TU WIEN Informatics

Threats and Limitations of an ARM Trustzone-based Rootkit Attacking the Android Binder

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering und Internet Computing

eingereicht von

Raphael Kiefmann, BSc
Matrikelnummer 01425609

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Thomas Grechenig
Mitwirkung: Clemens Hlauschek

Wien, 28. August 2024

Unterschrift Verfasser

Unterschrift Betreuung

TU WIEN Informatics

Threats and Limitations of an ARM Trustzone-based Rootkit Attacking the Android Binder

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering und Internet Computing

by

Raphael Kiefmann, BSc
Registration Number 01425609

to the Faculty of Informatics

at the TU Wien

Advisor: Thomas Grechenig
Assistance: Clemens Hlauschek

Vienna, 28th August, 2024

Signature Author

Signature Advisor

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

Threats and Limitations of an ARM Trustzone-based Rootkit Attacking the Android Binder

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering und Internet Computing

eingereicht von

Raphael Kiefmann, BSc
Matrikelnummer 01425609

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

Betreuung: Thomas Grechenig

Wien, 28. August 2024

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

Erklärung zur Verfassung der Arbeit

Raphael Kiefmann, 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.
Ich erkläre weiters, dass ich mich generativer KI-Tools lediglich als Hilfsmittel bedient habe und in der vorliegenden Arbeit mein gestalterischer Einfluss überwiegt. Im Anhang „Übersicht verwendeter Hilfsmittel“ habe ich alle generativen KI-Tools gelistet, die verwendet wurden, und angegeben, wo und wie sie verwendet wurden. Für Textpassagen, die ohne substantielle Änderungen übernommen wurden, haben ich jeweils die von mir formulierten Eingaben (Prompts) und die verwendete IT- Anwendung mit ihrem Produktnamen und Versionsnummer/Datum angegeben.

Wien, 28. August 2024

Raphael Kiefmann

Acknowledgements

I want to thank everyone involved in the process of writing this thesis. This includes family, especially my sister, friends, and colleagues who partook in my thesis journey that finally comes to an end after all this time.

I also would like to express special gratitude to a good friend of mine. He eventually sprinted away at the end of our academic run, encouraging me from the finish line with his wise words of sarcasm to finish this very piece.

Finally, I want to thank my colleagues Clemens, Daniel, Florian, Andreas, Richard, and Vanessa for their valuable input for this thesis.

Kurzfassung

Die Arm TrustZone ist eine hardwaregestützte Sicherheitserweiterung in Arm-basierten Prozessoren, die die Systemsicherheit verbessert, indem sie unter anderem das Speichern von sensitiven Daten und die Ausführung von sicherheitskritischem Code in einer *secure world* erlaubt. Im Vergleich zu einer *normal world*, können nur vorabbestimmte Akteure auf die Daten und Rechenleistung der *secure world* zugreifen.

Im Jahr 2013, hat ein Forscher Ideen präsentiert, was ein böswilliger Angreifer erreichen könnte, falls die Möglichkeit besteht, ein Rootkit in die Arm TrustZone einzuschleusen. Code in einer *secure world* hat kompletten Zugriff auf die Ressourcen eines Systems und ein Angreifer könnte dadurch die Integrität eines Systems beschädigen.

Nach unserem besten Wissen sind 8 Jahre zwischen der Idee eines Arm TrustZone Rootkits und einer ersten Implementierung vergangen. Die Implementierung hat gezeigt, dass ein Angreifer unter anderem die Möglichkeit hat, heimlich einem Prozess mehr Rechte zu ermöglichen.

Diese Diplomarbeit setzt diese Arbeit fort. Im Zuge dieser Arbeit wird der Android Binder attackiert, der zentrale Inter-Prozess Kommunikation Mechanismus von Android. Dieser Mechanismus ist zentral für den Austausch von Daten zwischen Android Applikationen. Der Binder überträgt die sensibelsten Nutzereingaben, wie Passwörter und Bankdaten. Diese Daten mitzuschneiden, führt zu einer schweren Verletzung der Vertraulichkeit, und kann zu Geldverlust, Identitätsdiebstahl oder Erpressung führen.

Diese Arbeit beschäftigt sich mit der Implementierung eines Arm TrustZone Rootkits für das Hikey 960 Entwicklungsboard. Der Hikey 960 wird mit Android, einer Linux-Distribution, betrieben. Zusätzlich beleuchtet diese Arbeit die notwendigen Schritte für das Design des Rootkits, Mechanismen zur Erkennung von Linux Kernel Strukturen sowie die Möglichkeiten und Einschränkungen von Arm TrustZone Rootkits.

Da die Position von Linux Kernel Strukturen von der Wahl des Compilers und den applizierten Optimierungen abhängt, beschränkt sich die Implementierung auf eine kleine Auswahl von Linux Kerneln. Eine generische Lösung hätten den Rahmen dieser Arbeit gesprengt.

Abstract

The Arm TrustZone is a hardware security extension in Arm-based processors that improves system security by allowing, e.g. to move sensitive data and security-critical computations to the *secure world*. Compared to the normal world, only a pre-defined set of actors can access data and computing power in a *secure world*.

In 2013, a researcher presented ideas of what a malicious actor could accomplish by deploying a rootkit in the Arm TrustZone. Code in the *secure world* has access to all of a system's resources, and an attacker could potentially cause complete havoc.

To the best of our knowledge, 8 years passed between the initial conception of the idea of an Arm TrustZone-based rootkit and the first implementation of an Arm TrustZone-based rootkit. The proof-of-concept has shown that an attacker would be able to covertly elevate privileges, among other things.

This thesis continues this work, attacking a different component: the Android Binder, Android's main mechanism for inter-process communication. This mechanism is central to the data exchange between Android applications. The Binder transceives even the most sensitive data, such as passwords and banking data. Being able to intercept data undermines the confidentiality of user data, allowing to steal money, steal identities, or make users prone to blackmail.

This thesis presents a proof-of-concept Arm TrustZone rootkit for the Hikey 960 development board. The Hikey 960 runs on Android, a Linux distribution. Alongside the proof-of-concept, this thesis provides the theory behind the design of such a rootkit, mechanisms to detect Linux kernel structures, such as functions, and a discussion of threats and limitations of Arm TrustZone-based rootkits.

Due to the location of Linux kernel structures being dependent on the chosen compiler and selected optimisations, the proof-of-concept targets only a small set of Linux kernels. A generic solution exceeds the scope of this thesis.

Contents

CHAPTER 1

Introduction

The Arm TrustZone is a security extension for Arm's instruction set architecture that aims to improve a system's security. This allows to establish a secure environment, also known as a Trusted Execution Environment (TEE), which only authorised actors can access, in combination with a secure operating system.

TEEs manage resources, such as memory and CPUs, that operating systems in the normal world system, such as Linux cannot, access. The policies and features that enforce access restrictions commonly cannot be modified at runtime. For example, the OP-TEE, a security solution building on top of the Arm TrustZone, requires developers to enable the secure storage feature at compile time [193]. Thus, changing the features and policies requires an update of the system firmware. Arm's firmware reference implementation, e.g. implements an update process that requires an update to be properly signed to boot [16]. To properly sign an update, an attacker would need to get hold of the cryptographic keys used by the developers.

In the last couple of years, the number of Arm-based processors that contain the Arm TrustZone has steadily increased, due to an increasing number of use cases, among other things. For example, Android relies on hardware-backed security elements [200, 8] to implement a key storage system, the Android Keystore [8]. In addition, Android offers the Trusty TEE [200] to implement custom functionality to, e.g. secure mobile payments and fingerprint processing. Another example of a manufacturer that uses the Arm TrustZone to secure their systems is Samsung. Samsung Knox [165], similarly to Android, uses Arm's security extension to, e.g., create their secure key storage, secure boot mechanism and implements a safe storage for banking data required for mobile payments. These mechanisms are generally optimised for specific environments and tasks, and cannot be extended to support custom use cases.

In addition to the previously mentioned use cases, various projects use the Arm TrustZone to implement other security critical functionalities. Projects such as OP-TEE [137] allow

developers to implement custom applications for TEEs, called Trusted Applications (TAs), on their own. For example, in 2016 Brown [31] presented an implementation of Widevine, a *Digital Rights Management* solution for video and audio files, that builds atop the OP-TEE. Compared to the implementation found in Android, where Widevine is part of the Android system [6], the OP-TEE-based approach is harder to bypass. Key material does not leave the memory area that is inaccessible to the Android system.

In 2019, the OP-TEE had three critical security issues (CVE-2019-1010296 [124], CVE-2019-1010297 [125], and CVE-2019-1010298 [126]). Using these vulnerabilities, attackers would have been able to execute arbitrary code in the OP-TEE.

This poses a fundamental problem: What if an attacker happens to gain access to the TEE and can execute arbitrary code using vulnerabilities?

1.1 Goals

Rootkits are a well-researched topic, and multiple works [43, 148, 121, 211, 74] have addressed this subject. Rootkits can affect different components of an operating system, like the kernel [42], or even the underlying hardware of a system, such as performance counters [180] and a system's hypervisor [155].

The Arm TrustZone is a rather new mechanism and Arm first integrated the security extension in 2004 [3]. In addition, until the release of Xilinx's development board [154] in 2011, development boards were not easily available to the public. Thus, there has been less research into rootkits in TEEs, compared to other types of rootkits. To the best of our knowledge, the only work that presented a working rootkit in the Arm TrustZone has been the work of Marth et al. [118]. Marth et al. show how a process' structures in the memory could be manipulated to gain root access.

This thesis aims to answer the question of what impact a rootkit in the Arm TrustZone that intercepts data from the Android Binder has. To answer this question, we create a proof-of-concept rootkit for the OP-TEE that can intercept data from the Android Binder, a central component of Android's inter-process communication. This proof-of-concept targets the Hikey 960 [5], a development board that runs Android and uses is a reference platform for the OP-TEE development.

The purpose of the proof-of-concept rootkit is to answer the following research questions:

1. How can a rootkit identify Linux kernel structures from the ARM TrustZone?
2. Where does malicious code need to be placed at runtime so that it is properly executed?
3. How can an ARM TrustZone-based rootkit manipulate the Android Binder to intercept data?

These questions lead up to the question: What threats do ARM TrustZone-based rootkits pose?

1.2 Methodological Approach

First, we conduct comprehensive academic literature research. In addition to academic literature, further sources, such as code repositories and blog posts, are part of the research. These sources contain a lot of relevant information that the respective authors published outside academic venues.

The main part of the thesis is the design and development of a proof-of-concept rootkit to answer the research questions. The proof-of-concept targets the OP-TEE [135], an implementation of a trusted execution environment, which targets the Hikey 960 development board. The methodology of the development matches the four three steps of the 5-step software development life cycle that Grechenig et al. [68] describe: requirement analysis, design, and implementation.

In addition to a regular analysis that needs to be made for software engineering projects, such as the selection of a proper build system and programming language, various fundamental technologies have to be researched.

