of the initial task.
- All future child processes of the target task inherit the elevated privileges.

Figure 5.6 visualizes the steps listed above.

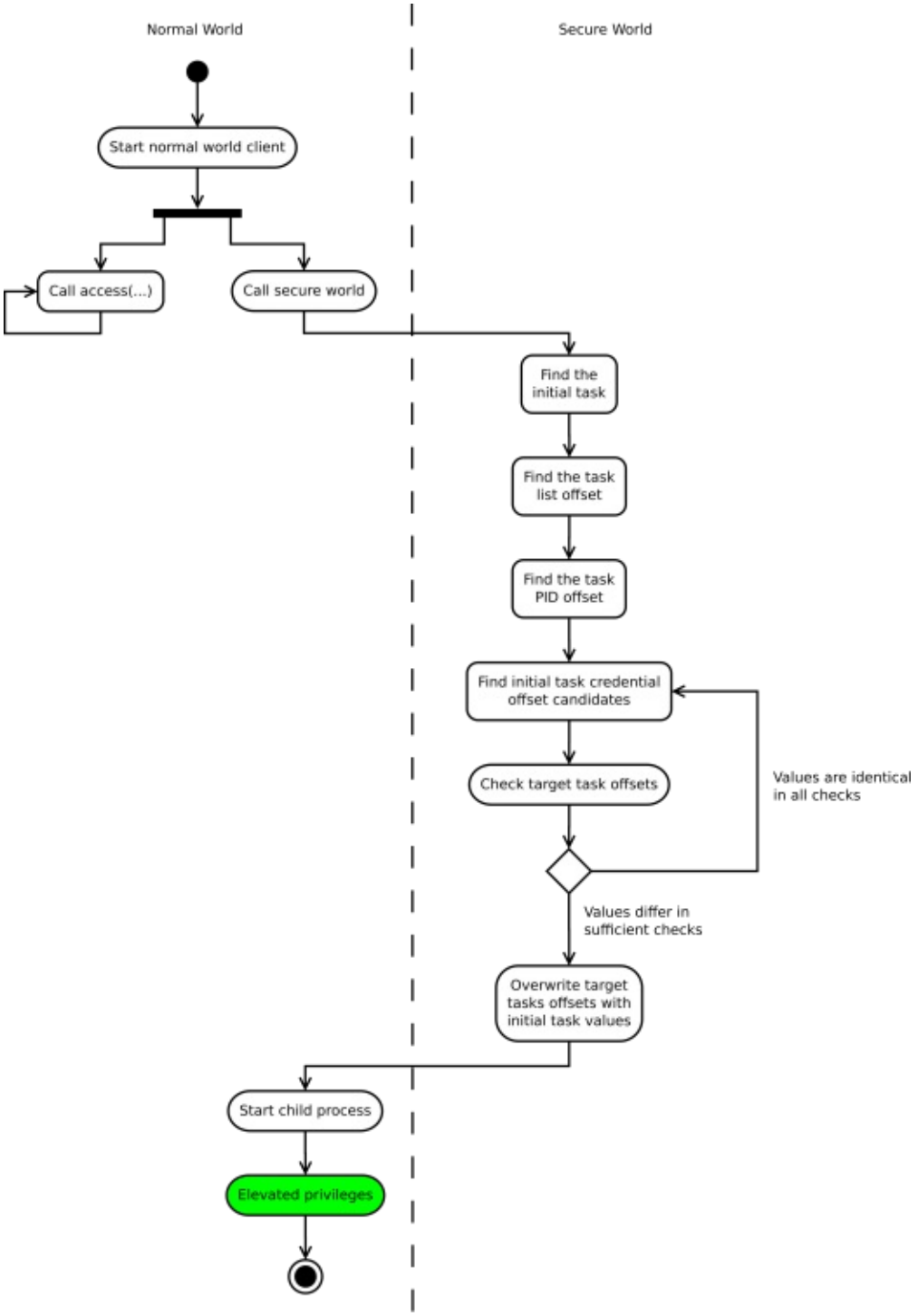

Figure 5.6: Steps to overwrite credentials of the target task and elevate privileges.

5.4.3 Process Starvation

The last rootkit feature developed as part of this work is the manipulation of process states. Modifying the state of a process changes the way the process is treated by the scheduler. Setting the respective state prevents the target process from being scheduled. Without being considered by the scheduler, the process execution is starved of CPU time and stalled. Graziano et al. [70] suggested antivirus systems or IDSs as target for process starvation.

Invocation of this feature starts with a call of the normal world client to the secure world. In addition to the PID of the target process to modify, the rootkit API expects the new state to be provided as second parameter. Memory operations form a DKOM to change the state of the selected target process to the passed parameter. Execution continues in the normal world and the target process is not scheduled anymore.

Initial steps of the exploitation are identical to Section 5.4.2. Instead of the final step of manipulating the `cred` and `real_cred` fields, the `state` field is used for this technique. Following lists the additional steps of the process starvation in detail. Figure 5.7 gives an overview of all necessary steps.

Identifying and Overwriting Process State Information

Current state of a process is represented via the `state` field of the respective `task_struct` field. Although the `state` field is not part of the randomized section of `task_struct`, the thread information stored at the beginning of the structure might change in size. To change the value of the `state` field, its offset within the structure needs to be recovered.

`task_struct` instances are searched for typical state values. All offsets within the `task_struct` instance are tested for the known values in Table 4.1. Because the `state` field is located before the randomized section, it is reasonable to start with low offsets.

Following state combinations are expected to be found in the task list [140].

- `TASK_RUNNING` (tasks ready to run)
- `TASK_INTERRUPTIBLE` (sleeping tasks)
- `TASK_UNINTERRUPTIBLE | TASK_NOLOAD` (idle kernel tasks)

If an offset is discovered that yields multiple processes in the states listed above, the `state` field was recovered successfully.

Knowing the offset of the field, it can be modified arbitrarily. Depending on the desired effect, multiple process state values come into consideration.

Assigning the process a state of `EXIT_ZOMBIE` prevents it from being scheduled in the future. However, this modification is visible to normal world EL0. Possibilities to view the change include the tools `ps` and `top` as well as the `/proc` file system.

A more stealthy alternative is the `TASK_DEAD` state. Aforementioned possibilities still show the process as running. Caveat of this approach is that the kernel panics when the target process is running while the process state is changed.

Summary

Summarizing the prerequisites explained in Section 5.4.2 and the feature implementation to achieve process starvation described in this section, the steps are as follows:

- The normal world client calls the secure world with the PID of the target process and its desired state as parameters.
- The secure world component executes a text-based search for the initial task name.
- Having the offset of the task name available, the start address of the `task_struct` instance is calculated.
- Offsets for the task list, PID and state are detected.
- State of the target task is overwritten with `EXIT_ZOMBIE` to represent a zombie process.
- As a result, execution of the target process is stalled.

Figure 5.7 visualizes the steps listed above.

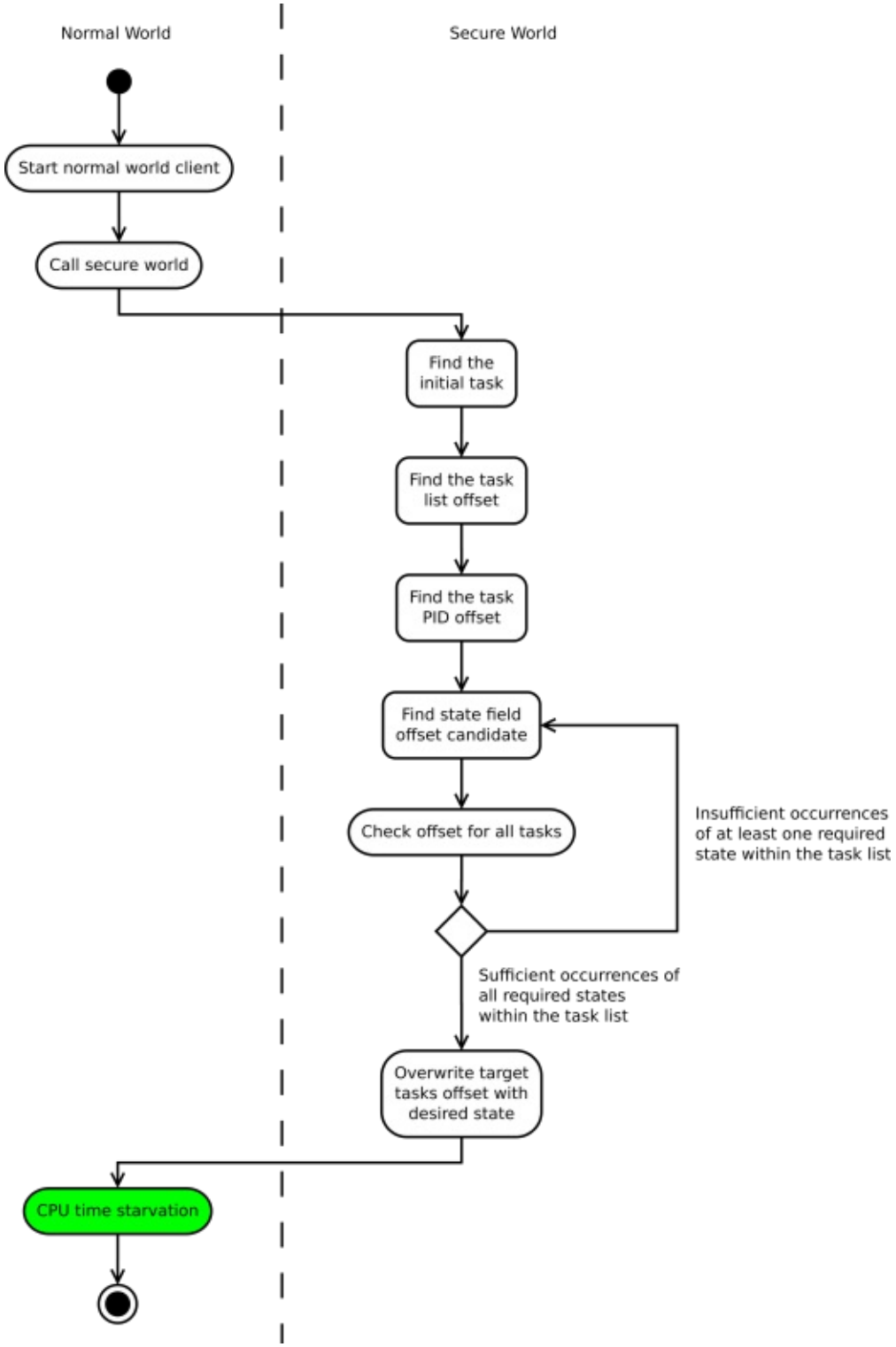

Figure 5.7: Steps to starve the CPU time of the target task.

CHAPTER 6

Evaluation and Impact Analysis

Throughout the scope of this thesis the Linux kernel runtime memory was considered an unknown structure in general. Invariants were used by the secure world rootkit to reconstruct internal information. Chapter 5 explained the implementation in detail. However, as the Linux kernel is an actively developed software project at the time of writing, it needs to be assumed that the implementation changes over time. A stable rootkit implementation should be able to cope with minor changes in the kernel while relying on established concepts and properties. This chapter benchmarks the rootkit implemented in the scope of this thesis against various versions of the Linux kernel.