For this thesis, a good understanding of the Linux kernel, the Android Binder, the Arm TrustZone, and the OP-TEE is required. The firmware of the Hikey 960, the core of the OP-TEE to be more specific, embeds the proof-of-concept rootkit. Even a small bug in the implementation could lead to an error that the system cannot recover from, resulting in a restart of the development board. Thus, it is highly important to ensure that the code runs and causes no side effects that could lead to a restart of the development board.

Another significant component of the analysis step is to understand the Arm assembly language and topics such as Arm's calling convention. The analysis provides the required knowledge to develop shellcode snippets that are needed to, e.g. forward data from the Linux kernel to the Arm TrustZone.

Finally, prebuilt Linux kernels for the Hikey 960 are reverse-engineered to devise a proper entry point for the rootkit, which can be reliably found across multiple kernel versions.

Based on this analysis, requirements for the proof-of-concept are determined. These requirements address topics, such as the range of functions, the selection of a search algorithm, and the performance overhead caused by the additional code execution.

Followed by the outline of requirements, a design for the proof-of-concept is made. In the design phase we construct that consists of a small set of functions to verify that fundamental mechanisms, such as the communication between the Linux kernel and the TEE are properly working. The implementation phase succeeds the design phase. The initial implementation of the rootkit and its tests, triggers the later stages of the software development life cycle. Through alternating phases of implementation and testing, the

rootkit and the shellcode gradually improve until the rootkit can intercept data from the Android Binder.

The resulting proof-of-concept rootkit eventually answers the research questions.

1.3 Structure

The related work is presented in chapter 2. Chapters 3 to 8, outline the fundamental knowledge base for this work. To go more into detail, chapter 3 discusses some IT-security fundamentals. Chapter 4 introduces the fundamentals of rootkits and presents various implementations. In chapter 5 the Arm architecture's technical details and some general information regarding CPU architecture are provided. Chapter 6 presents the Arm TrustZone alongside some trusted execution environments. Chapter 7 addresses the fundamentals of the Linux kernel; followed by a chapter discussing the Android Binder in detail.

In chapter 9 the proof-of-concept is presented, followed by a discussion of the threats and limitations this type of rootkit has.

The chapter 11 outlines future work showing, e.g., what further mechanisms are useful to an Arm TrustZone-based rootkit. Lastly, chapter 12 concludes this thesis and gives an outlook.

CHAPTER 2

Related Work

Most topics in IT security are in a steady arms race between attackers and defenders; one side improving their mechanisms to attack more easily and the other side improving defensive and detection mechanisms to increase a system's security. Due to this never-ending back and forth, there is various academic work that is related, e.g., to the implementation of rootkits and countermeasures to prevent and detect them.

To the best of our knowledge, only a single published academic work addresses rootkits in the Arm TrustZone and also presents a proof-of-concept. Marth et al. [118] developed an Arm TrustZone-based rootkit, which, among other things, can give processes additional privileges by manipulating a process' structure in the memory.

2.1 Exploiting the Arm TrustZone

Multiple papers address the exploitation of the ARM TrustZone itself.

For example, Machiry et. al. [114] present an attack that uses the Arm TrustZone to access memory that should be inaccessible to an attacker. The attack exploits the fact that trusted applications are commonly unaware of properties such as the ownership of a certain memory region.

Chen et al. [36] present a trustlet, a trusted application, downgrade attack that leverages exploits found in a smartphone's firmware.

Shakevsky, Ronen & Wool [170] have shown several attacks on Samsung's TrustZone firmware, which is responsible for securing keys and mobile payment data, among other things. They analysed the code by reverse engineering the firmware of numerous Samsung devices.

Shen [171] found a vulnerability in HiSilicon's TEE that allows to execute shellcode in the Arm TrustZone.

In his work, Rosenberg [159] presents an attack on the Qualcomm Secure Execution Environment that various Android devices contained at the time of the work's publication in 2014.

2.2 Attacking Systems Using Rootkits

Rootkits are a well-researched topic. There is various academic work that presents implementation (mis-)using all kinds of technology found in a computer system.

In 2008 David et. al [43] presented a non-persistent rootkit deployed on Arm-based systems. The rootkit uses various hardware state modifications to operate and hide itself.

Spisak [180] takes advantage of hardware found in x86- and Arm-based systems to create a rootkit that is more performant compared to a solely software-based solution.

Song et. al [178] explore a Linux rootkit that allows the execution of malicious code by exploiting the memory address mapping table structure.

You & Noh [224] presented a Linux kernel rootkit for the Android platform.

To the best of our knowledge, Roth [160] was the first to publicly propose the idea of an ARM TrustZone-based rootkit. Roth's work outlines the approach that would be needed to develop an Arm TrustZone-based rootkit.

2.3 Securing Systems Against Rootkits

At the same time the hiding mechanisms of rootkits improved, the mechanisms to detect also improved.

2.3.1 Detecting Rootkits

The detection of rootkits is a vital step in identifying systems infected by rootkits so that they can be studied.

For example, the work of Tian et al. [197] and Wang et al. [210] use virtualisation of a system in combination with machine learning to detect rootkits.

The tool of Wang et al. [208] can detect rootkits in guest Virtual Machines (VMs) leveraging hardware performance counters.

McDonald et al. [121] use a system's embedded power sensors to detect suspicious power spikes that identify rootkits.

Singh et al. [176] use a similar approach, but combine a system's hardware performance counters with machine learning to detect rootkits.

Zhou et al. [231] also use a hardware-assisted approach that the authors enhanced by machine learning.

Pham, Marion & Heuser [148] created a mechanism to detect Linux kernel rootkits on Arm- and MIPS-based Internet of Things (IoT) devices.

Han et. al [73] developed a mechanism to detect advanced persistent threats.

The kernel integrity check of Heo et.al [78] identifies issues by analysing the memory traffic.

2.3.2 Ensuring System Integrity

Another way to combat rootkits is to prevent their deployment in the first place. System integrity checks can help prevent the installation of malicious software.

The tool developed by Zhou et al. [230] uses functionality found in Intel processors to harden a system during runtime against rootkits. Their tool allows updating running kernels, allowing to patch security vulnerabilities.

The tool of Fend et. al [50] uses a behaviour pattern analysis to find system integrity violations during runtime.

Surendrababu [188] researched measures providing system integrity protection and their limitations.

Ha et al. [69] present a hardware mechanism for RISC-V processors to prevent modifications to the Linux kernel at runtime.

2.4 Securing Systems Using the Arm TrustZone

Ge, Vijayakumar & Jaeger [57] presented *SPROBES*, an instrumentation mechanism for the Linux kernel. SPROBES consists of 2 components; probes in 12 places of a Linux kernel and an Arm TrustZone-based component. Once a system passes over a probe, leading to its execution, the control flow changes to the component in the Arm TrustZone. The trustlet verifies the legitimacy of the function execution in the normal world. Theoretically, this approach could be used as a rootkit too though, as the placed probes could be used to intercept data.

Already in 2006, Hussin, Edwards, and Colton [83] sought ways to secure data on a mobile device using Nokia's then-mobile OS *Symbian*. An application in the *secure world* validates the authenticity of an application before during the installation process.

Brite, Duarte, and Santos [30] explored how to secure cloud-based image processing using the Arm TrustZone.

Huar et al. [81] researched how trusted execution environments (TEE) can be virtualised to provide one for each virtual machine running on a system.

Similarly, Kwon et al. [97] focus on virtualising the Arm TrustZone, but instead of providing a virtual TEE for VMs, their tool provides a virtual TEE for each trusted

application. The virtualisation isolates the trusted applications from each other, providing better protection against attacks.

Chan, Pasco, & Cheng [34] use a blockchain-based approach to ensure the system integrity of autonomous vehicles.

Santos et al. [166] developed a language runtime that can be used in Arm's TEE using a stripped-down version of the Mono C# runtime.

Azab et al. [22] were able to secure Linux-based systems using the Arm TrustZone.

Song et al. [177] built a mechanism that ensures an application's integrity by checking if the about-to-be-executed application matches a reference hash.

2.5 The Android Binder

Due to its importance in the Android system, the Binder is the subject of various papers.

Lemos et al. [104] collected data transported via the Binder. A machine learning-based algorithm uses the collected data to identify malware via the Binder.

Shen et al. [172] use virtualisation mechanisms found in Armv7-A processors to protect data transmitted in the Binder from attackers.

The tool *BinderCracker* of Feng & Shin [48] uses the Binder to fuzz Android's system services by invoking Remote Procedure Calls (RPCs) found in the system services.

Similarly, Xiang et al. [218] also fuzzed systems services to attack Android devices. Compared to BinderCracker, Xiang et al. reverse the control flow; the Android system services invoke the RPCs implemented by Xiang et al.

The fuzzing mechanism developed by Liu et al. [108] searches interfaces and automatically builds test cases based on the gathered data.

Artenstein and Revivo [21] created a rootkit that intercepts data from the Binder, using a kernel rootkit.

CHAPTER 3

IT-Security Fundamentals

For a good understanding of this thesis, an understanding of some IT-security fundamentals is required. The following chapter gives an overview of several fundamental IT-security concepts.

3.1 Information Security

Information security is an important part of IT-security and addresses the processing of data. A common termin in information security is the “CIA Triad” [113], depicted in Figure 3.1.

The CIA in CIA Triad is an abbreviation for the words“confidentiality”, “integrity”, and “availability”. Together, these three properties are crucial for the proper protection of information.

3.1.1 Confidentiality

Confidentiality is a property that allows only authorised entities, such as people or processes, to get access to data. Ensuring the confidentiality of data can be achieved in various ways, such as organisational measures (e.g. only give the people access that need it) or technical measures (data encryption using cryptographic systems, checking the access granted to a user, etc.)

Cryptographic systems rely on the confidentiality of certain components in the system, namely the keys. In 1883, the Dutch linguist August Kerckhoff stated the following fundamental requirement for cryptographic systems:

“The system must not require secrecy and can be stolen by the enemy without causing trouble” [147].

This is known as Kerckhoff’s principle and roughly translates to:

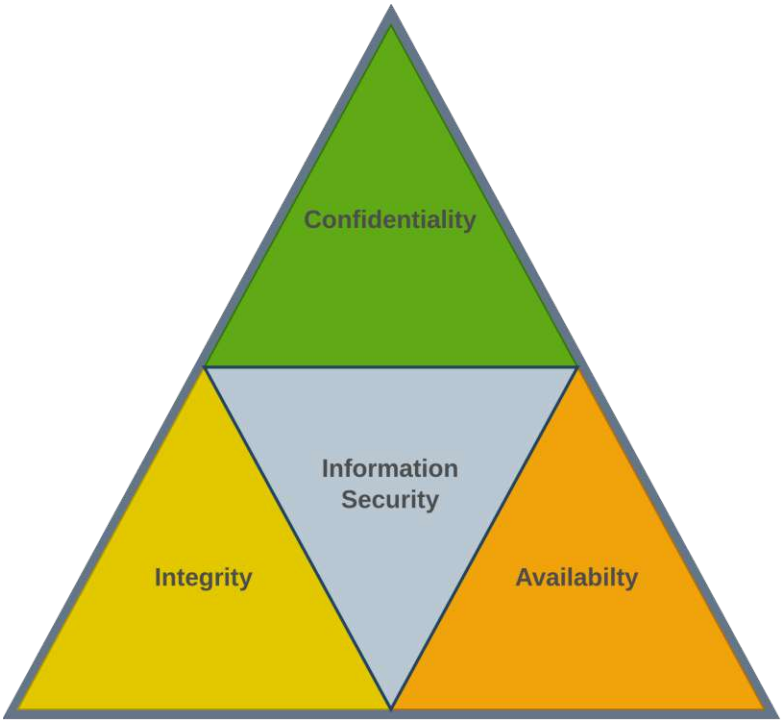

Figure 3.1: A graphic showing the CIA Triad.

The security of a cryptographic system should not be based on secret implementation details but the confidentiality of the key. In the case of cryptography, data (a cryptographic key) protects other data that contains sensitive content to restrict the access to a group of people, which knows the passphrase to decrypt the data.

3.1.2 Integrity

The integrity of data forbids the unauthorised alteration or removal of data. To ensure the integrity of data, cryptographic systems can be used to ensure this property. Checksums are cryptographic functions [56] that can be used to check if, e.g. something modified a blob of data between two points in time. This checksum can also be signed, creating so-called signatures, to ensure that a specific entity created this checksum. Signatures are, for example, used by Android [207] to distinguish application developers.

3.1.3 Availability

Confidentiality and integrity are two properties of data itself. The property of availability, on the other hand, concerns the accessibility and usability of data. Without the possibility to access data when needed, it would be unnecessary to store it in the first place.

The availability of information can be achieved in different ways. For example, a commonly suggested way to back up data is the "3-2-1 backup strategy" [110]. This backup strategy proposes to store data 3 times, on 2 different mediums (e.g. SSD, HDDs, or CDs), and that one backup is in a different location (e.g. in a different data centre). The idea behind this strategy is to prevent a complete loss of data due to a technical failure and external factors, such as the destruction of one storage site.

3.2 Extensions of the CIA Triad

The CIA triad contains only the most fundamental properties for information security. Over the years, various authors extended the CIA triad to address additional problems. The suggested properties mostly build directly upon the CIA triad, providing a finer granularity. The authors [157, 164] argue that since the inception of the CIA triad in 1977 [162] new issues appeared and definitions changed, requiring an adaption of the original CIA triad.

For example, Donn B. Parker created the "Parkerian Hexard" [142] that introduces 3 additional properties

1. **Authenticity.** Reid and Gilbert [157] define "authenticity" as an extension to the CIA triad that "deals with identifying the source of a document or the actual website that is being visited".
2. **Possession or Control.** Ensure that data cannot be obtained by entities that are not supposed to have access to the data.
3. **Utility.** Ensure that data keeps being useful. For example, using a hash function instead of some sort of encryption function to encrypt data would violate the requirement.

Samonas and Coss [164] suggested an additional 8 properties.

1. **Authenticity.** See the definition of "Authenticity" for the Parkerian hexard.
2. **Non-repudiation or Accountability.** This property ensures that it is, e.g., reliably possible to identify who/what modified a file.
3. **Correctness in specification.** Errors in the behaviour of a program should not be due to mistakes made during the specification phase of a project.
4. **Responsibility.** Identify and establish structures of responsibility that, e.g., handle data breaches.
5. **Integrity of people.** As the name of this property states, it is necessary to ensure the integrity of the people involved in a project and remove potentially malicious actors.

6. **Trust.** Mayer, Davis and Schoorman [120] define "trust" as "the willingness of a party to be vulnerable to the actions of another party based on the expectation that the other will perform a particular action important to the trustor, irrespective of the ability to monitor or control the other party".
7. **Ethicality.** Samonas and Coss [164] define ethicality in this context as "the adherence to commonly accepted principles and value".
8. **Identity management.** Ensure the availability of a proper identity management infrastructure.

3.3 Software Security

The definition of software security is rather broad. McGraw describes software security as "the idea of engineering software so that it continues to function correctly under malicious attack" [122]. Only a few pieces of software are proven to be completely bug-free, i.e. not containing any issues that a malicious attacker can use to exploit the respective software. For example, Klein et al. [91] have formally proven the absence of bugs in the seL4 micro-kernel.

Engineering software that is hardened against malicious attacks is a tedious task. There are programming mistakes that can be addressed by, e.g. picking a different programming language, using verification tools, or using only a certain set of functions. On the other hand, there are also more complex issues that are difficult to find. For instance, in 2022 certain Linux kernel version were vulnerable to the *Dirty Pipe* [128] vulnerability, which allowed an attacker to write to any arbitrary file. The cause of this issue has been a commit to add some new functionality to the Linux kernel that inadvertently caused some side effects, which were found accidentally.

The following sections present a few common issues and how they can be addressed.

3.3.1 (Improper) Memory Handling

A major cause of exploitable issues is related to the (improper) memory handling in system programming languages, such as C and C++. One of the first computer worms [140], the "Morris worm", used a buffer overflow in a system service to deploy itself to a system. The service did not validate the amount of read data, eventually overwriting the stack of a program with malicious instructions that were subsequently executed.

There are various further memory vulnerabilities [35], such as heap overflows, integer overflows, and NULL pointer dereferences. Over the years, developers improved compilers [24] to locate these issues and either warn the developers or sanitise them on their own. In addition, static analysis tools, such as SonarQube [105] can also locate such issues and warn developers about them.

Besides improved tooling, there is also the possibility of using memory-safe programming languages. Programming languages, such as Kotlin and Python, use an intermediate

```
1       ...
2       location /stuff/ {
3           alias /var/www/htmx/;
4       }
5       ...
```

```
1       ...
2       location /stuff {
3           alias /var/www/htmx/;
4       }
5       ...
```

Listing 1: An invulnerable nginx configuration (left) and one vulnerable to path traversal (right).

layer that does the memory handling and ensures that no improper memory handling happens.

On the other hand, Rust and Go can skip this intermediate layer and run directly on hardware like C and C++. The programming languages' respective compilers ensure that the source code does not contain issues regarding memory handling.

3.3.2 Improper Configuration

Another common issue is the improper configuration of an application. For example, web servers commonly require a configuration that defines the directory that contains the data that should be made available, the cryptographic algorithms for secure connections, etc. An issue in the server's configuration could allow an attacker to traverse the directories on a system or intercept/manipulate data if, e.g., if the configuration uses an insecure cryptographic algorithm to secure the network traffic.

In 2018 Tsai [201] presented how to break the parsing of file paths in various applications. One of the targeted applications was the nginx web server. In the case of the nginx web server, the absence of a single trailing slash makes the system vulnerable. The configuration on the right side of Listing 1 shows a vulnerable configuration. Without the slash after `stuff`, the file path parser appends everything as-is, allowing an attacker to break out of the web server's root directory.

A bad configuration of cryptographic algorithms is also a potential vulnerability. In 2017, Stevens et. al [182] showed a hash-collison using the SHA-1, where a file was forged to have the same hash as another, entirely different file. Before researcher broke SHA-1, the hashing algorithm MD5 and the AES mode ECB have already been broken [221]. Using these algorithms allows attackers to subvert the cryptography that is used to protect the handled data.

Sanitising improper configuration is difficult to completely automate. Configurations are commonly customised for a specific setup. Thus, it is challenging to create generic patterns that can be checked in every configuration. Testing for the usage of insecure cryptographic algorithms and missing slash characters can be easily done by, e.g. using regular expressions. Anything more complex than checking specific strings is difficult to accomplish using a generic approach. Thus, thorough peer code reviews and (pen-)testing of a setup, are suitable approaches for locating domain-specific issues in a configuration.

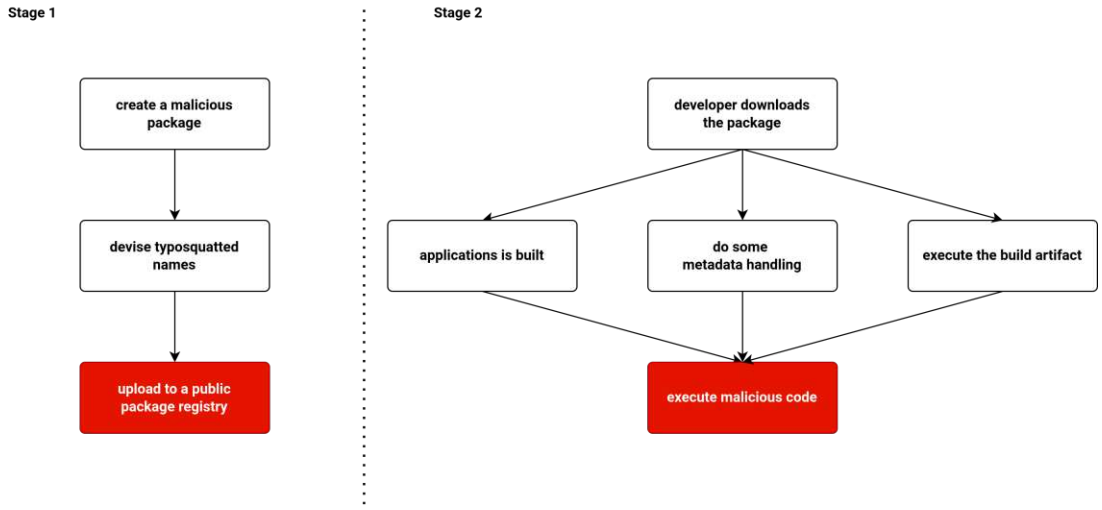

Figure 3.2: A simplified example of a supply chain attack.

3.3.3 Erroneous Dependency Management

Nowadays, most applications are not self-contained but rather rely on dependencies that provide functionality such as JSON parsing, image rendering, or cryptographic algorithms. Dependency management can be done in various ways [184]. Programming languages, such as Go, Rust, and Swift include dependency managers as part of their toolchains. On the other hand, programming languages, such as C, C++, and Java rely on external dependency managers. An issue could be that project maintainers do not properly update dependency lists. Thus, dependencies of projects may contain bugs or security vulnerabilities that attackers can exploit. While some package managers, such as `npm` [232] or `cargo-audit` [80], check for reported (security) issues in the dependency list, most package managers do not include any further checks.

In addition, there have been multiple attempts to actively attack the dependency management mechanisms of different programming languages recently. For example, attackers have uploaded malicious Python packages [204] and JavaScript packages [169, 167] to the respective package repositories. These malicious packages can include code that can create backdoors to grant an attacker access to a system or exfiltrate data, such as passwords that may be used for further attacks.

A common attack to get these malicious packages included is typo squatting [204]. Typo squatting is an attack, where an attacker names malicious packages similar to genuine ones. Thus, when a developer makes a typo, the dependency manager downloads the malicious package. Examples are replacing an “l” (L) with a “1” (one) or an uppercase “i”, appending or removing a trailing “s” (to make the name singular or plural), or replacing letters with one in the surrounding of the key on the keyboard (e.g. replacing a “r” with a “t” or an “e”).

Figure 3.2 depicts a simplified approach for a supply chain attack. In the first stage, an attacker needs to prepare the malicious package. For example, the malicious code can be obfuscated to make it harder to detect, using, e.g. encrypted code that is decrypted at runtime. In addition, using different keys and encryption algorithms allows evading signature-based malware detection. Eventually, an attacker searches for typo-squats and uploads the malicious package to a public package registry under the typo-squatted names.