6.1 Scope

According to the OP-TEE documentation, the required generic TEE framework is part of the official Linux kernel since version 4.12 [61]. Starting from release 4.12, all major versions that are supported by the Linaro fork of Linux [95] are evaluated. Table 6.1 lists the tested Linux kernel versions with their release date.

Linux Version	Release Date
4.12	2017-07-02
4.13	2017-09-03
4.14	2017-11-12
4.15	2018-01-28
4.16	2018-04-01
4.17	2018-06-03
4.18	2018-08-12
4.19	2018-10-22
4.20	2018-12-23
5.0	2019-03-03
5.1	2019-05-05
5.2	2019-07-07
5.3	2019-09-15
5.4	2019-11-24
5.5	2020-01-26
5.6	2020-03-29

Table 6.1: Tested Linux kernel versions and their respective release date [85].

By default, only a shallow clone of the Linaro Linux repository is created. Accessing individual releases in the form of Version Control System (VCS) tags requires a complete clone. Listing 6.1 shows the command to turn the shallow clone done by the OP-TEE system into a complete clone.

```
$ git fetch --all --unshallow
```

Listing 6.1: Command to make VCS tags available in a shallow repository.

To get an impression on the quantity of changes between two Linux kernel versions, simple statistics are generated. Listing A.1 shows the script used to generate these numbers from the Linaro Linux kernel repository fork [95] and the VCS tags shown in Listing A.2.

Table 6.2 lists each tested Linux release with the corresponding number of changed files, inserted lines and deleted lines since the previous release.

6.2 Procedure

Identical default configuration values are set by the OP-TEE build system for all tested releases. A memory translation granule size of 4KB is used for all tests. Kernel security features might be ignored by the OP-TEE build configuration and are enabled or disabled according to the default configuration of the Linux kernel itself.

All rootkit functionalities presented in Chapter 5 are evaluated. The evaluation procedure works as follows. For each of the Linux versions to test, the kernel repository is checked

Linux Version	Files Changed	Insertions	Deletions
4.12	N/A	N/A	N/A
4.13	10676	808313	212165
4.14	23143	610573	336296
4.15	13162	600291	276768
4.16	11931	491999	297996
4.17	14227	631104	810344
4.18	12928	508161	606853
4.19	11593	514241	206654
4.20	11238	615429	248361
5.0	11887	518937	271423
5.1	11801	512729	255989
5.2	30524	557864	465517
5.3	13189	918080	328913
5.4	12378	712146	315234
5.5	11556	548877	232674
5.6	11408	543044	228000

Table 6.2: Changes between Linux releases extracted from the output in Listing A.3.

out. Next, the complete TEE environment is built and started up. Once the system is ready, the unprivileged user “test” logs in at the normal world terminal and starts the normal world rootkit client.

First, the privilege escalation is tested. It is expected that after a successful execution the credentials of the target process changed from “test” to “root”. If the execution fails or the user does not match “root” for all future children of the target task after the execution finished, the functionality is considered to be broken.

Second feature to test is the starvation of a user space process. Another process that creates a file in an endless loop is launched. Before the invocation of the rootkit, the modification time of the file is expected to change continuously. Upon successful execution of the function, it is expected that the process is in the “zombie” state but the modification time of the file remains unchanged.

Independently of the result of the previous tests, the memory carving feature is tested. RSA private keys are placed within the normal world client (EL0) as well as a kernel module (EL1). While the total set of detected keys might vary between tests, a successful execution must include at least both keys intentionally put in place.

Building the system with different kernel versions requires significant compilation time. Test scripts were developed to completely avoid the need for manual interactions. Appendix B contains the code of the scripts and the changes to the OP-TEE build system necessary to invoke the scripts.

6.3 Results

Evaluation runs are classified according to the following categories.

- **Compatible (C):** A kernel version is compatible, if the rootkit is invoked successfully and the expected result is achieved. Processes were modified as expected and the positioned RSA keys were found.
- **Incompatible (I):** Incompatible versions are invoked successfully as well. However, the rootkit is not able to produce the expected result. Either the processes were not modified as expected or the RSA keys could not be found.
- **Failed (F):** Lastly, the compatibility test might fail. Failures are considered to be triggered externally, e.g., invocation of OP-TEE 3.11.0 is broken in the respective kernel version. This type of error does not depend on the rootkit implementation.

Final results of the evaluation are listed in Table 6.3.

Linux Version	Privilege Escalation	Process Starvation	Memory Carving
4.12	C	C	I
4.13	C	C	I
4.14	C	C	I
4.15	C	C	I
4.16	C	C	I
4.17	C	C	I
4.18	C	C	I
4.19	C	C	I
4.20	C	C	C
5.0	C	C	C
5.1	C	C	C
5.2	F	F	F
5.3	F	F	F
5.4	F	F	F
5.5	C	C	C
5.6	C	C	C

Table 6.3: Evaluation results setting rootkit functions into relation with Linux kernel version.
C = Compatible
I = Incompatible
F = Failed

6.4 Discussion

Discussions and explanations of the results presented in Section 6.3 are presented in this section.

Memory carving is incompatible with all versions prior to v4.20. Reason for this is that the property the heuristic uses for discovering the `swapper_pg_dir` symbol was introduced in that version [18]. Within the scope of this work, no alternative heuristic could be found. However, coming up with a working heuristic for the currently incompatible versions should not be considered impossible.

Versions v5.2, v5.3 and v5.4 of the Linux kernel are not compatible with OP-TEE 3.11.0. Launching the rootkit or the "xtest" application shipped with OP-TEE yields an error. Because of that, all functionalities are marked as failed and no further evaluation on that versions was conducted.

Neglecting the two error categories explained above, all of the tested Linux kernel versions are compatible with the rootkit. Multiple address translation functions were necessary to overcome significant changes in the ARM64-specific memory management [16]. Section 5.4.2 explained this process in detail. Further changes to the kernel impacting the compatibility of the rootkit with future versions need to be expected.

These results clearly show that generic rootkits utilizing the Arm TrustZone are possible and may impact Linux-based systems across kernel and OS recompilations and updates, even when state-of-the-art exploitation countermeasures such as randomization are enabled.

CHAPTER 7

Protection against Secure World Rootkits

Defensive techniques protect the normal world kernel and its user space applications from conventional rootkits introduced in Section 4.3. Due to the logical separation between the normal world and the secure world, existing defensive mechanisms face a protected environment as potential attacker. A theoretical discussion on the effectiveness of the mechanisms is provided in this chapter.

7.1 Injection of False-Positives

Targeted modifications of the memory content can be used to cause general assumptions of the rootkit implementation to fail. For example, an artificial but correctly aligned kernel image header instruction (see Listing 5.5) could be inserted before the actual start of the kernel image. Current implementation of the rootkit would not be able to differentiate between the artificial and true kernel image start. Execution of the rootkit would simply continue with the kernel image header detected at the lower physical address, causing later stages of the attack to fail. A similar approach could be used to break the search for the "swapper/0" task name described in Section 5.4.2.

7.2 Randomization

Randomization of addresses significantly complicates exploitation for an attacker. Instead of directly accessing fixed locations in memory, invariants are required to resolve the necessary addresses and offsets. Chapter 5 explained how such techniques can be implemented in the secure world.

ASLR is a feature of the Linux kernel randomizing the addresses of the stack, heap and shared libraries of normal world applications [114]. Position-independent Code (PIC) is a compiler feature that additionally covers the main executable code itself and the Procedure Linkage Table (PLT) [114]. Executables consisting only of code compiled with the PIC option enabled are called Position-independent Executables (PIEs) [64]. With these two protection features enabled, addresses of a process are considered to be fully randomized at runtime.

KASLR applies the idea of user space ASLR to the kernel. When KASLR is enabled, the kernel code is loaded at a randomized location at boot time [53]. All features presented in Chapter 5 applied a bruteforce search to find the kernel image header and effectively bypass KASLR.

Section 5.4.1 described a heuristic approach to find the initial page table. Localizing the initial page table is a relatively fragile step in the developed rootkit. PT-Rand [48] randomizes the location of the initial page table during the kernel startup, which would break the current memory carving implementation.

Another type of randomization supported by Linux is the randomization of kernel data structures such as `task_struct`. Field offsets are randomized at compile time by a GNU Compiler Collection (GCC) plugin. While the implementation presented in Chapter 5 is expected to be robust against structure randomization, the OP-TEE system did not boot when enabling this feature.

7.3 Integrity Checking

The Linux Kernel Runtime Guard (LKRG) is a Linux kernel module that adds integrity checks to the kernel at runtime to protect it from exploits [98]. A separate task list is maintained by the module to validate the integrity of the kernel's task list [52]. Overwriting credential pointers like demonstrated in Section 5.4.2 could be detected by the LKRG with this mechanism [99, 100]. In the same way as the kernel itself, the LKRG module is residing in memory accessible by the secure world. Because of this, the rootkit could first detect the credential pointer offsets in the `task_struct` list and then search for the addresses in the mapped memory as described in Section 5.4.1. Values can be overwritten in the `task_struct` instance and the LKRG memory immediately following each other, which results in a race condition. If both locations are overwritten with the same value before LKRG runs the integrity check, the exploitation was successful. Due to the fact that the relevant memory locations can be identified before violating the integrity constraint, the overwriting itself can be done fast and the attacker is likely to win the race.

Regular Control-flow Integrity (CFI) [1] mechanisms do not have any effect on the currently implemented data-only attacks. Code paths are not modified directly, but in general benign decisions are taken based on tampered data.

7.4 Hardware-based Measures

Removing the capability of the Arm TrustZone to access normal world memory would prevent all rootkit functions presented in this work. Disabling this feature would break compatibility with all existing software implementations relying on it. Therefore, this approach is not considered a viable option.

Zhou and Makris [172] analyzed a similar threat model as this work. Particularly, malware with full access to the normal world OS memory image was considered. A custom hardware component collecting information about the system was proposed to evade the possibility of software tampering. However, the actual data interpretation was done in a trusted software environment. Aligning this suggestion to a compromised Arm TrustZone is not considered effective but only another step in the arms race.

CHAPTER 8

Future Work and Research Directions

During this thesis, a proof-of-concept rootkit utilizing the Arm TrustZone was developed. Being the first step into this research field, various topics for future work were identified and are outlined in this chapter.

8.1 Evaluation of Defensive Mechanisms

Defensive mechanisms were discussed from a theoretical point of view in Chapter 7. Two immediate follow-up topics are viable.

The proof-of-concept implementation developed as part of this work can be benchmarked against the described defensive techniques. Possible bypasses may be investigated and implemented. Given the access to the physical memory, it is expected that at least some traditional defensive mechanisms can be rendered ineffective by the secure world following similar approaches as presented in this work. Evaluation of existing defensive techniques would help to design and implement more effective techniques.