The second stage already takes place on the target's system. In the first step, the package manager downloads the malicious package. Now, there are three ways the malicious code can be executed.

For example, malicious code can be executed during the build of Rust applications. Rust has a feature called "build scripts" [54] that is used to do additional preparation steps so that a Rust project successfully builds. These tasks include, e.g. the compilation of required libraries or generating Rust code based on SQL files. This build script could also be used directly to execute malicious code.

Another way to execute the malware is during metadata aggregation. At least since 2014 [198], the maintainers of **pip** have been publicly made aware that during the download of a Python package, arbitrary code can be executed. The code execution is necessary in the packaging process to obtain metadata [198]. There have been solutions proposed [198] that were not implemented yet.

Finally, the malicious code contained in the package could simply be built into an artefact. Once a user executes the artefact, the application executes the malicious code.

To counter this kind of attack, there are technical and organisational measures. The countermeasures include, e.g. the verification of a package's signatures and reviewing packages, storing them in a custom package registry, and using only the custom registry to build and develop artefacts.

CHAPTER 4

Fundamentals of Rootkits

The initial idea of a rootkit dates back to a theory presented by Ken Thompson, a main developer of UNIX, in a speech for his Turing Award in 1983 [196]. He theorised that it would be feasible to deposit a master password inside the binary of UNIX's `login` application to create a simple backdoor in a UNIX system. In 1990, Lane Davis and Steven Dake developed the first known rootkit for Sun OS [29].

As rootkits are commonly used as entry points for further attacks [132], attackers added various functionalities to, e.g. gain access apart from inserting a master password. For example, the rootkit *KBeast* [219] exploited the `telnet` application to allow remote access to a system. Common mechanisms include functionality to hide the malicious code from detection systems, add network capabilities to communicate with a control server, and potentially load further malware.

Alongside the improvement of offensive and defensive capabilities, rootkits adapted to work outside the user- and kernelspace of a system. For example, deploying rootkits in places such as the UEFI allows for a better disguise, as the UEFI is a less supervised part of a system. The better disguise comes with a major trade-off:

The further away in the privilege hierarchy a rootkit is positioned from the userspace, the more functionality needs to be self-implemented.

The following sections present various types of rootkits and give a small example for each type.

4.1 Userspace Rootkits

A userspace rootkit [74, 88] is considered the easiest type of rootkit to deploy. Unlike other rootkits, there is no need for elevated user privileges to deploy this type of rootkit, as much as a double click on the rootkit in the system's file explorer is enough to start

and run it. At the same time, this kind of rootkit has the least number of privileges compared to the other types of rootkits. To obtain additional privileges, the rootkit either needs to exploit some security issues or trick the user into granting more privileges by, e.g. appearing as a legitimate application.

Due to the missing privileges, userspace rootkits cannot cloak themselves as well as other types of rootkits. Thus, they need to employ other techniques [41, 74]. For example, a common strategy is to replace known programs, hide executed processes by using unsuspicious names, and hook applications.

Userspace rootkits, such as the one theorised by Thompson [196] can take various forms. In a Linux system, there are commonly several executables that handle sensitive data. For example, a well-known program is `sudo` [213] that requests a user's password. The program can elevate the privileges of other programs, executing them as another user or the `root` user, which the application defaults to. An attacker could replace `sudo`, or simply redirect the call to another executable that could, e.g. intercept and store the entered password. One way to accomplish this is to create an alias in the shell's configuration file. Appending the following line to configuration file

```
alias sudo='<path to evil sudo>'
```

results in a redirection of all `sudo` invocations to its evil counterpart.

Another, more generic approach of a userspace rootkit is shown in Figure 4.1. The rootkit works by replacing (parts of) an application's required shared library with a malicious one. Applications commonly link certain libraries, such as the `libc`, which carry out basic tasks, such as opening or reading a file. Replacing some function, such as `read`, `write`, or `open` could allow an attacker to easily intercept or modify data written or read. The left side shows a *normal* setup. If an application requires a shared library, the system will look for it and link the proper library during the application's startup.

On the right side, an attacker added a malicious library to the system. Instead of the system's library, the system will link the malicious library during the application's startup.

Linux, Windows, and macOS all support letting users overwrite the search order of linked libraries [37, 223, 152]. A path specified using `LD_LIBRARY_PATH` and `LD_PRELOAD` on Linux, `PATH` on Windows, or `DYLD_LIBRARY_PATH` and `DYLD_INSERT_LIBRARIES` on macOS has a higher priority when a system is searching for libraries compared to, e.g. the default ones provided by the system. An attacker can easily persist the changes in the respective operating system. For example, adding the line `LD_LIBRARY_PATH="<directory path to malicious library>:$LD_LIBRARY_PATH"` to `.zshrc` will automatically add the path to the malicious library to the search index in a zsh-session.

This mechanism also has some legitimate applications. For example, Liu, Olivier & Ravindran [109] use this mechanism to replace the default memory allocator found in the `libc` with a more efficient and secure one.

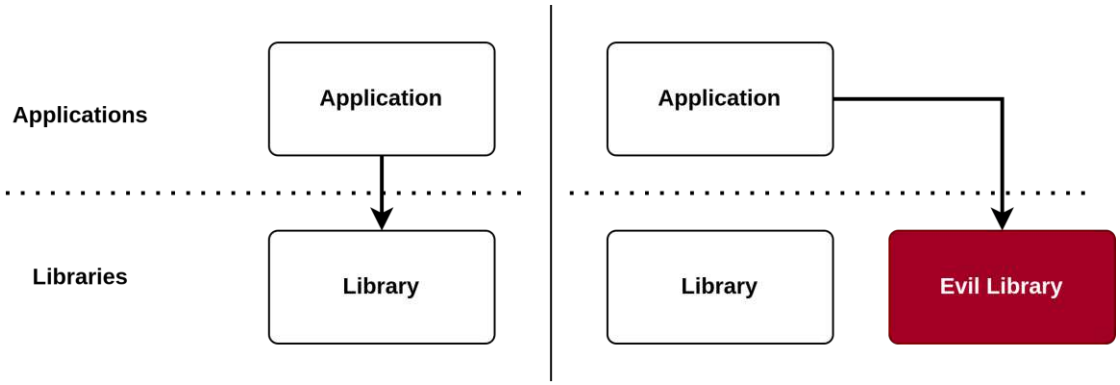

Figure 4.1: An example of a userspace rootkit.

4.2 Kernel Rootkits

Kernel rootkits are deployed in the kernelspace [42, 150] and have access to most resources of a computer system. For example, they can hook system calls and filter their return values for relevant information.

Kernel rootkits offer a big versatility as they have access to the kernel API and can be adapted to fit almost any use case. Its functionality can easily be extended by either deploying further kernel modules or recompiling the rootkit with an updated code base. For example, Linux kernel rootkits can be deployed by either programming and loading a Linux Kernel Module (LKM) or exploiting a security flaw, loading malicious code in the kernelspace and executing it. While the former technique might not be applicable on hardened Linux systems [229] that do not allow kernel modules to be loaded, the latter technique is harder to execute, as a fitting exploit needs to be found.

A well-known example of a kernel rootkit is part of the Stuxnet malware [102], which destroyed Iranian uranium enrichment infrastructure. The rootkit had several parts; one was deployed in the kernelspace and another part was deployed in the userspace.

Figure 4.2 depicts a kernel rootkit. In this example, a userspace process tries to obtain data from a virtual filesystem [161], such as `/dev/`, `/sys/` and `/proc`. A unique property of virtual filesystems is that they are generated on demand. When data of a process is inquired by reading from `/proc/<process pid>`, the Linux kernel calls multiple functions to obtain the data. In an untampered setup, shown on the left in Figure 4.2, the Linux kernel calls the normal functions. An attacker could modify one of these functions, as shown on the right side, to return tampered data. The tampered data could, e.g. change the number of used resources or even hide processes completely.

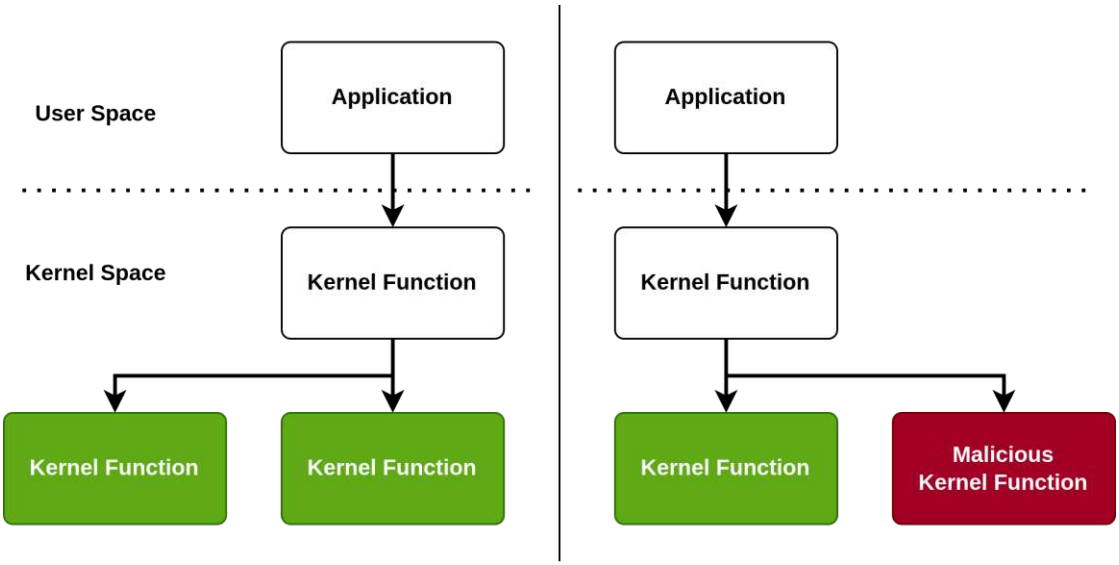

Figure 4.2: An example of a kernel rootkit.

4.3 Hardware-Assisted Rootkits

Hardware-assisted rootkits access the layer below the kernel [180, 43, 75]. Unlike kernel rootkits, hardware-assisted rootkits access a system's underlying hardware on its own, instead of using interfaces provided by the kernel. The drawback of this type is the increased maintenance needed to support different hardware and their unique traits, such as different microcontrollers or software versions. On the other hand, they are less likely to be detected, due to being in a component of the system that is hardly checked for malware.

An oversimplified setup of Spisak's approach [180] is illustrated in Figure 4.3. The rootkit uses a system's Performance Monitor Unit (PMU) to intercept important events. Developers can register monitoring interrupts for the PMU that trigger at specific events (shown on the left). Once the interrupt for a specific event is triggered, handler code, which is defined beforehand, is executed, and e.g. data can be intercepted (shown on the right) and transferred to the rootkit.

4.4 Hardware Virtualisation Rootkits

Hardware virtualisation rootkits [155, 163, 90, 132] also work a layer deeper than kernel rootkits. They use the virtualisation techniques of processors, such as Intel-VT or AMD-V, to host the operating system in a virtual machine. Following this procedure, it places itself between the generated VM and the hardware [47].