Section 4.1.5 provided a survey of TEEs. As noted, all TEEs relevant for consumer devices currently rely on closed source secure world code. Effects of defensive techniques on proprietary TEEs could provide valuable insights on the internal behavior of these secure world implementations.

8.2 Improvements to the Developed Rootkit

Existing code of the developed rootkit can be improved in several aspects.

Dependencies to kernel internals can be reduced further. For example, the current implementation strictly requires a memory translation granule size of 4KB. Page table

management internals depend on this value. Configuration flags in the kernel image can be used to detect this property and adjust the page table walk algorithm accordingly to increase general compatibility [33].

Kernel symbols are resolved by identifying invariants in the kernel state. “Bits, Please!” [30] demonstrated the detection and interpretation of the kernel symbol table. Usage of this technique could enable access to additional kernel components and open up possibilities for the development of new rootkit features.

No internal state is maintained by the rootkit. Keeping once gathered information between invocations would improve performance of subsequent calls.

Memory carving described in Section 5.4.1 uses a rather inefficient search algorithm to identify the patterns in memory pages and blocks. Performance can be significantly improved by switching to an algorithm known to be more efficient [35, 90]. Improving performance would make the rootkit faster and less recognizable by detection systems based on resource usage.

Static strings are used by the memory carving mechanism. Dynamic file formats would benefit from a pattern-based matching. Yet Another Recursive Acronym (YARA) [163] and regular expressions [62] are well-established pattern systems that could be used for this purpose.

`TASK_DEAD` was mentioned as a more stealthy but in its trivial initial implementation unstable alternative for process starvation. An in-depth analysis of the Linux scheduler implementation could reveal options to avoid the caveats of this approach.

Chapter 6 explored the compatibility of the implemented rootkit with recent Linux kernel releases. Future releases of the kernel may adhere to changed concepts and protection mechanisms. Keeping the rootkit up to date while ensuring backwards compatibility with the currently supported versions of the kernel is a topic on its own.

The GlobalPlatform API is used to explicitly call the rootkit pseudo TA from the normal world. No concealment techniques or latency optimizations were put into effect as part of the proof-of-concept rootkit. Roth [124] proposed less suspicious ways of scheduling malware that do not require any interaction from the normal world.

EL1 of the secure world has been used as runtime environment for the rootkit. Further expanding the rootkit to EL3 would provide access to normal world registers [145]. Heuristic parts of the implementation could be replaced by information gathered from these registers. For example, register `TTBR1_EL1` contains the address of the initial page table (`swapper_pg_dir`) and the `VBAR_EL1` register contains the address of the EVT [7].

8.3 Addition of Rootkit Functionalities

Carving and manipulating normal world memory from the secure world has been thoroughly researched in this work. Next to the normal world, access to the physical memory

includes the secure world itself with its own OS. Assuming a memory corruption vulnerability in the secure world OS, similar techniques as proposed in this work could be implemented by malware for the secure world [59]. Other targets available in memory are images of previous boot stages (see Section 4.2.1).

Rootkit functionalities implemented for now only modify data to redirect legitimate code paths. Another attempt is the modification of kernel or user application code. Opcodes are written directly to the memory region representing instructions to be executed [48]. Attacks on the EVT have been shown before [40].

By having access to the physical memory and the CPU registers, the Arm TrustZone is able to directly communicate with hardware. Hardware components such as a network card can be directly interfaced with [124]. Potential use cases include sniffing network traffic invisible to the normal word. Similar work has already been done e.g., on the basis of Intel SMM [55, 56].

8.4 Deployment to Development Hardware and Consumer Devices

Possibilities to deploy custom secure world code were hinted in Section 5.1.2. Physical devices can be used to demonstrate the practical relevance of this work. Development boards that allow the deployment of custom secure world code are available. Running the rootkit on a consumer device such as a smartphone would require support from the vendor or a vulnerability in the firmware authenticity verification of that device.

Once the rootkit can be deployed to physical devices, its compatibility with the popular Linux-based Android OS can be investigated. Mobile applications could be evaluated as target but also for the use as normal world rootkit clients. Android assigns each application a unique user [96]. A secure world rootkit could allow extraction of information across application and user boundaries.

CHAPTER 9

Conclusion

Arm TrustZone is a security extension splitting the device into a normal (REE) and a secure (TEE) world. Sensitive services such as DRM are provided in an isolated environment by the secure world to the normal world. Physical memory including the regions used by the normal world can be accessed by the secure world [7].

Ideas of misusing the capabilities of the Arm TrustZone for rootkits have been brought up several years ago [124]. Since then, no specific implementation was proposed to the best of our knowledge. This work fills this gap and describes the implementation of a data-only rootkit residing in the secure world.

OP-TEE, an open-source secure world OS, was used as basis. The rootkit is implemented as a so-called "pseudo TA", that provides API-based access to its functionality. At the same time, it may use functionality that is part of the OP-TEE OS. Most notably, arbitrary normal world memory pages can be easily mapped into a shared space accessible by the secure world.

Invocation of the rootkit functions is handled by a normal world client using the aforementioned API. Even though the client is running as an unprivileged process in the normal world, it can provide valuable information to the secure world and alter the state of the normal world Linux kernel via system calls. Configuration options (e.g., header and footer of memory regions to identify, PIDs of target processes) are passed by the client to the secure world rootkit. For now, no latency optimizations or concealment techniques were implemented.

Deployment of custom secure world code to consumer devices is not trivially possible. Authentication checks ensure only software verified by the vendor can be run. To simplify research without specific development hardware, QEMU was chosen as a virtual target platform. OP-TEE provides scripts to build and run a complete environment in QEMU with an emulated Armv8 CPU. Verifying compatibility with development hardware or consumer devices is a topic for future research.

Exact source code, version and compilation configuration of the normal world kernel were assumed to be unknown. This scenario requires to reverse engineer kernel internals such as symbol addresses at runtime by investigating the raw physical memory. Invariants are used to partially recover addresses and offsets of kernel data structures [161].

Three rootkit functions were implemented in the scope of this work.

- Memory Carving
- Privilege Escalation
- Process Starvation

Searching for memory regions delimited by static strings is enabled by the memory carving module. Page tables of the normal world OS are identified, processed recursively and searched for a header and footer provided by the normal world client as invocation parameters. For demonstration purposes, the mapped normal world memory is searched for RSA private keys in the PEM format. Further processing of the found memory regions was out of scope for this work.

Escalation of privileges for regular normal world processes was implemented. The PID of an arbitrary target process is passed to the secure world by the normal world client. Initially, meta information about the kernel runtime needs to be detected. Once all information is reconstructed, the credential structure pointers of a chosen target process are overwritten with those of the privileged `init_task` called "swapper/0". All future children of the target process inherit the elevated privileges.

Starvation of processes was presented as final feature of the rootkit. A target task is specified via its PID and passed to the secure world. In addition, the desired state of the target process is provided as parameter. Similarly to the steps of the privilege escalation, meta information about the kernel data structures is required. Changing the state of a process precisely, lets the scheduler ignore the process and stall its execution. Disabling protection mechanisms like antivirus systems and IDSs are exemplary attack scenarios [70].

Compatibility of the rootkit was evaluated for all Linux versions supporting OP-TEE. Stability of the implementation can be estimated by observing the quantity of changes between subsequent releases. Results presented in Chapter 6 showed that the implementations of the privilege escalation and process starvation are stable across recent Linux releases. Multiple address translation mechanisms were implemented to keep compatibility after major changes within the ARM64-specific Linux memory management. Memory carving relies on a heuristic property introduced with version v4.20 of the kernel. Later kernel releases that were evaluated were compatible without modifying the implementation.

Verified by the results shown in this work, relevant properties required by the rootkit functionalities can be reverse engineered at runtime without insights into the normal world

kernel. Full access to the physical memory constitutes an inherent source of possibilities to analyze the normal world runtime state. Widespread usage of Arm CPUs in smartphones and their increasing significance for other types of devices such as notebooks turn the Arm TrustZone into a rewarding target for attackers.

Defensive techniques were discussed theoretically.

Systematic injection of false-positives breaks too general assumptions in the rootkit implementation. Incorrect detection of properties of the normal world kernel causes subsequent steps of attacks to fail.

Randomization increases the effort necessary for the rootkit. Bruteforce searches are applied to find cornerstones of the runtime state like the UEFI header. State invariants are used to overcome randomized memory locations. Credential pointers within the `task_struct` field can be recovered despite of varying offsets between kernel versions.

Integrity checks which are, e.g., employed by the LKRG, may protect from the initial version of the developed rootkit. However, given the full access to the physical memory, it is considered an arms race between developers of rootkits and defensive mechanisms. Memory representing the state of protection mechanisms may be manipulated to avoid detection of the rootkit.

Finally, it can be concluded that the Arm TrustZone is a viable environment for the development of rootkits. Even when ignoring direct access to the hardware, it was proven in this work that it is realistic to implement rootkit functionality supporting multiple versions of the Linux kernel. The results of this thesis highlight that improvements to the existing defensive mechanisms are urgently needed to mitigate against exploits targeting TEEs for Arm devices and to protect the normal world effectively against rootkits and malicious code residing in the Arm TrustZone.

CHAPTER 10

Appendix

A Linux Kernel Release Statistics Generation

```
#!/bin/sh
set -e

kernel_repository_path="$1"
kernel_releases_file="$2"

# Get first kernel release in the file
previous_release=$(head -n 1 "$kernel_releases_file")
# Get all remaining releases in the file
releases=$(tail -n +2 "$kernel_releases_file")

cd "$kernel_repository_path"

# Print the difference statistic for each release and its
   ↪ predecessor
for current_release in $releases; do
    echo -n "${current_release}: "
    git diff --ignore-all-space -l0 --shortstat "
       ↪ $previous_release".."$current_release"
    previous_release="$current_release"
done
```

Listing A.1: Script to generate release statistics.

```
v4.12
v4.13
```

```
v4.14
v4.15
v4.16
v4.17
v4.18
v4.19
v4.20
v5.0
v5.1
v5.2
v5.3
v5.4
v5.5
v5.6
```

Listing A.2: List of Linux kernel releases to consider in the statistics.

```
v4.13:  10676 files changed, 808313 insertions(+), 212165
   ↪ deletions(-)
v4.14:  23143 files changed, 610573 insertions(+), 336296
   ↪ deletions(-)
v4.15:  13162 files changed, 600291 insertions(+), 276768
   ↪ deletions(-)
v4.16:  11931 files changed, 491999 insertions(+), 297996
   ↪ deletions(-)
v4.17:  14227 files changed, 631104 insertions(+), 810344
   ↪ deletions(-)
v4.18:  12928 files changed, 508161 insertions(+), 606853
   ↪ deletions(-)
v4.19:  11593 files changed, 514241 insertions(+), 206654
   ↪ deletions(-)
v4.20:  11238 files changed, 615429 insertions(+), 248361
   ↪ deletions(-)
v5.0:  11887 files changed, 518937 insertions(+), 271423
   ↪ deletions(-)
v5.1:  11801 files changed, 512729 insertions(+), 255989
   ↪ deletions(-)
v5.2:  30524 files changed, 557864 insertions(+), 465517
   ↪ deletions(-)
v5.3:  13189 files changed, 918080 insertions(+), 328913
   ↪ deletions(-)
v5.4:  12378 files changed, 712146 insertions(+), 315234
   ↪ deletions(-)
```

```
v5.5:  11556 files changed, 548877 insertions(+), 232674
   ↪ deletions(-)
v5.6:  11408 files changed, 543044 insertions(+), 228000
   ↪ deletions(-)
```

Listing A.3: Output of the script in Listing A.1 for the releases in Listing A.2.

B Evaluation Scripts

```
diff --git a/qemu_v8.mk b/qemu_v8.mk
index 8a7ee48..02673b7 100644
--- a/qemu_v8.mk
+++ b/qemu_v8.mk
@@ -194,8 +194,8 @@ run-only:
        ln -sf $(ROOT)/out-br/images/rootfs.cpio.gz $(
           ↪ BINARIES_PATH)/
        $(call check-terminal)
        $(call run-help)
-       $(call launch-terminal,54320,"Normal World")
-       $(call launch-terminal,54321,"Secure World")
+       mkdir -p "${EVALUATION_LOG_DIRECTORY}/${
   ↪ EVALUATION_KERNEL_VERSION}"
+       unbuffer $(ROOT)/evaluation_controller.py 54320 54321
   ↪ 4444 "${EVALUATION_LOG_DIRECTORY}/${
   ↪ EVALUATION_KERNEL_VERSION}" &>"${EVALUATION_LOG_DIRECTORY
   ↪ }/${EVALUATION_KERNEL_VERSION}/evaluation_controller.log"
   ↪  &
        $(call wait-for-ports,54320,54321)
        cd $(BINARIES_PATH) && $(QEMU_PATH)/aarch64-softmmu/
           ↪ qemu-system-aarch64 \
                -nographic \
@@ -207,6 +207,7 @@ run-only:
                -bios bl1.bin \
                -initrd rootfs.cpio.gz \
                -kernel Image -no-acpi \
+               -qmp tcp:localhost:4444,server,nowait \
                -append 'console=ttyAMA0,38400 keep_bootcon
                   ↪ root=/dev/vda2' \
                $(QEMU_EXTRA_ARGS)
```

Listing B.4: Necessary changes to the "OP-TEE/build" repository [111] to run the evaluation.

```
#!/bin/sh
set -ex

repository_root_path="$1"
kernel_releases_file="$2"
evaluation_output_directory="$3"
export EVALUATION_LOG_DIRECTORY="${evaluation_output_directory
   ↪ }"
```

```
mkdir -p "$evaluation_output_directory"

while read release; do
    echo "Running evaluation for release ${release}"
    cd "${repository_root_path}/linux"
    git checkout "$release"
    cd "${repository_root_path}/build"
    export EVALUATION_KERNEL_VERSION="$release"
    mkdir -p "${evaluation_output_directory}/${release}"
    make -j clean > "${evaluation_output_directory}/${release}/
       ↪ clean.log" 2>&1 || true
    make run -j$(nproc) > "${evaluation_output_directory}/${
       ↪ release}/make.log" 2>&1
done <"$kernel_releases_file"
```

Listing B.5: Main script to run the evaluation.

```
#!/usr/bin/env python3
import enum
import json
import os
import re
import socket
import socketserver
import subprocess
import sys
import threading
import time


class ThreadedTCPServer(socketserver.ThreadingMixIn,
   ↪ socketserver.TCPServer):
    pass


class NormalWorldState(enum.Enum):
    NONE = 0
    LOGIN_PROMPT_RECEIVED = 1
    SHELL_PROMPT_RECEIVED = 2
    INITIAL_USER_RECEIVED = 3
    ROOTKIT_EXECUTION_FINISHED = 4
    FINAL_USER_RECEIVED = 5
    FAILED = 6
```

```
class NormalWorldHandler(socketserver.BaseRequestHandler):
    _LOGIN_PROMPT: bytes = b"buildroot login: "
    _LOGIN_USER: bytes = b"test"
    _SHELL_PROMPT: bytes = b"$ "
    _ID_COMMAND: bytes = b"id"
    _ROOTKIT_COMMAND: bytes = b"rootkit"

    _ID_OUTPUT_PATTERN: bytes = b"uid=[^\\r\\n]+"
    _PROCESS_STATUS_OUTPUT_PATTERN: bytes = b"child status:\\r
       ↪ \\n([^\\r\\n]+)"
    _LOG_FILENAME: str = "normal_world.log"

    log_directory = os.path.curdir

    def log(self, message):
        print("N: {}".format(message))

    def setup(self):
        self.state = NormalWorldState.NONE

    def handle(self):
        self.log("Received connection from {}".format(self.
           ↪ client_address[0]))

        all_data = b""

        with open(os.path.join(self.log_directory,
           ↪ NormalWorldHandler._LOG_FILENAME), "wb",
           ↪ buffering=0) as log_file:
            while True:
                data = self.request.recv(4096)
                if not data:
                    return

                all_data += data
                log_file.write(data)

                if self.state is NormalWorldState.NONE and
                   ↪ NormalWorldHandler._LOGIN_PROMPT in
                   ↪ all_data:
                    self.log("login prompt")
```

```
                self.state = NormalWorldState.
                    ↪ LOGIN_PROMPT_RECEIVED
                all_data = b""
                self.request.send(NormalWorldHandler.
                    ↪ _LOGIN_USER + b"\n")

            if self.state is NormalWorldState.
                ↪ LOGIN_PROMPT_RECEIVED and
                ↪ NormalWorldHandler._SHELL_PROMPT in
                ↪ all_data:
                self.log("shell prompt")
                self.state = NormalWorldState.
                    ↪ SHELL_PROMPT_RECEIVED
                all_data = b""
                self.request.send(NormalWorldHandler.
                    ↪ _ID_COMMAND + b"\n")

            if self.state is NormalWorldState.
                ↪ SHELL_PROMPT_RECEIVED and
                ↪ NormalWorldHandler._SHELL_PROMPT in
                ↪ all_data:
                m = re.search(NormalWorldHandler.
                    ↪ _ID_OUTPUT_PATTERN, all_data)
                if m:
                    initial_user_id = m.group(0).decode("
                        ↪ ascii")
                    self.log("initial user: {}".format(
                        ↪ initial_user_id))
                else:
                    self.log("could not detect initial user
                        ↪ ")
                self.state = NormalWorldState.
                    ↪ INITIAL_USER_RECEIVED
                all_data = b""
                self.request.send(NormalWorldHandler.
                    ↪ _ROOTKIT_COMMAND + b"\n")

            if self.state is NormalWorldState.
                ↪ INITIAL_USER_RECEIVED and
                ↪ NormalWorldHandler._SHELL_PROMPT in
                ↪ all_data:
                self.log("rootkit execution finished")
                self.state = NormalWorldState.
```

```
                    ↪ ROOTKIT_EXECUTION_FINISHED
                m = re.findall(NormalWorldHandler.
                    ↪ _PROCESS_STATUS_OUTPUT_PATTERN,
                    ↪ all_data)
                if m:
                    process_states = [s.decode("ascii") for
                        ↪  s in m]
                    self.log("process states: {}".format(
                        ↪ process_states))
                else:
                    self.log("could not detect process
                        ↪ state change: {}".format(all_data
                        ↪ ))
                    self.state = NormalWorldState.FAILED
                    return
                all_data = b""
                self.request.send(NormalWorldHandler.
                    ↪ _ID_COMMAND + b"\n")

            if self.state is NormalWorldState.
                ↪ ROOTKIT_EXECUTION_FINISHED and
                ↪ NormalWorldHandler._SHELL_PROMPT in
                ↪ all_data:
                m = re.search(NormalWorldHandler.
                    ↪ _ID_OUTPUT_PATTERN, all_data)
                if m:
                    final_user_id = m.group(0).decode("
                        ↪ ascii")
                    self.log("final user: {}".format(
                        ↪ final_user_id))
                else:
                    self.log("could not detect final user")
                self.state = NormalWorldState.
                    ↪ FINAL_USER_RECEIVED
                return

    def finish(self):
        self.log("finish called in state {}".format(self.state)
            ↪ )
        if self.state is NormalWorldState.FINAL_USER_RECEIVED:
            self.log("finished successfully")
            self.server.other_world_server.shutdown()
            self.server.shutdown()
```

```
            NormalWorldHandler.qmp_handler.quit_execution()
        elif self.state is not NormalWorldState.NONE:
            # Terminated in some intermediate state
            self.log("failed")
            self.server.other_world_server.shutdown()
            self.server.shutdown()
            NormalWorldHandler.qmp_handler.quit_execution()
        # Continue in NONE state to not stop on connection test

    def handle_error(self, request, client_address):
        self.log("handle_error")
        self.server.other_world_server.shutdown()
        self.server.shutdown()


class SecureWorldState(enum.Enum):
    NONE = 0
    BEFORE_PRIVILEGE_ESCALATION = 1
    AFTER_PRIVILEGE_ESCALATION = 2
    BEFORE_CHANGE_TASK_STATE = 3
    AFTER_CHANGE_TASK_STATE = 4
    BEFORE_MEMORY_CARVING = 5
    AFTER_MEMORY_CARVING = 6


class SecureWorldHandler(socketserver.BaseRequestHandler):
    _BEFORE_PRIVILEGE_ESCALATION: bytes = b"ELEVATE_PRIVILEGES:
       ↪  before"
    _AFTER_PRIVILEGE_ESCALATION: bytes = b"ELEVATE_PRIVILEGES:
       ↪ after"
    _BEFORE_CHANGE_TASK_STATE: bytes = b"CHANGE_TASK_STATE:
       ↪ before"
    _AFTER_CHANGE_TASK_STATE: bytes = b"CHANGE_TASK_STATE:
       ↪ after"
    _BEFORE_MEMORY_CARVING: bytes = b"MEMORY_CARVING: before"
    _AFTER_MEMORY_CARVING: bytes = b"MEMORY_CARVING: after"

    _LOG_FILENAME: str = "secure_world.log"

    log_directory = os.path.curdir

    def log(self, message):
        print("S: {}".format(message))
```

```
def setup(self):
    self.state = SecureWorldState.NONE

def handle(self):
    self.log("Received connection from {}".format(self.
       ↪ client_address[0]))

    all_data = b""

    with open(os.path.join(self.log_directory,
       ↪ SecureWorldHandler._LOG_FILENAME), "wb",
       ↪ buffering=0) as log_file:
        while True:
            data = self.request.recv(4096)
            if not data:
                return

            all_data += data
            log_file.write(data)

            if self.state is SecureWorldState.NONE and
               ↪ SecureWorldHandler.
               ↪ _BEFORE_PRIVILEGE_ESCALATION in all_data:
                self.state = SecureWorldState.
                   ↪ BEFORE_PRIVILEGE_ESCALATION
                all_data = b""

            if self.state is SecureWorldState.
               ↪ BEFORE_PRIVILEGE_ESCALATION and
               ↪ SecureWorldHandler.
               ↪ _AFTER_PRIVILEGE_ESCALATION in all_data:
                self.state = SecureWorldState.
                   ↪ AFTER_PRIVILEGE_ESCALATION
                all_data = b""

            if self.state is SecureWorldState.
               ↪ AFTER_PRIVILEGE_ESCALATION and
               ↪ SecureWorldHandler.
               ↪ _BEFORE_CHANGE_TASK_STATE in all_data:
                self.state = SecureWorldState.
                   ↪ BEFORE_CHANGE_TASK_STATE
                all_data = b""
```

```
                if self.state is SecureWorldState.
                   ↪ BEFORE_CHANGE_TASK_STATE and
                   ↪ SecureWorldHandler.
                   ↪ _AFTER_CHANGE_TASK_STATE in all_data:
                    self.state = SecureWorldState.
                       ↪ AFTER_CHANGE_TASK_STATE
                    all_data = b""

                if self.state is SecureWorldState.
                   ↪ AFTER_CHANGE_TASK_STATE and
                   ↪ SecureWorldHandler._BEFORE_MEMORY_CARVING
                   ↪  in all_data:
                    self.state = SecureWorldState.
                       ↪ BEFORE_MEMORY_CARVING
                    all_data = b""

                if self.state is SecureWorldState.
                   ↪ BEFORE_MEMORY_CARVING and
                   ↪ SecureWorldHandler._AFTER_MEMORY_CARVING
                   ↪ in all_data:
                    self.state = SecureWorldState.
                       ↪ AFTER_MEMORY_CARVING
                    all_data = b""

    def finish(self):
        self.log("finish called in state {}".format(self.state)
           ↪ )
        if self.state is SecureWorldState.AFTER_MEMORY_CARVING:
            self.log("finished successfully")
        elif self.state is not SecureWorldState.NONE:
            # Terminated in some intermediate state
            self.log("failed")
        # Continue in NONE state to not stop on connection test

    def handle_error(self, request, client_address):
        self.server.other_world_server.shutdown()
        self.server.shutdown()
        SecureWorldHandler.qmp_handler.quit_execution()


_HOST = "127.0.0.1"
```

```
class QmpHandler:
    _CONNECTION_ATTEMPTS = 10
    _CONNECTION_INTERVAL = 3

    def __init__(self, port: int):
        self.socket = self._connect(port)
        if not self.socket:
            raise Exception("Could not connect to QEMU on port
               ↪ {}".format(port))
        else:
            print("Connected to QEMU")
        # Needs to be executed before all other commands
        self._execute_command("qmp_capabilities")

    def _connect(self, port: int):
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        for i in range(QmpHandler._CONNECTION_ATTEMPTS):
            try:
                s.connect((_HOST, port))
            except:
                time.sleep(QmpHandler._CONNECTION_INTERVAL)
            else:
                return s

    def _execute_command(self, cmd: str):
        self.socket.sendall(json.dumps({"execute": cmd}).encode
           ↪ ("ascii") + b"\n")

    def continue_execution(self):
        self._execute_command("cont")

    def quit_execution(self):
        self._execute_command("quit")
        self.socket.recv(4096)
        self.socket.close()


if __name__ == "__main__":
    if len(sys.argv) != 5:
        print(__file__ + " <normal world port> <secure world
           ↪ port> <qmp port> <log directory>", file=sys.
           ↪ stderr)
```

```
        sys.exit(1)

    normal_world_port = int(sys.argv[1])
    secure_world_port = int(sys.argv[2])
    qmp_port = int(sys.argv[3])
    log_directory = sys.argv[4]

    os.makedirs(log_directory, exist_ok=True)
    NormalWorldHandler.log_directory = SecureWorldHandler.
       ↪ log_directory = log_directory

    print("Starting servers on ports {} and {}".format(
       ↪ normal_world_port, secure_world_port))

    with ThreadedTCPServer((_HOST, normal_world_port),
       ↪ NormalWorldHandler) as normal_world_server, \
         ThreadedTCPServer((_HOST, secure_world_port),
            ↪ SecureWorldHandler) as secure_world_server:
        normal_world_server.other_world_server =
           ↪ secure_world_server
        secure_world_server.other_world_server =
           ↪ normal_world_server

        normal_world_server_thread = threading.Thread(target=
           ↪ normal_world_server.serve_forever)
        secure_world_server_thread = threading.Thread(target=
           ↪ secure_world_server.serve_forever)

        normal_world_server_thread.start()
        secure_world_server_thread.start()
        print("Server threads running")

        # Must be initialized after the servers started to make
           ↪  sure the connection check is
        # passed and QEMU started.
        qmp_handler = QmpHandler(qmp_port)
        NormalWorldHandler.qmp_handler = SecureWorldHandler.
           ↪ qmp_handler = qmp_handler
        qmp_handler.continue_execution()

        normal_world_server_thread.join()
        secure_world_server_thread.join()
```