As shown in Figure 4.4, a bare-metal hypervisor, such as KVM, is placed between a

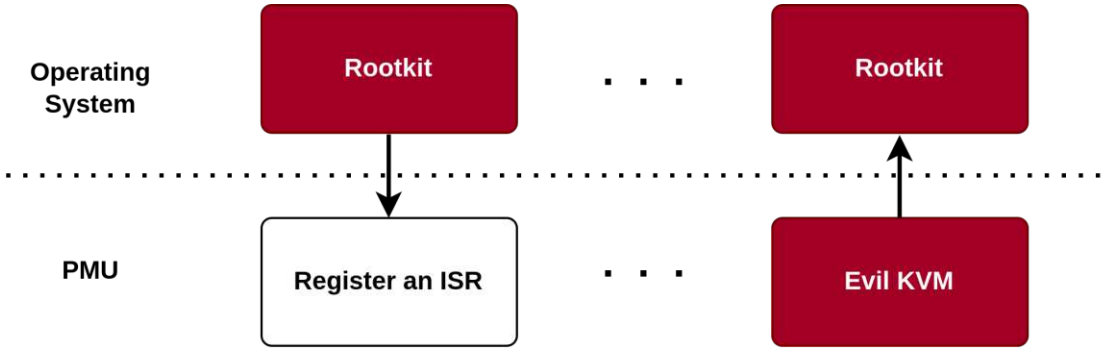

Figure 4.3: An example of a kernel rootkit.

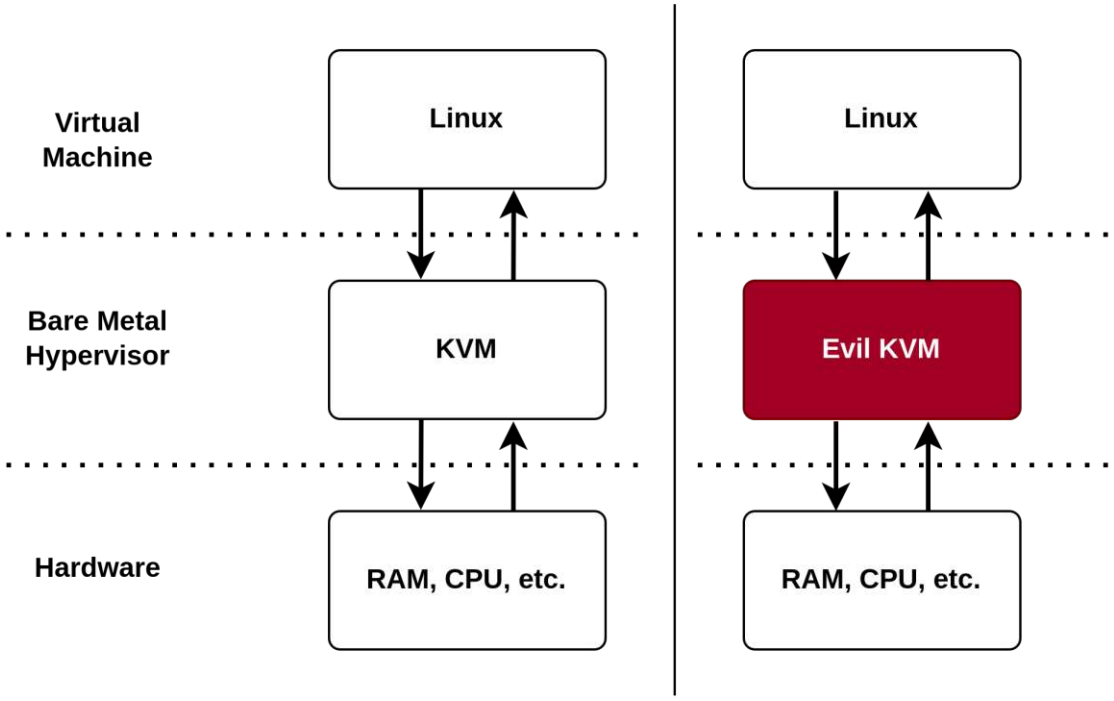

Figure 4.4: An example of a hardware virtualisation rootkit.

virtual machine and a system's hardware (shown on the left). Because of this position in the system, the hypervisor can access all of a virtual machine's hardware resources. A hardware virtualisation rootkit can make use of this property, and e.g. manipulate data written to storage, read a virtual machine's memory, or starve the virtual machine by denying it resources, leading to a denial of service.

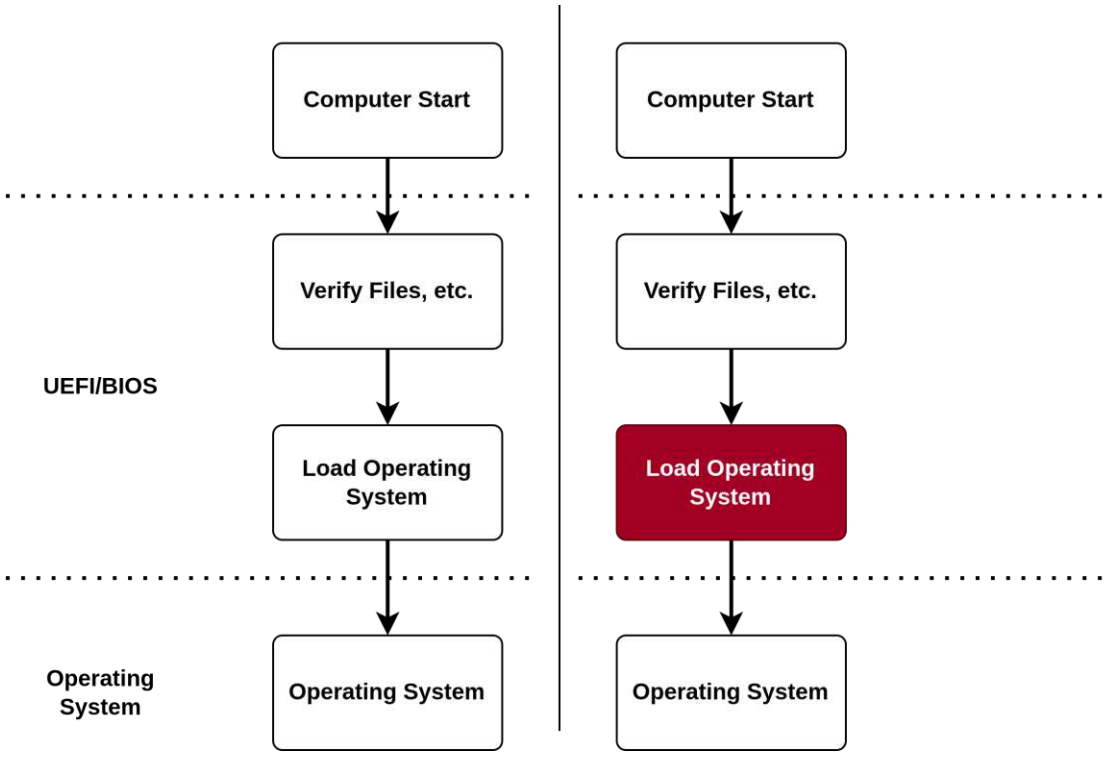

Figure 4.5: An example of a UEFI rootkit.

4.5 BIOS/UEFI Rootkits

A BIOS/UEFI rootkit nests itself in the part of the system that is responsible for booting the actual operating system [94]. The rootkit can infect the boot chain and alter the boot process of the operating systems. Thus, everything associated with the boot process of a system is under its rule, thus it can set, *e.g.*, boot options. This type of rootkit has two major drawbacks due to being UEFI/BIOS bound:

Firstly, it needs plenty of modifications for each respective target, as manufacturers tend to customise the BIOS/UEFI for each series of devices differently.

Secondly, according to Linus Torvalds the BIOS should "just load the OS and get the hell out of there" [199]. The boot system is often discarded after successfully booting the operating system, rendering the rootkit potentially useless after the boot process.

In Figure 4.5 simplified versions of a normal boot process and a manipulated one are shown. In both cases, plugging in the power cable or any other power source, pressing the power button, etc. starts off the boot process [55].

A part of the UEFI boot process, if enabled, is *Secure Boot* [72] process, which makes various checks before continuing the boot process or aborting it, if a check failed. During

the secure boot process, e.g. the signature of certain files is verified using public key cryptography [72]. A major certificate authority in this area is Microsoft. For example, Ubuntu provides UEFI certificates that are in turn signed by Microsoft. For operating systems without a UEFI certificate signed by a certificate authority, it is necessary to disable secure boot when installing a Linux distribution.

After the verification succeeded, the boot process is continued and everything is put in place to boot the operating system. In the given example, the rootkit manipulates the boot process at this stage. For example, Linux supports various options that can be configured at boot time [206]. These parameters allow, among other things, to

- enable the support of USB devices and USB mass storage,
- activate the serial port,
- or disable the enforcement of Linux kernel module signature checks.

Changing these parameters allows an attacker to use an additional attack surface by, e.g., being able to deploy malware using USB keys or downloading data to external USB drives.

CHAPTER 5

ARM Processor Architecture

The initial design of the Arm processor architecture dates to the 1980s [205]. In the 1980s, the British government had a plan to create affordable computers to put in classrooms, as computers were a rather expensive luxury item at the time. The company Acorn Computers Ltd won the bid to produce the BBC Micro for the British government. A few years after the release of the BBC Micro's initial processor architecture `Arm1` the company released the `Arm2` architecture.

In the early 1990s, Acorn Computers Ltd., Apple Computer, and VLSI Technology founded the successor of Acorn Computers Ltd: *Advanced RISC Machines Ltd* (known as `arm` nowadays). After a cooperation with Texas Instruments, `arm` changed its business model, which is still in use today. From the initial business model of designing and producing silicon chips, `arm` moved on to create the processor designs and licence them to different companies, such as NXP, Nordic Semiconductors, and STMicroelectronics.

5.1 Difference Between the RISC and CISC Architectures

The Arm processor architecture is an Instruction Set Architecture (ISA) that belongs to the Reduced Instruction Set Computer (RISC) processor design philosophy.

The design of RISC-based architectures dates to the 1980s [144], with the initial ideas and concepts dating back up to two decades earlier.

In the 1980s, the Complex Instruction Set Computer (CISC) was the predominant processor design philosophy at the time and continues to be to the present day in computers and servers [27]. Over the years, it accumulated different design flaws, such as the ever-growing complexity of the processors produced. The objective of the RISC design philosophy was to address the flaws of the CISC design philosophy by, e.g., relying on a minimal set of instructions and limiting the execution time of an instruction to one

CPU cycle. These two design philosophies differ fundamentally across various points. The most significant differences will be presented in the following paragraphs.

Instruction Size. RISC-based processors all share the property of a constant instruction size that is independent of the used instruction. For example, the standard Arm assembly instruction, called A32 [12] for the 32-bit platform and A64 [185] for the 64-bit platform, has a fixed instruction size of 4 bytes. An exception to this rule is the Arm Thumb instruction set. The Thumb ISA has an instruction size of 16-bit and represents a subset of Arm assembly.