Listing B.6: Evaluation controller communicating with QEMU, recording the output and partially interpreting it.

List of Figures

List of Listings

List of Tables

Acronyms

AMT Advanced Management Technology. 2

API Application Programming Interface. 22, 33, 37–39, 43, 45, 49, 52, 66, 69

ASLR Address Space Layout Randomization. 7, 62

BYOD Bring Your Own Device. 1

CFI Control-flow Integrity. 62

CISC Complex Instruction Set Computer. 17

COFF Common Object File Format. 25

CPU Central Processing Unit. 8, 17, 24, 26, 27, 33, 45, 52, 54, 67, 69, 71, 87

DDoS Distributed Denial of Service. 15

DKOM Direct Kernel Object Manipulation. 31, 45, 52

DMA Direct Memory Access. 33

DRM Digital Rights Management. 2, 69

ECDSA Elliptic Curve Digital Signature Algorithm. 8

EDK II EFI Developer Kit II. 24

EL Exception Level. 19–21, 38, 39, 52, 57, 66, 87

ERET Exception Return. 19

EVT Exception Vector Table. 19, 66, 67

FIQ Fast Interrupt Request. 23

GCC GNU Compiler Collection. 62

HIEE Hardware-assisted Isolated Execution Environment. 2, 7

HPC Hardware Performance Counters. 10

HVC Hypervisor Call. 19

IDS Intrusion Detection System. 10, 32, 52, 70

IDT Interrupt Descriptor Table. 31

IoT Internet of Things. 1, 10

IP Intellectual Property. 17

IRQ Interrupt Request. 22

ISA Instruction Set Architecture. 17

KAISER Kernel Address Isolation to have Side channels Efficiently Removed. 11

KASLR Kernel Address Space Layout Randomization. 9, 41, 62

KPTI Kernel Page-Table Isolation. 11

LKRG Linux Kernel Runtime Guard. 62, 71

LSB Least Significant Bit. 42

ME Management Engine. 2

MIPS Microprocessor without Interlocked Pipelined Stages. 23

MMU Memory Management Unit. 19, 22

NIST National Institute of Standards and Technology. 13

OS Operating System. 1–4, 7–10, 19, 21–24, 26, 29, 30, 33, 35–37, 59, 63, 67, 69, 70

PCI Peripheral Component Interconnect. 33

PE Portable Executable. 25

PEM Privacy-Enhanced Mail. 40, 70, 89

PIC Position-independent Code. 62

PID Process Identifier. 25, 45, 47–50, 52, 53, 69, 70

PIE Position-independent Executable. 62

PLT Procedure Linkage Table. 62

QSEE Qualcomm Secure Execution Environment. 23

RAM Random Access Memory. 22

RCU Read-Copy-Update. 27

REE Rich Execution Environment. 21, 69

RISC Reduced Instruction Set Computer. 17

RSA Rivest–Shamir–Adleman. 40, 57, 58, 70, 89

SCADA Supervisory Control and Data Acquisition. 29

SCR Secure Configuration Register. 21

SGX Software Guard Extensions. 8, 9

SMC Secure Monitor Call. 19, 22, 23

SMM System Management Mode. 2, 9, 11, 67

SMP Symmetrical Multiprocessing. 45

SoC System on a Chip. 23

SVC Supervisor Call. 19

TA Trusted Application. 2, 3, 7, 21, 37, 38, 66, 69

TEE Trusted Execution Environment. 7, 8, 21, 23, 55, 57, 65, 69, 71

TTBR Translation Table Base Register. 20, 87

UEFI Unified Extensible Firmware Interface. 24, 25, 41, 43, 45, 71, 89

USB Universal Serial Bus. 9

VCS Version Control System. 56, 89

VM Virtual Machine. 8–10, 19, 30, 33

YARA Yet Another Recursive Acronym. 66

Bibliography

References

[1] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. "Control-flow integrity principles, implementations, and applications". In: *ACM Transactions on Information and System Security (TISSEC)* (2009). DOI: 10.1145/1609956.1609960.

[4] Izzat Alsmadi, Robert Burdwell, Ahmed Aleroud, Abdallah Wahbeh, Mahmoud Ali Al-Qudah, and Ahmad al omari. *The Ontology of Malwares*. 2018. ISBN: 9783319721187.

[7] *Arm Architecture Reference Manual: Armv8, for Armv8-A architecture profile*. 2020.

[8] *ARM Compiler toolchain: Developing Software for ARM Processors*. 2013.

[9] *ARM Cortex-A Series: Programmer's Guide for ARMv8-A*. 2015.

[11] *ARM Security technology: Building a secure system using TrustZone technology*. 2009.

[19] Ahmed Azab, Peng Ning, Jitesh Shah, Quan Chen, Rohan Bhutkar, Guruprasad Ganesh, Jia Ma, and Wenbo Shen. "Hypervision across worlds: Real-time kernel protection from the arm trustzone secure world". In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. 2014. DOI: 10.1145/2660267.2660350.

[20] Andrei Bacs, Cristiano Giuffrida, Bernhard Grill, and Herbert Bos. "Slick: an intrusion detection system for virtualized storage devices". In: *Proceedings of the 31st Annual ACM Symposium on Applied Computing*. 2016. DOI: 10.1145/2851613.2851795.

[21] Arati Baliga, Vinod Ganapathy, and Liviu Iftode. "Detecting kernel-level rootkits using data structure invariants". In: *IEEE Transactions on Dependable and Secure Computing* (2010). DOI: 10.1109/TDSC.2010.38.

[22] Ulrich Bayer, Paolo Milani Comparetti, Clemens Hlauschek, Christopher Kruegel, and Engin Kirda. "Scalable, behavior-based malware clustering". In: *NDSS*. 2009.

[23] Ulrich Bayer, Andreas Moser, Christopher Kruegel, and Engin Kirda. “Dynamic Analysis of Malicious Code”. In: *Journal in Computer Virology* (2006). DOI: 10.1007/s11416-006-0012-2.

[24] Fabrice Bellard. “QEMU, a fast and portable dynamic translator”. In: *USENIX Annual Technical Conference, FREENIX Track*. 2005.

[25] Jeffrey Bickford, Ryan O’Hare, Arati Baliga, Vinod Ganapathy, and Liviu Iftode. “Rootkits on Smart Phones: Attacks, Implications and Opportunities”. In: *Proceedings of the Eleventh Workshop on Mobile Computing Systems & Applications*. ACM, 2010. DOI: 10.1145/1734583.1734596.

[31] Emily Blem, Jaikrishnan Menon, and Karthikeyan Sankaralingam. “Power struggles: Revisiting the RISC vs. CISC debate on contemporary ARM and x86 architectures”. In: *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA2013)*. 2013. DOI: 10.1109/hpca.2013.6522302.

[32] Bill Blunden. *The Rootkit Arsenal: Escape and evasion in the dark corners of the system*. Jones & Bartlett Publishers, 2012. ISBN: 9781449626365.

[34] Daniel Bovet and Marco Cesati. *Understanding the Linux Kernel: from I/O ports to process management*. O’Reilly Media, Inc., 2005. ISBN: 9780596005658.

[35] Robert S Boyer and J Strother Moore. “A fast string searching algorithm”. In: *Communications of the ACM* (1977).

[36] Ferdinand Brasser, David Gens, Patrick Jauernig, Ahmad-Reza Sadeghi, and Emmanuel Stapf. “SANCTUARY: ARMing TrustZone with User-space Enclaves.” In: *NDSS*. 2019. DOI: 10.14722/ndss.2019.23448.

[37] Rory Bray, Daniel Cid, and Andrew Hay. *OSSEC host-based intrusion detection guide*. Syngress, 2008. ISBN: 9781597492409.

[38] David Bretthauer. “Open source software: A history”. In: *UConn Libraries Published Works* (2001).

[39] Robert Buhren, Julian Vetter, and Jan Nordholz. “The Threat of Virtualization: Hypervisor-Based Rootkits on the ARM Architecture”. In: *International Conference on Information and Communications Security*. 2016. DOI: 10.1007/978-3-319-50011-9_29.

[41] Claudio Canella, Michael Schwarz, Martin Haubenwallner, Martin Schwarzl, and Daniel Gruss. “KASLR: Break It, Fix It, Repeat”. In: *ASIA CCS 2020-Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*. 2020. DOI: 10.1145/3320269.3384747.

[42] David Cerdeira, Nuno Santos, Pedro Fonseca, and Sandro Pinto. “SoK: Understanding the Prevailing Security Vulnerabilities in TrustZone-assisted TEE Systems”. In: *2020 IEEE Symposium on Security and Privacy (SP)* (2020). DOI: 10.1109/sp40000.2020.00061.

[43] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers: Where the Kernel Meets the Hardware*. O'Reilly Media, Inc., 2005. ISBN: 9780596005900.

[48] Lucas Davi, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. "PT-Rand: Practical Mitigation of Data-only Attacks against Page Tables". In: *NDSS*. 2017. DOI: 10.14722/ndss.2017.23421.

[49] Francis David, Ellick Chan, Jeffrey Carlyle, and Roy Campbell. "Cloaker: Hardware Supported Rootkit Concealment". In: *2008 IEEE Symposium on Security and Privacy (sp 2008)*. 2008. DOI: 10.1109/SP.2008.8.

[50] Joel Dawson, Jeffrey Todd McDonald, Jordan Shropshire, Todd Andel, Patrick Luckett, and Lee Hively. "Rootkit detection through phase-space analysis of power voltage measurements". In: *2017 12th International Conference on Malicious and Unwanted Software (MALWARE)*. 2017. DOI: 10.1109/MALWARE.2017.8323953.

[51] Frank Dickson. "'Hardening' Android: Building Security into the Core of Mobile Devices". In: *Secure Networking in Frost & Sullivan* (2014).

[55] Shawn Embleton, Sherri Sparks, and Cliff Zou. "SMM rootkit: a new breed of OS independent malware". In: *Security and Communication Networks* (2013). DOI: 10.1002/sec.166.

[56] Shawn Embleton, Sherri Sparks, and Cliff Zou. "SMM Rootkits: A New Breed of OS Independent Malware". In: *Proceedings of the 4th International Conference on Security and Privacy in Communication Netowrks*. ACM, 2008. DOI: 10.1145/1460877.1460892.

[59] Fabian Fleischer, Marcel Busch, and Phillip Kuhrt. "Memory corruption attacks within Android TEEs: a case study based on OP-TEE". In: *Proceedings of the 15th International Conference on Availability, Reliability and Security*. 2020. DOI: 10.1145/3407023.3407072.

[60] Bogdan Florea. "Smartphone input/output interface for IoT applications". In: *2017 25th Telecommunication Forum (TELFOR)*. 2017. DOI: 10.1109/TELFOR.2017.8249402.

[62] Jeffrey Friedl. *Mastering regular expressions*. O'Reilly Media, Inc., 2006. ISBN: 9780596002893.

[63] Xinyang Ge, Hayawardh Vijayakumar, and Trent Jaeger. "Sprobes: Enforcing kernel code integrity on the trustzone architecture". In: *Proceedings of the Third Workshop on Mobile Security Technologies (MoST) 2014*. 2014.

[64] Hector Marco Gisbert and Ismael Ripoll. "On the effectiveness of NX, SSP, RenewSSP and ASLR against stack buffer overflows". In: *2014 IEEE 13th International Symposium on Network Computing and Applications*. 2014. DOI: 10.1109/NCA.2014.28.

[65] *GlobalPlatform Device Committee: TEE Protection Profile*. 2020.

[66] *GlobalPlatform Device Technology: TEE Client API Specification*. 2010.

[67] *GlobalPlatform Device Technology: TEE Sockets API Specification*. 2021.

[69] Michael Goodrich and Roberto Tamassia. *Introduction to Computer Security: Pearson New International Edition*. Pearson Higher Ed, 2013. ISBN: 9781292025407.

[70] Mariano Graziano, Lorenzo Flore, Andrea Lanzi, and Davide Balzarotti. "Subverting operating system properties through evolutionary DKOM attacks". In: *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. 2016. DOI: 10.1007/978-3-319-40667-1_1.

[71] Bernhard Grill, Andrei Bacs, Christian Platzer, and Herbert Bos. "'Nice Boots!'-A Large-Scale Analysis of Bootkits and New Ways to Stop Them". In: *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. 2015. DOI: 10.1007/978-3-319-20550-2_2.

[72] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. "KASLR is dead: long live KASLR". In: *International Symposium on Engineering Secure Software and Systems*. 2017. DOI: 10.1007/978-3-319-62105-0_11.

[73] Le Guan, Peng Liu, Xinyu Xing, Xinyang Ge, Shengzhi Zhang, Meng Yu, and Trent Jaeger. "Trustshadow: Secure Execution of Unmodified Applications with ARM TrustZone". In: *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*. 2017. DOI: 10.1145/3081333.3081349.

[74] Mauricio Gutierrez, Ziming Zhao, Adam Doupé, Yan Shoshitaishvili, and Gail-Joon Ahn. "CacheLight: Defeating the CacheKit Attack". In: *Proceedings of the 2018 Workshop on Attacks and Solutions in Hardware Security*. 2018. DOI: 10.1145/3266444.3266449.

[75] Weijie Han, Jingfeng Xue, Yong Wang, Lu Huang, Zixiao Kong, and Limin Mao. "MalDAE: Detecting and explaining malware based on correlation and fusion of static and dynamic characteristics". In: *Computers & Security* (2019). DOI: 10.1016/j.cose.2019.02.007.

[79] Alexander Ilic and Elgar Fleisch. *Augmented Reality and the Internet of Things*. Tech. rep. ETH Zurich, 2016.

[82] Xingbin Jiang, Michele Lora, and Sudipta Chattopadhyay. "Efficient and Trusted Detection of Rootkit in IoT Devices via Offline Profiling and Online Monitoring". In: *Proceedings of the 2020 on Great Lakes Symposium on VLSI*. 2020. DOI: 10.1145/3386263.3406939.

[83] Jestin Joy, Anita John, and James Joy. "Rootkit Detection Mechanism: A Survey". In: *Advances in Parallel Distributed Computing*. Springer Berlin Heidelberg, 2011. DOI: 10.1007/978-3-642-24037-9_36.

[86] Paul Ketelaar and Mark van Balen. "The smartphone as your follower: The role of smartphone literacy in the relation between privacy concerns, attitude and behaviour towards phone-embedded tracking". In: *Computers in Human Behavior* (2018). DOI: 10.1016/j.chb.2017.09.034.

[87] Asma Khatoon and Peter Corcoran. "Privacy concerns on Android devices". In: *2017 IEEE International Conference on Consumer Electronics (ICCE)*. 2017. DOI: 10.1109/ICCE.2017.7889265.

[88] Samuel King and Peter Chen. "SubVirt: Implementing malware with virtual machines". In: *2006 IEEE Symposium on Security and Privacy*. 2006. DOI: 10.1109/sp.2006.38.

[89] Richard Kissel. *Glossary of key information security terms, NISTIR 7298 Rev. 2*. 2013.

[90] Donald E Knuth, James H Morris Jr, and Vaughan R Pratt. "Fast pattern matching in strings". In: *SIAM journal on computing* (1977).

[92] Christopher Kruegel, William Robertson, and Giovanni Vigna. "Detecting kernel-level rootkits through binary analysis". In: *20th Annual Computer Security Applications Conference*. 2004. DOI: 10.1109/CSAC.2004.19.

[93] John Levine, Brian Culver, and Henry Owen. "A methodology for detecting new binary rootkit exploits". In: *Proceedings IEEE SoutheastCon*. 2003.

[96] Martina Lindorfer, Matthias Neugschwandtner, Lukas Weichselbaum, Yanick Fratantonio, Victor Van Der Veen, and Christian Platzer. "Andrubis–1,000,000 apps later: A view on current Android malware behaviors". In: *2014 third international workshop on building analysis datasets and gathering experience returns for security (BADGERS)*. 2014. DOI: 10.1109/BADGERS.2014.7.

[97] John Linn. *RFC 1421: Privacy Enhancement for Internet Electronic Mail: Part I: Message Encryption and Authentication Procedures*. Tech. rep. 1993.

[101] Aravind Machiry, Eric Gustafson, Chad Spensky, Christopher Salls, Nick Stephens, Ruoyu Wang, Antonio Bianchi, Yung Ryn Choe, Christopher Kruegel, and Giovanni Vigna. "BOOMERANG: Exploiting the Semantic Gap in Trusted Execution Environments". In: *NDSS*. 2017. DOI: 10.14722/ndss.2017.23227.

[102] Aleksandr Matrosov, Eugene Rodionov, David Harley, and Juraj Malcho. "Stuxnet under the microscope". In: *ESET LLC (September 2010)* (2010).

[103] Alex Matrosov, Eugene Rodionov, and Sergey Bratus. *Rootkits and Bootkits: Reversing Modern Malware and Next Generation Threats*. No Starch Press, 2019. ISBN: 9781593277161.

[105] Keith Miller, Jeffrey Voas, and George Hurlburt. "BYOD: Security and Privacy Considerations". In: *IT Professional* (2012). DOI: 10.1109/MITP.2012.93.