On the other hand, CISC-based processors such as AMD's x86-64 instruction set [4] have instructions of variable length. The minimal instruction size of an AMD x86-64-based processor is 1 byte, and the longest instruction contains up to 15 bytes. Instructions exceeding these sizes trigger an invalid-opcode interrupt that renders the respective instruction invalid.

Processor Microcode. The instructions of Arm-based processors may consist of multiple macro-operations [14]. The macro-operations are split immediately after the decoding stage in the execution pipeline into the smaller micro-operations. An unmodifiable integrated circuit, handles the decoding process, and e.g., in the case of the Arm-Cortex-A78 [14], disassembles a macro-operation into two micro-operations [116].

In contrast, CISC-based processors, such as the ones of AMD and Intel, usually rely on microcode [116]. A coprocessor executes the microcode and transpiles complex instructions into simpler instructions that are similar to micro-operations of the RISC design philosophy. In contrast to the unmodifiable hardware units of Arm processors that decode the macro-operations, the microcode used by companies, such as AMD or Intel, in their CISC-based processors is modifiable. Microcode can be updated like pretty much every other software, and manufacturers regularly issue updates for bugs, and exploits, such as Meltdown [107], or some versions of Spectre [92].

CPU Cycle Length. The CPU cycle is the process of fetching the next instruction and executing it [130].

CISC-based instructions have no specific limitation on the number of CPU cycles used in an instruction's execution.

The idea behind the simplistic instructions of the RISC design philosophy is to minimise the CPU cycles required for the processing of an instruction. In reality, RISC-based processors cannot meet the one CPU cycle per instruction requirement. Using pipelines allows the execution of multiple instructions at once, resulting in the average required CPU cycles of an instruction being close to 1.

Memory Handling. RISC and CISC use different strategies regarding memory management. Commonly, the instructions of CISC-based processors implicitly include the

```
; A64 instruction to load a value from memory
; ldr <register> <memory address (stored in a register)>
ldr x0, [x1]

; A64 instruction to store a value in memory
; str <register> <memory address (stored in a register)>
str x0, [x1]
```

Listing 2: Example of loading and storing values in Arm assembly.

operations that load and store values in the memory [93], which are part of a computation. Eventually, CISC-based processors flush the registers when needed, to prepare for the next set of instructions. The implicit memory accesses are not free and add additional computation time.

Due to RISC's strategy of making instructions as computationally small as possible, there are no instructions that implicitly load and store data from the memory. For example, Arm's 64-bit instruction set [185] includes two base instructions to handle the transfer value transfer from memory to register: `ldr` and *vice versa* `str`.

An example of the load and store instructions can be seen in listing Listing 2.

Present processors do not strictly follow their respective design philosophy, but rather take some influence from the competing design philosophy as well [129]. For example, a single instruction on Arm-based (RISC) processors may take longer than one CPU cycle, but through different techniques, the average instructions appear to be completed in one CPU cycle. On the other hand, the microcode used in CISC-based processors splits up the instructions into smaller instructions, resembling the instructions used in the RISC design philosophy.

5.1.1 Deciding Between RISC and CISC-based Processors

The initial goal of RISC-based designs was to address the constantly increasing complexity of instructions found in CISC-based processor architectures and the resulting complexity of the resulting processor. The upcoming processor design philosophy led to a competition between the RISC and the CISC architecture in the 1980s [27], which continues until today. This war was mostly fought about two factors:

- the complexity of a processor
- the size of the chip area

Nowadays, two different factors are more decisive when choosing a system's processor [27]:

- the performance of a processor

- the energy efficiency

Each of these factors will be addressed in the following paragraphs.

The Complexity of Processors. To execute a program, it needs to be compiled into basic instructions, also known as machine code, which can be used directly by the underlying hardware, such as a system's processor. For example, programming languages such as C [84], C++ [32], or Golang [60] are able to run directly on the hardware after the compilation of the source code. On the other hand, applications written in programming languages, such as C# [76], Java [134, 183], or Kotlin [1], need to pass through their respective virtual machines before being executed. These virtual machines act as a middle layer between the underlying soft- and hardware, so developers can write (mostly) platform-independent code. Typically, the VM s are programmed in C/C++. For example, the JVM (Java, Kotlin) and the Python interpreter CPython are mostly written in C and C++. Unlike the platform-independent code, the virtual machines may need a specific implementation that translates the platform-independent code into platform-dependent instructions. Assembly programming languages take it a step further and can be considered the textual representation of a platform's machine code. Due to the close relationship between an architecture's machine code and the respective assembly language, each new architecture may introduce a new variant of Assembly language.

The major difference between RISC and CISC is the complexity of instructions found in the instruction sets of each respective processor design philosophy. The idea behind the RISC design philosophy is to provide small and simple instructions. They can be executed in one CPU cycle, also called fetch-execute cycle [130]. In such a cycle, the CPU fetches the instruction and executes it immediately afterwards.

CISC-based architectures, on the other hand, contain more complex instructions that sometimes require more than one CPU cycle to finish. The complexity of instructions in CISC-based architectures originates from the idea of providing instructions that are like functions found in higher-level languages. Thus, the compiler needs to do less work, shifting the work to the processor's microcode and subsequently to its manufacturer. Multiple instructions only exist on CISC-based architectures and are sometimes unique to their respective architecture. Listing 3 shows a simple example of a complex instruction and its respective reduced counterpart. The multiplication operation appears to be rather simple but on the hardware level, it is quite complex. Additionally, it is more complex if the multiplicand or multiplier is unknown beforehand, as the compiler might not be able to apply some compile-time optimisations.

In the case of the x86 architecture, two instructions suffice to do a simple multiplication. Firstly, CPU loads the multiplicand into the x86-register `eax` via the `mov` operation. Then, the value in `eax` is multiplied with the value provided as a parameter to the `mul` operation. During this step, the processor saves the resulting product in the register `eax`.

A multiplication operation made on Arm-based system requires more instructions and additional parameters compared to a multiplication made on x86-based processors. First,

```
; Multiply Instruction

; x86-CISC (32-bit)
; Syntax
; mul <value>
mov eax, 0x5
mul 0x6

; Arm-RISC (32-bit)
; Syntax
; mul <destination> <source of the multiplicand> <source of the
↪  multiplier> which is an alias for
; madd <destination> <source of the multiplicand> <source of
↪  the multiplier> WZR (special register that always returns
↪  0)
mov r0, 0x5
mov r1, 0x6
mul r0, r0, r1
; alternatively
; madd r2, r0, r1, WZR
```

Listing 3: Example of multiplication in x86 (32-bit) and Arm (32-bit) assembly.

```
poly <argument> <degree> <table address>
```

Listing 4: Example of the instruction `poly` found in the VAX architecture.

the processor needs to load the values `0x5` and `0x6` into different registers. After these two instructions, the multiplication instructions are invoked. This instruction has 3 arguments: the register for the result of the multiplication, the register for the multiplicand, and the register of the multiplier.

The given multiplication example shown in listing Listing 3 is rather small and only differs in a single instruction. Commonly, the same function implemented in different ISAs will differ in more than a single instruction.

A more complex example is the `poly` instruction found in the AVX architecture (see Patt et al. [143]), showcased in listing Listing 4. This instruction allows the evaluation of a polynomial given an argument, its degree, and a pointer to a coefficient table. The `poly` instruction was used to calculate values such as sine or cosine. Unlike instructions such as `mul`, the `poly` instruction is a function unique to the VAX architecture and has no counterpart in any RISC-based architecture.

The problem arising from incorporating more, and more complex instructions is the

increasing production complexity of the silicon chips. A CISC-based architecture that incorporates instructions such as `poly` is quite complex. The complexity was, and still is, an important cost factor, and it is difficult to justify a rather big price bump for the normal user due to features that might be relevant to only a small group of users.

The Chip Area Size. Alongside the issue of the complexity of a chip's integrated circuit, there was another predominant concern regarding chip construction: the chip's size.

Currently, state-of-the-art processors, such as AMD's Zen 4 architecture [186] uses transistors as small as 4 nm, or Ampere's Altra [151], an Arm-based 64-bit processor used in servers, uses transistors as small as 7 nm. In the coming years, it is planned to further decrease the transistor's size. In comparison to the AMD Zen 4 series and the Ampere Altra, the IBM ROMP [175], a RISC-based 32-bit chip that was released in the 1980s, has a transistor size of 2 µm. The transistor size has almost decreased by a factor of up to 500, which allowed for the integration of more and more transistors in the same size area, resulting in more powerful processors. Due to the rather large size of transistors in the 1980s, chip manufacturers had to focus more on a feature-trade-off compared to chip manufacturers nowadays. Thus, RISC-based process architectures had a slight advantage over their CISC-based counterparts in chip size, as their number of integrated instructions was smaller. Therefore, fewer transistors had to be placed in the chip die, easing the design process.

The Processor Performance. At present, the performance of a processor is, besides energy efficiency, the deciding factor in choosing the processor of a computer system. Compared to the 1980s, the common computing platform nowadays is mobile, like laptops and smartphones, rather than powerful desktop workstations. Thus, the processors need to be as performant as possible, while maintaining a rather low power consumption to provide a long runtime on battery. It is not feasible to say that either the RISC or CISC is the more performant design philosophy. In reality, CISC-based processors still have a slight upper hand performance-wise [151] due to the investments committed by several companies, such as AMD or Intel. In the previous two decades, AMD and Intel have dominated the computer markets with their x86(-64)-based processors that are based on the CISC design philosophy. Due to the demand, the processors of AMD and Intel got more performant every couple of years. Different companies, such as Ampere and Amazon [151], strive to produce Arm-based processors that match the most potent processors of AMD and Intel. In 2021, Poenaru et. al [151] benchmarked various systems using synthetic benchmarks. They showed that Fujitsu's Arm-based server processor came close to the performance of x86-based ones. Simakov et al. [174] repeated a performance and efficiency analysis in 2023. Alike Poenaru et al. they concluded that Arm-based servers are a viable alternative regarding performance and efficiency to x86-based servers.

The Processor Efficiency. Like the problem of pinpointing the more performant design philosophy, it is also not feasible to identify the more efficient design philosophy

solely based on the respective design philosophy's fundamentals. As AMD and Intel rose to power in the computer system market with their CISC-based processors, RISC-based processors conquered the market of embedded devices, which commonly require a low-energy consumption and focus less on high performance. Microcontroller architectures, such as AVR (produced by Atmel, which was later acquired by Microchip Technology), PIC (produced by Microchip Technology), MIPS (produced by MIPS Technologies), and Arm (produced by Arm), are all processor architectures that are based on the RISC design philosophy. Microcontrollers are a specialised variety of processors that are different in one specific property:

The package containing the microcontrollers always includes additional peripherals, such as memory, to use during execution, and read-only memory, to store the application's code. Processors on the other hand only contain the processing unit and must interface their peripherals, such as memory, via their I/O lanes.

The AVR and PIC series [133] mainly stuck to being used as low-power platforms. Both chip series were, and still are, commonly produced as 8-bit or 16-bit Microcontroller Units (MCUs). There are also 32-bit variants of both chipsets, but they were less common due to the limited demand. The designs of MCUs such as Arm and MIPS, on the other hand, also eventually found applications in processors but initially kept being used in systems with lower performance requirements, such as network devices (routers, switches, hubs, etc.), IoT devices, and phones.

5.1.2 Current State of RISC and CISC-based Processors

For the last couple of decades, mostly CISC-based processors from AMD and Intel [151] dominated the market of personal computers and servers. An exception to this were, for example, the PowerPC processors, a RISC-based processor architecture that was used in Apple's computers until they moved on to processors built by Intel. Eventually, Apple moved to a RISC-based processor architecture again by using the Arm-design for their silicon [228].

On the other hand, RISC-based architectures dominated the market for embedded systems, which typically focuses more on the power efficiency of the processor. With the rise of smartphones, this separation slightly dissolved. Initially, smartphones mostly relied on Arm-based processors. Eventually, Intel also started to produce processors that were used in smartphones and tablets [95], basically downgrading their desktop processors to the point that they only use a low single-digit wattage number. At the same time, Arm-based processors found use in computers, such as Chromebooks, or servers, using processors of Ampere or Amazon. This required Arm processors to transform, so they better fit their new field of application. New designs increased the performance to match the ones of CISC-based processors in the field, at the cost of their low-power consumption.

In each case, the new processor had to adapt to the dominating platform. At the current time, it is not possible to draw a proper conclusion regarding the future of Arm-based

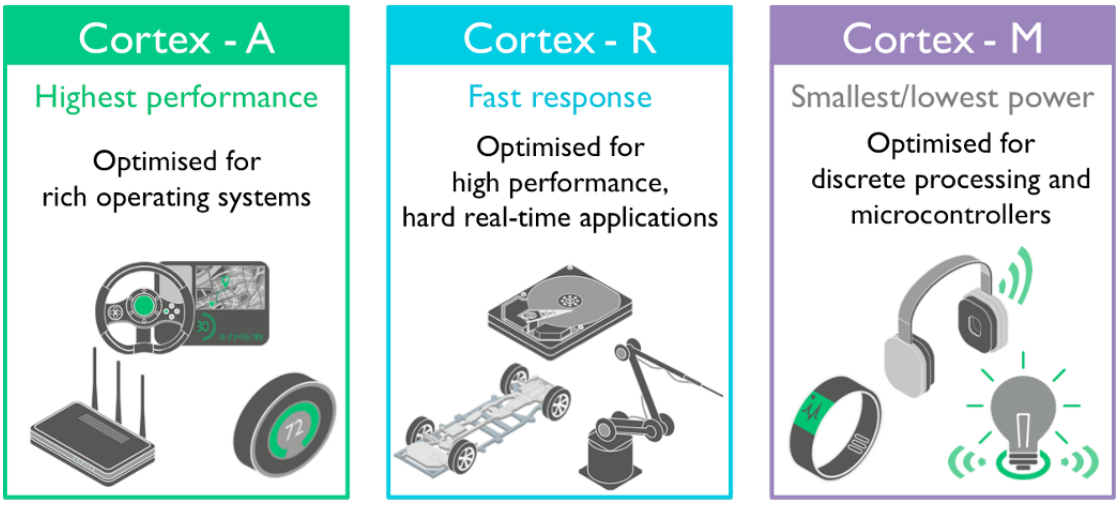

Figure 5.1: Different processor types of Arm depicted in an infographic of Arm [20].

processors in the personal computer and server market. Apple successfully creating their own Arm chips [228] is an indicator that more devices may follow. The processors have a small market share in the personal computer market and are not yet widely used in the server market, as Amazon's and Ampere's processors are rather new and still need to prove themselves. Intel, considered one of the biggest manufacturers of CISC-based processors, on the contrary, gave up their attempt on trying to get a hold in the market of low-power processors. Their attempt only lasted a few years and due to the rather small market share, Intel did not consider it economically viable to further invest.

5.2 Comparison of Different Arm Processor Types

Over the years, Arm designed three different chip variants [202] that can be identified by the naming scheme. Currently, most of Arm's recent processor designs bear the stub "ARM Cortex" and are equipped with one of three suffixes:

- -A(pplication)
- -R(eal Time)
- -M(icrocontroller)

as can be seen in an infographic of Arm Figure 5.1.

Their usage and features are summarised in the following paragraphs.

ARM Cortex-A. The Arm Cortex-A series is intended for application use, such as smartphones and computer systems [13]. The first release of in this category is the Arm Cortex-A8 design. This processor design featured a single 32-bit CPU based on the Armv7 architecture [28]. The Armv7 design also incorporates the Arm TrustZone and optionally the Single Instruction, Multiple Data (SIMD) instructions that allow executing a single instruction on multiple data values. Throughout the years, Arm improved the Cortex-A series and included additional features, such as multiple cores, simultaneous multithreading (currently only found in one processor design of this series: the Cortex-A65AE), performance monitoring units, L3 cache, and 64-bit computing. Processors stemming from this series are commonly more powerful than the ones of the Arm Cortex-R and Cortex-M series. In general, this series of processors has a strong focus on performance.

ARM Cortex-R. The Arm Cortex-R series finds application in devices that need to meet real-time constraints in fields, such as avionics and automation [86]. There are fewer variants and features compared to the Arm Cortex-A and Cortex-M series due to the limited requirements. Performance-wise, the R-series is situated slightly above the Cortex-M series. Additionally, features like the Arm TrustZone security extension are not available on this platform.

ARM Cortex-M. The M-series is the microcontroller series of Arm [18]. Its primary use is in small embedded systems, such as smart keylocks, keyboards, and IoT devices.

Contrary to the R-series, chips based on the M-series may include plenty of additional features. Arm incorporates some of these features, such as complete Arm TrustZone support starting with the Armv8-M architecture [19]. Beforehand, the Cortex-M series only incorporated selected functions of the Arm TrustZone, such as the Cryptocell [115], which offers hardware support for some cryptographic operations. Over the years, some chip manufacturers added custom functionality to meet their specific use case. For example, some MCUs produced by Nordic Semiconductors [115] include different wireless technologies, such as Bluetooth, Bluetooth Low Energy, NFC (tag-emulation), and LTE, due to their popularity in the IoT field.

In addition to the Arm Cortex series, Arm also designs the Neoverse and SecurCore variants.

ARM Neoverse. Systems that require more performance than the Arm Cortex-A series, which is commonly found in smartphones, use Arm Neoverse series [145]. This includes heavy workloads and makes the design suitable for deployment in servers and high-performance computing. Ampere's Altra processor is using, alongside Amazon's Graviton processor, the Arm Neoverse as the base design. The same applies to Fujitsu's A64FX, a processor used in supercomputers. From a technical perspective, the Arm Neoverse can be regarded as a more performant version of the Arm Cortex-A series. This series includes all features of the Arm Cortex-A series, such as the Arm TrustZone

security extension, support for SIMD , and simultaneous multithreading (ARM Neoverse E1).

ARM SecurCore. The Arm SecurCore series is a small chip used in embedded applications with high-security requirements, such as smartcards [233]. Due to their field of appliance, they are low-powered and feature much less performance than even the Arm Cortex-M series. The Arm SecurCore series includes features, such as memory protection and other anti-tempering mechanisms.

CHAPTER 6

Arm TrustZone and Other Trusted Execution Environments

The Arm TrustZone is a hardware security extension that can be used to implement a TEE together with a secure operating system, such as the OP-TEE [137]. TEEs are used to ensure that sensitive data is handled separately from non-sensitive data to ensure that it cannot be accessed or manipulated by attackers. This is done by creating a secure environment that can only be accessed by pre-selected actors.

Taking the protection ring model, depicted in Figure 6.1, as a reference, the Arm TrustZone cannot be pictured in the standard ring layer model. Pinto & Santos [149] use an extended model to correctly classify the Arm TrustZone. The standard ring

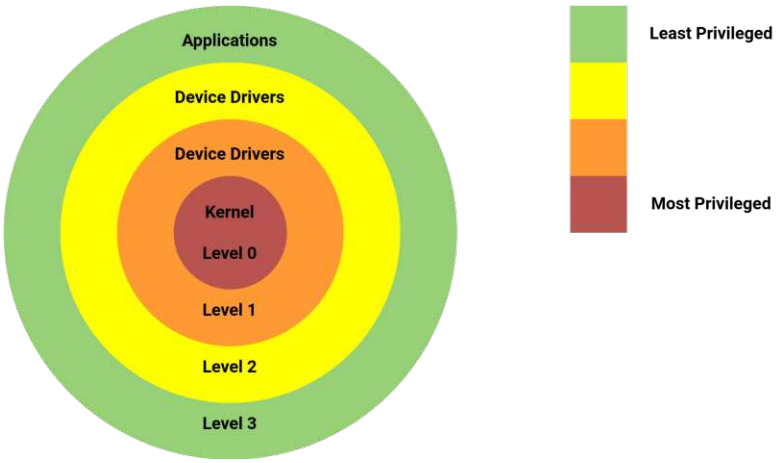

Figure 6.1: A common protection ring model.

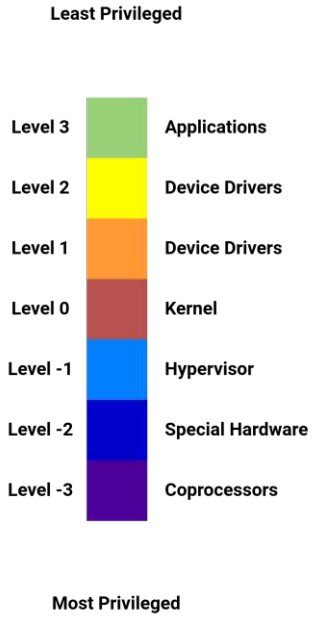

Figure 6.2: A depiction of the extended ring model by Pinto and Santos [149].

layer model shown in Figure 6.1 misses 3 additional layers that are all bearing negative numbers. The extended ring layer model by Pinto and Santos is depicted in Figure 6.2.

Below the kernel, which is found at level 0, there are hypervisors at level -1 that are used in hardware-assisted virtualisation. At level -2 solutions, such as the Intel System Management Mode or the Arm TrustZone reside. Intel's system management handles functionality, such as power management and system hardware control [190]. These basic functions are not accessible by the operating system but by a system's firmware. The last ring level -3 encompasses external security coprocessors such as AMD PSP and Google Titan M.

The following section present the Arm TrustZone, as well as alternatives to it.

6.1 The Arm TrustZone

The Arm TrustZone is an optional hardware-based security extension found in Arm's processors since 2004. The first specification to include the security extension is the ARMv6K ISA [3]. In the first years of its existence, the Arm TrustZone was inaccessible to the public and only in proprietary solutions. The chip manufacturer Xilinx [154] was the first manufacturer to produce publicly available development boards that allowed to access the Arm TrustZone.

Alongside Xilinx, manufacturers, such as NXP, HiSilicon, which is a subsidiary of Huawei, and Broadcom have started to produce development boards that feature CPUs with the

Arm TrustZone accessible to developers [149]. The availability of accessible hardware kickstarted the development of software for the Arm TrustZone, such as the OP-TEE [137], Apache Teaclave TrustZone SDK [52] and the SierraTEE [173]. They allow developers to create applications that operate in the respective TEE. The OP-TEE, and subsequently the Apache Teaclave, follow the Global Platform API [189] specification. This document specifies common APIs to, e.g. encrypt data, access storage, and access the secure world from the normal one.