[106] Andreas Moser, Christopher Kruegel, and Engin Kirda. "Exploring multiple execution paths for malware analysis". In: *2007 IEEE Symposium on Security and Privacy (SP'07)*. 2007. DOI: 10.1109/sp.2007.17.

[107] Seyyedeh Atefeh Musavi and Mehdi Kharrazi. “Back to Static Analysis for Kernel-Level Rootkit Detection”. In: *IEEE Transactions on Information Forensics and Security* (2014). DOI: 10.1109/TIFS.2014.2337256.

[113] David Patterson and John Hennessy. *Computer Organization and Design ARM Edition: The Hardware Software Interface*. Morgan Kaufmann, 2016. ISBN: 9780128017333.

[114] Mathias Payer. *Too much PIE is bad for performance*. Tech. rep. ETH Zurich, 2012.

[116] J Aaron Pendergrass and Kathleen N McGill. “LKIM: The Linux Kernel Integrity Measurer”. In: *Johns Hopkins APL technical digest* (2013).

[117] Sandro Pinto and Nuno Santos. “Demystifying Arm TrustZone: A Comprehensive Survey”. In: *ACM Computing Surveys (CSUR)* (2019). DOI: 10.1145/3291047.

[120] Attia Qamar, Ahmad Karim, and Victor Chang. “Mobile malware attacks: Review, taxonomy & future directions”. In: *Future Generation Computer Systems* (2019). DOI: 10.1016/j.future.2019.03.007.

[122] Ronald Rivest, Adi Shamir, and Leonard Adleman. “Cryptographic communications system and method”. Pat. US4405829A. 1983.

[123] Dan Rosenberg. “Reflections on trusting trustzone”. In: *BlackHat USA* (2014).

[125] Xiaoyu Ruan. *Platform Embedded Security Technology Revealed: Safeguarding the Future of Computing with Intel Embedded Security and Management Engine*. Apress, 2014. ISBN: 9781430265719.

[126] Ethan Rudd, Andras Rozsa, Manuel Günther, and Terrance Boult. “A Survey of Stealth Malware Attacks, Mitigation Measures, and Steps Toward Autonomous Open World Solutions”. In: *IEEE Communications Surveys & Tutorials* (2017). DOI: 10.1109/comst.2016.2636078.

[132] Angeline Close Scheinbaum. *The dark side of social media: A consumer psychology perspective*. Routledge, 2017. ISBN: 9781138052567.

[133] Joshua Schiffman and David Kaplan. “The SMM Rootkit Revisited: Fun with USB”. In: *2014 Ninth International Conference on Availability, Reliability and Security*. 2014. DOI: 10.1109/ARES.2014.44.

[134] Michael Schwarz, Samuel Weiser, and Daniel Gruss. “Practical Enclave Malware with Intel SGX”. In: *CoRR* (2019). DOI: 10.1007/978-3-030-22038-9_9.

[135] Di Shen. “Attacking your "Trusted Core": Exploiting TrustZone on Android”. In: *Black Hat USA* (2015).

[137] Baljit Singh, Dmitry Evtyushkin, Jesse Elwell, Ryan Riley, and Iliano Cervesato. “On the detection of kernel-level rootkits using hardware performance counters”. In: *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*. 2017. DOI: 10.1145/3052973.3052999.

[138] Steven Skiena. *The Algorithm Design Manual*. Springer London, 2008. ISBN: 9781848000698.

[139] Dokyung Song, Felicitas Hetzelt, Dipanjan Das, Chad Spensky, Yeoul Na, Stijn Volckaert, Giovanni Vigna, Christopher Kruegel, Jean-Pierre Seifert, and Michael Franz. "PeriScope: An Effective Probing and Fuzzing Framework for the Hardware-OS Boundary". In: *NDSS*. 2019. DOI: 10.14722/ndss.2019.23176.

[140] Kedar Sovani. "Kernel Korner - Sleeping in the Kernel". In: *Linux Journal* (2005).

[141] William Stallings. *Operating Systems: Internals and Design Principles*. Pearson, 2017. ISBN: 9781292214290.

[142] William Stallings and Lawrie Brown. *Computer Security: Principles and Practice, Global Edition*. 4th ed. Pearson, 2018. ISBN: 9781292220611.

[143] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. "Driller: Augmenting Fuzzing Through Selective Symbolic Execution". In: *NDSS*. 2016. DOI: 10.14722/ndss.2016.23368.

[144] Patrick Stewin and Iurii Bystrov. "Understanding DMA malware". In: *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. 2012. DOI: 10.1007/978-3-642-37300-8_2.

[145] He Sun, Kun Sun, Yuewu Wang, Jiwu Jing, and Sushil Jajodia. "Trustdump: Reliable memory acquisition on smartphones". In: *European Symposium on Research in Computer Security*. 2014. DOI: 10.1007/978-3-319-11203-9_12.

[146] Pengfei Sun, Luis Garcia, and Saman Zonouz. "Tell Me More Than Just Assembly! Reversing Cyber-Physical Execution Semantics of Embedded IoT Controller Software Binaries". In: *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 2019. DOI: 10.1109/dsn.2019.00045.

[147] Peter Szor. *The Art of Computer Virus Research and Defense*. Pearson Education, 2005. ISBN: 9780321304544.

[148] Kimberly Tam, Salahuddin Khan, Aristide Fattori, and Lorenzo Cavallaro. "CopperDroid: Automatic Reconstruction of Android Malware Behaviors". In: *Ndss*. 2015. DOI: 10.14722/ndss.2015.23145.

[149] Andrew Tanenbaum and Herbert Bos. *Modern operating systems*. Pearson, 2015. ISBN: 9780133591620.

[155] Jiang Wang, Angelos Stavrou, and Anup Ghosh. "HyperCheck: A hardware-assisted integrity monitor". In: *International Workshop on Recent Advances in Intrusion Detection*. 2010. DOI: 10.1007/978-3-642-15512-3_9.

[156] Xueyang Wang and Ramesh Karri. "Reusing Hardware Performance Counters to Detect and Identify Kernel Control-Flow Modifying Rootkits". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2016). DOI: 10.1109/TCAD.2015.2474374.

[157] Brian Ward. *How Linux works: What every superuser should know*. No Starch Press, 2014. ISBN: 9781593275679.

[159] Johannes Winter, Paul Wiegele, Martin Pirker, and Ronald Tögl. “A Flexible Software Development and Emulation Framework for ARM TrustZone”. In: *International Conference on Trusted Systems*. 2011. DOI: 10.1007/978-3-642-32298-3_1.

[161] Jidong Xiao, Lei Lu, Haining Wang, and Xiaoyun Zhu. “HyperLink: Virtual Machine Introspection and Memory Forensic Analysis without Kernel Source Code”. In: *2016 IEEE International Conference on Autonomic Computing (ICAC)*. 2016. DOI: 10.1109/ICAC.2016.46.

[162] Xi Xiong, Donghai Tian, Peng Liu, et al. “Practical Protection of Kernel Integrity for Commodity OS from Untrusted Extensions”. In: *NDSS*. 2011.

[164] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. “Panorama: Capturing System-wide Information Flow for Malware Detection and Analysis”. In: *Proceedings of the 14th ACM Conference on Computer and Communications Security*. CCS ’07. 2007. DOI: 10.1145/1315245.1315261.

[165] Fengwei Zhang and Hongwei Zhang. “SoK: A Study of Using Hardware-assisted Isolated Execution Environments for Security”. In: *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016*. ACM, 2016. DOI: 10.1145/2948618.2948621.

[166] Ning Zhang, He Sun, Kun Sun, Wenjing Lou, and Thomas Hou. “CacheKit: Evading memory introspection using cache incoherence”. In: *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*. 2016. DOI: 10.1109/eurosp.2016.34.

[167] Ning Zhang, Kun Sun, Wenjing Lou, and Y Thomas Hou. “CaSE: Cache-assisted secure execution on arm processors”. In: *2016 IEEE Symposium on Security and Privacy (SP)*. 2016. DOI: 10.1109/SP.2016.13.

[168] Ning Zhang, Kun Sun, Deborah Shands, Wenjing Lou, and Thomas Hou. “TruSpy: Cache Side-Channel Information Leakage from the Secure World on ARM Devices”. In: *IACR Cryptology ePrint Archive* (2016).

[169] Ning Zhang, Ruide Zhang, Kun Sun, Wenjing Lou, Thomas Yizhao Hou, and Sushil Jajodia. “Memory Forensic Challenges Under Misused Architectural Features”. In: *IEEE Transactions on Information Forensics and Security* (2018). DOI: 10.1109/TIFS.2018.2819119.

[170] Xiaolan Zhang, Leendert van Doorn, Trent Jaeger, Ronald Perez, and Reiner Sailer. “Secure Coprocessor-based Intrusion Detection”. In: *Proceedings of the 10th Workshop on ACM SIGOPS European Workshop*. ACM, 2002. DOI: 10.1145/1133373.1133423.

[171] Zhangkai Zhang, Yueqiang Cheng, and Zhoujun Li. “Super Root: A New Stealthy Rooting Technique on ARM Devices”. In: *International Conference on Applied Cryptography and Network Security*. 2020. DOI: 10.1007/978-3-030-57878-7_17.

[172] Liwei Zhou and Yiorgos Makris. “Hardware-assisted rootkit detection via on-line statistical fingerprinting of process execution”. In: *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*. 2018. DOI: 10.23919/DATE.2018.8342267.

Online References

[2] *About OP-TEE - OP-TEE documentation*. URL: https://optee.readthedocs.io/en/3.10.0/general/about.html#history (visited on 2021-03-04).

[3] *access(2) - Linux manual page*. 2020. URL: https://man7.org/linux/man-pages/man2/access.2.html (visited on 2021-03-04).

[5] *application/efi*. 2016. URL: https://www.iana.org/assignments/media-types/application/efi (visited on 2021-03-04).

[6] *Architecting a Smarter World - Arm*. URL: https://www.arm.com/ (visited on 2021-03-04).

[10] *Arm Limited: Roadshow Slides Q2 2020*. 2020. URL: https://group.softbank/system/files/pdf/ir/presentations/2020/arm-roadshow-slides_q2fy2020_01_en.pdf (visited on 2021-03-04).

[12] *ARM-software/arm-trusted-firmware*. URL: https://github.com/ARM-software/arm-trusted-firmware/ (visited on 2021-03-04).

[13] *arm-trusted-firmware/platform_def.h at v2.3 - ARM-software/arm-trusted-firmware*. URL: https://github.com/ARM-software/arm-trusted-firmware/blob/v2.3/plat/qemu/qemu/include/platform_def.h#L75 (visited on 2021-03-04).

[14] *arm64: efi: add EFI stub*. 2014. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=3c7f255039a2ad6ee1e3890505caf0d029b22e29 (visited on 2021-03-04).

[15] *arm64: introduce VA_START macro - the first kernel virtual address*. 2015. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=127db024a7baee9874014dac33628253f438b4da (visited on 2021-03-04).

[16] *arm64: mm: Flip kernel VA space*. 2019. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=14c127c957c1c6070647c171e72f06e0db275ebf (visited on 2021-03-04).

[17] *arm64-stub.c « libstub « efi « firmware « drivers - kernel/git/torvalds/linux.git - Linux kernel source tree*. 2020. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/drivers/firmware/efi/libstub/arm64-stub.c?h=v5.6 (visited on 2021-03-04).

[18] *arm64/mm: move runtime pgds to rodata*. 2018. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=8eb7e28d4c642c310f25c18f80a44dd4b01c694e (visited on 2021-03-04).

[26] *Bits, Please!* URL: https://bits-please.blogspot.com/ (visited on 2021-03-04).

[27] *Bits, Please!: Exploring Qualcomm's Secure Execution Environment*. 2016. URL: https://bits-please.blogspot.com/2016/04/exploring-qualcomms-secure-execution.html (visited on 2021-03-04).

[28] *Bits, Please!: QSEE privilege escalation vulnerability and exploit (CVE-2015-6639)*. 2016. URL: https://bits-please.blogspot.com/2016/05/qsee-privilege-escalation-vulnerability.html (visited on 2021-03-04).

[29] *Bits, Please!: TrustZone Kernel Privilege Escalation (CVE-2016-2431)*. 2016. URL: https://bits-please.blogspot.com/2016/06/trustzone-kernel-privilege-escalation.html (visited on 2021-03-04).

[30] *Bits, Please!: War of the Worlds - Hijacking the Linux Kernel from QSEE*. 2016. URL: https://bits-please.blogspot.com/2016/05/war-of-worlds-hijacking-linux-kernel.html (visited on 2021-03-04).

[33] *booting.rst « arm64 « Documentation - kernel/git/torvalds/linux.git - Linux kernel source tree*. 2020. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/Documentation/arm64/booting.rst?id=v5.6 (visited on 2021-03-04).

[40] Amat Cama. *Corrupting the ARM Exception Vector Table*. 2014. URL: https://doar-e.github.io/blog/2014/04/30/corrupting-arm-evt/ (visited on 2021-03-04).

[44] *Core - OP-TEE documentation*. URL: https://optee.readthedocs.io/en/3.10.0/architecture/core.html#shared-memory (visited on 2021-03-04).

[45] *CPU Architecture - Arm Developer*. URL: https://developer.arm.com/architectures/cpu-architecture (visited on 2021-03-04).

[46] *CRED: Differentiate objective and effective subjective credentials on a task*. 2009. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=3b11a1decef07c19443d24ae926982bc8ec9f4c0 (visited on 2021-03-04).

[47] *CRED: Inaugurate COW credentials*. 2009. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=d84f4f992cbd76e8f39c488cf0c5d123843923b1 (visited on 2021-03-04).

[52] Jake Edge. *A "runtime guard" for the kernel [LWN.net]*. 2018. URL: https://lwn.net/Articles/749707/ (visited on 2021-03-04).

[53] Jake Edge. *Kernel address space layout randomization [LWN.net]*. 2013. URL: https://lwn.net/Articles/569635/ (visited on 2021-03-04).

[54] *EDK II - tianocore/tianocore.github.io Wiki*. URL: https://github.com/tianocore/tianocore.github.io/wiki/EDK-II (visited on 2021-03-04).

[57] Nicolas Fabretti. *Lexfo's security blog - CVE-2017-11176: A step-by-step Linux Kernel exploitation (part 4/4)*. 2018. URL: https://blog.lexfo.fr/cve-2017-11176-linux-kernel-exploitation-part4.html (visited on 2021-03-04).

[58] *Firmware Design - Trusted Firmware-A documentation*. URL: https://trustedfirmware-a.readthedocs.io/en/v2.3/design/firmware-design.html (visited on 2021-03-04).

[61] *Frequently Asked Questions - OP-TEE documentation*. URL: https://optee.readthedocs.io/en/3.10.0/faq/faq.html#q-where-is-the-linux-kernel-tee-driver (visited on 2021-03-04).

[68] *GlobalPlatform Homepage - GlobalPlatform*. URL: https://globalplatform.org/ (visited on 2021-03-04).

[76] Greg Houglund. *A Real NT rootkit*. 1999. URL: http://phrack.org/issues/55/5.html (visited on 2021-03-04).

[77] *Huawei - Building a Fully Connected, Intelligent World*. URL: https://www.huawei.com/ (visited on 2021-03-04).

[78] Nur Hussein. *Randomizing structure layout [LWN.net]*. 2017. URL: https://lwn.net/Articles/722293/ (visited on 2021-03-04).

[80] *init_task.c « init - kernel/git/torvalds/linux.git - Linux kernel source tree*. 2020. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/init/init_task.c?id=v5.6 (visited on 2021-03-04).

[81] *Introducing the Arm Cortex-A32 - Processors blog - Processors - Arm Community*. 2016. URL: https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/introducing-cortex-a32-arm-s-smallest-lowest-power-armv8-a-processor-for-next-generation-32-bit-embedded-applications (visited on 2021-03-04).

[84] Corey Kallenberg and Xeno Kovah. *How Many Million BIOSes Would you Like to Infect*. 2015. URL: http://legbacore.com/Research_files/HowManyMillionBIOSesWouldYouLikeToInfect_Whitepaper_v1.pdf (visited on 2021-03-04).

[85] *kernel/git/torvalds/linux.git - Linux kernel source tree*. 2020. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/refs/ (visited on 2021-03-04).

[91] Daniel Komaromy. *Unbox Your Phone — Part I.* 2018. URL: https://medium.com/taszksec/unbox-your-phone-part-i-331bbf44c30c (visited on 2021-03-04).

[94] *Linaro - Leading collaboration in the Arm Ecosystem.* URL: https://www.linaro.org/ (visited on 2021-03-04).

[95] *linaro-swg/linux: Linux kernel source tree.* URL: https://github.com/linaro-swg/linux/ (visited on 2021-03-04).

[98] *LKRG - Linux Kernel Runtime Guard.* 2020. URL: https://www.openwall.com/lkrg/ (visited on 2021-03-04).

[99] *LKRG Exploit detection main module.* 2020. URL: https://github.com/openwall/lkrg/blob/7eae8d3101ced6c004820ef6812fe82da4c9299c/src/modules/exploit_detection/p_exploit_detection.c#L977-L1003 (visited on 2021-03-04).

[100] *LKRG in a Nutshell.* 2020. URL: https://www.openwall.com/presentations/OSTconf2020-LKRG-In-A-Nutshell/OSTconf2020-LKRG-In-A-Nutshell.pdf (visited on 2021-03-04).

[104] Paul McKenney. *What is RCU, Fundamentally? [LWN.net].* 2007. URL: https://lwn.net/Articles/262464/ (visited on 2021-03-04).

[108] *Neo FreeRunner - Openmoko.* 2013. URL: http://wiki.openmoko.org/wiki/Neo_FreeRunner (visited on 2021-03-04).

[109] *NVD - CVE-2017-5689.* 2017. URL: https://nvd.nist.gov/vuln/detail/CVE-2017-5689 (visited on 2021-03-04).

[110] *OP-TEE.* URL: https://github.com/OP-TEE/ (visited on 2021-03-04).

[111] *OP-TEE/build: Makefiles to use OP-TEE on various platforms.* URL: https://github.com/OP-TEE/build (visited on 2021-03-04).

[112] *Open Portable Trusted Execution Environment - OP-TEE.* URL: https://www.op-tee.org/ (visited on 2021-03-04).

[115] *PE Format - Win32 apps.* 2020. URL: https://docs.microsoft.com/en-us/windows/win32/debug/pe-format (visited on 2021-03-04).

[118] *Platforms supported - OP-TEE documentation.* URL: https://optee.readthedocs.io/en/3.10.0/general/platforms.html (visited on 2021-03-04).

[119] *Project Zero: Trust Issues: Exploiting TrustZone TEEs.* 2017. URL: https://googleprojectzero.blogspot.com/2017/07/trust-issues-exploiting-trustzone-tees.html (visited on 2021-03-04).

[121] *QEMU v7 - OP-TEE documentation.* URL: https://optee.readthedocs.io/en/3.10.0/building/devices/qemu.html#qemu-v8 (visited on 2021-03-04).

[124] Thomas Roth. *Next generation mobile rootkits*. 2013. URL: https://hackinparis.com/data/slides/2013/Slidesthomasroth.pdf (visited on 2021-03-04).

[127] Joanna Rutkowska. *Intel x86 considered harmful*. 2015. URL: https://blog.invisiblethings.org/papers/2015/x86_harmful.pdf (visited on 2021-03-04).

[128] Keegan Ryan. *Hardware-Backed Heist: Extracting ECDSA Keys from Qualcomm's TrustZone*. 2019. URL: https://www.nccgroup.trust/globalassets/our-research/us/whitepapers/2019/hardwarebackedhesit.pdf (visited on 2021-03-04).

[129] Eloi Sanfelix. *TEE Exploitation: Exploiting Trusted Apps on Samsung's TEE*. 2019. URL: https://downloads.immunityinc.com/infiltrate2019-slidepacks/eloi-sanfelix-exploiting-trusted-apps-in-samsung-tee/TEE.pdf (visited on 2021-03-04).

[130] *sched/core: Allow putting thread_info into task_struct*. 2016. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=c65eacbe290b8141554c71b2c94489e73ade8c8d (visited on 2021-03-04).

[131] *sched.h « linux « include - kernel/git/torvalds/linux.git - Linux kernel source tree*. 2020. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/include/linux/sched.h?id=v5.6 (visited on 2021-03-04).

[136] *SierraTEE Virtualization for ARM TrustZone and MIPS*. URL: https://www.sierraware.com/open-source-ARM-TrustZone.html (visited on 2021-03-04).

[150] *task_struct: Allow randomized layout*. 2017. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=29e48ce87f1eaaa4b1fe3d9af90c586ac2d1fb74 (visited on 2021-03-04).

[151] *tianocore/edk2: EDK II*. URL: https://github.com/tianocore/edk2 (visited on 2021-03-04).

[152] *Trusted Applications - OP-TEE documentation*. URL: https://optee.readthedocs.io/en/3.10.0/architecture/trusted_applications.html (visited on 2021-03-04).

[153] *Trustonic: Mobile Application Protection, Device Security & IoT Security*. URL: https://www.trustonic.com/ (visited on 2021-03-04).

[154] *Trusty TEE | Android Open Source Project*. URL: https://source.android.com/security/trusty (visited on 2021-03-04).

[158] *What is RCU? – "Read, Copy, Update" - The Linux Kernel documentation*. 2020. URL: https://www.kernel.org/doc/html/v5.6/RCU/whatisRCU.html (visited on 2021-03-04).

[160] *Wireless Technology & Innovation | Mobile Technology | Qualcomm.* URL: https://www.qualcomm.com/ (visited on 2021-03-04).

[163] *YARA - The pattern matching swiss knife for malware researchers.* 2020. URL: https://virustotal.github.io/yara/ (visited on 2021-03-04).