Besides these 3 projects, manufacturers developed their own solutions, fitting their specific use cases. Two well-known projects are the Android Keystore [8] and Samsung Knox [165]. The Android Keystore's main functionality is to securely store, provide secure access to cryptographic material, and use the cryptographic material for operations such as encryption and decryption. On the other hand, Samsung Knox is a full-fledged security solution that builds upon the Arm TrustZone to secure a device.

Until 2011, only microprocessors contained the Arm TrustZone. The specification for the ARMv8 ISA [19] was the first specification that provides support for the Arm TrustZone on microcontrollers. An Arm whitepaper written by Philip Sparks [179] outlines the importance of this step. According to Sparks, Arm is expecting 1 trillion IoT devices to be in use around the world as of 2035. Most of the devices do not need a lot of computing power. Thus, microcontrollers are the preferred solution for these use cases, as they require less power and are cheaper to produce, among other things. As some of these devices have high-security requirements, it is necessary to bring the Arm TrustZone to microcontrollers.

Most IoT devices only handle insensitive data, such as outdoor temperature, or air humidity. As outlined by Zaidenberg [225], fields such as healthcare or highly automated industries also rely on IoT devices. Commonly, IoT devices in these fields process sensitive data, requiring protection from malicious actors.

6.2 The Arm TrustZone on Different Processor Types

The Arm TrustZone is found in Arm's microcontroller and microprocessor line-up. Although bearing the same name, the implementation in microprocessors fundamentally differs from the implementation found in microcontrollers. In the following sections, these differences will be addressed.

6.2.1 Arm TrustZone in Arm Cortex-A-based Processors

Figure 6.3 shows a schematic of an Arm-A processor's concerning the security extension. In total, there are 4 layers present on an Arm-A-based system, which correspond to the so-called *Exception Levels* (abbreviated as ELs). In Arm's case, the higher the level, the more privileges an exception level has. These levels are directly comparable to the extended ring layer model [149] (see Figure 6.2).

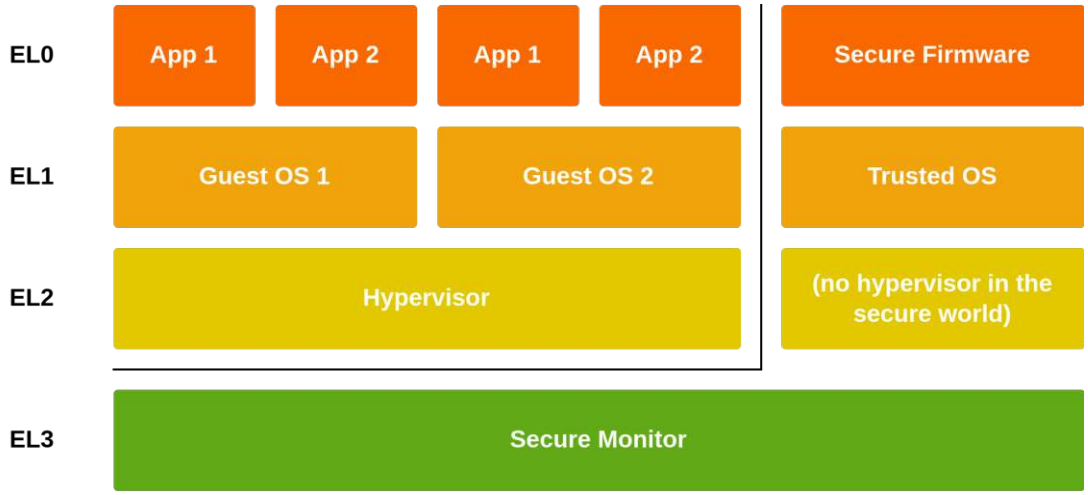

Figure 6.3: Example of an Arm-A processor's security model based on the reference manual for Arm-A processors [17].

The level at the top is EL0, the most unprivileged of all ELs. In this layer, applications, commonly known as user applications, such as browsers and text editors are executed.

EL0 is followed by EL1, which has quite a few more privileges compared to the former EL. This layer would be, e.g., the Linux kernel.

The hypervisor corresponds to EL2 and is found in a system that supports hosting multiple guest operating systems using virtualisation.

The secure monitor is the most privileged component in an Arm-A-based system, thus being assigned EL3.

In the secure world, the setup looks slightly different. The first big difference is the absence of a hypervisor. This results in the inability to host multiple secure operating systems by virtualising them.

In addition, there is also only a single instance of the secure firmware in the secure EL0 layer. As a secure operating system, the OP-TEE resides in the secure EL1.

The secure monitor is responsible for handling the transition to the secure state. The component represents the software component necessary for the Arm TrustZone implementation for microprocessors. Arm offers a reference implementation in the form of the *Arm Trusted Firmware* [15]. The *Arm Trusted Firmware* is explicitly advertised by Arm as a starting point to implement applications for the secure world [15]. Together with the secure monitor, the hardware security extension forms the Arm TrustZone.

To enter the secure world, a process can invoke the `SMC` instruction, which is short for *Secure Monitor Call*. Besides using the `SMC` instruction, a processor can also enter the secure world via an exception, interrupts, and fast interrupts, if configured beforehand.

It is important to note that at any given time, a processor core is either in the secure world or the non-secure world. This design allows multicore systems to process data in the secure world while the remainder of the system can proceed working in the non-secure world. The status of the processor can be determined by the *Non-Secure* flag that can be read from the *Secure Configuration Register*.

Once in the secure world, the processor can access special registers that are only accessible in the secure world [209]. Arm added these registers, called *banked registers*, in the processor design to further strengthen the isolation between the secure and non-secure world. These special registers provide a one-to-one mapping of certain registers found in the non-secure world. Among these banked registers are the stack pointer and the exception link register, which stores the return address of an exception.

Another important set of registers in the context of the Arm TrustZone are the Translation Table Base Registers (TTBRs). They are responsible for storing the addresses of different translation tables to enable virtual memory mapping, a mechanism that will be more thoroughly discussed in Section 7.3. Each world, the secure and the non-secure one, has its set of translation tables. In addition, the TTBRs are available once to access the translation table in the global context (TTBR1) and once to access the translation tables for a smaller scope (TTBR0), such as processes.

In Figure 6.4 Arm's translation tables and its permission system are outlined. The non-secure world's translation table allows code running in the exception levels 0 and 1 to access the memory the peripherals, and the flash memory that is located in the non-secure world. The secure world's translation table works similarly, but in addition, it also grants access to resources in the non-secure world. This mechanism allows, e.g., a trusted application to access memory that is needed for some sort of computation and write out the result so that an application in the non-secure world can access it.

6.2.2 Arm TrustZone in Arm Microcontrollers

As can be seen in Figure 6.5, the Arm TrustZone implementation on Arm's microcontrollers lacks the *Secure Monitor* found in the microprocessor implementation.

The absence of the secure monitor is due to the power efficiency and performance constraints that are imposed on the microcontrollers. Compared to the implementation found in processors, the implementation for microcontrollers allows for a faster context switch and lower power consumption due to the absence of the *Secure Monitor*. Due to the new mechanism, additional instructions had to be introduced as the secure monitor, which acts as a bridge between the normal and secure world, has been removed. Instead of the `SMC` instruction, microcontrollers have three new instructions [18]:

1. **SG (Secure Gateway).** The `SG` instruction indicates an entry point for code from the non-secure world into the secure world.

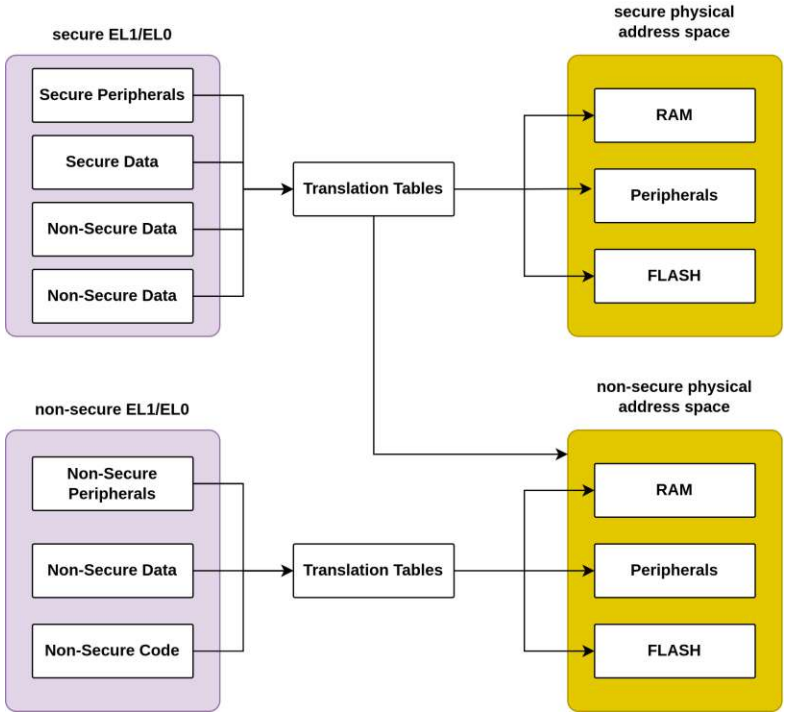

Figure 6.4: Example of an Arm-A processor's memory management based on the reference manual for Arm-A processors [13].

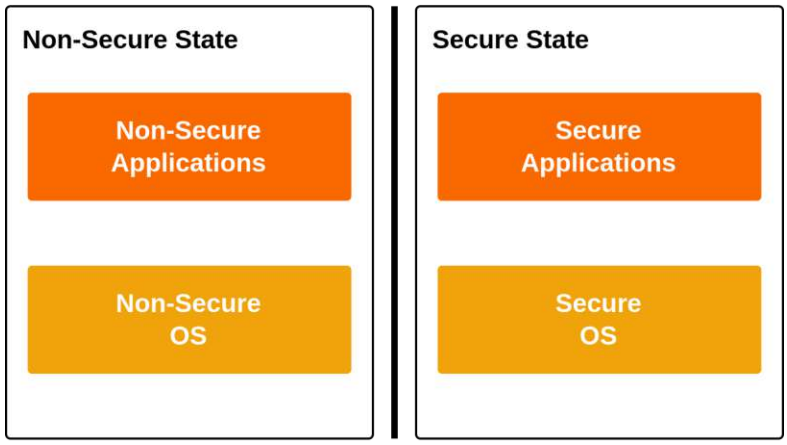

Figure 6.5: Example of the security model in an Arm microcontroller [149].

2. **BXNS (Branch with exchange to Non-secure state).** The `BXNS` instruction allows calling functions found in the non-secure world and to either branch or return to the non-secure side.

3. **BLXNS (Branch with link and exchange to Non-secure state).** Similarly to the `BXNS` instruction, the `BLXNS` instruction also allows calling functions found in the non-secure world. Additionally, the link register stores the address of the next instruction so that, once the called functions returns, the execution continues with the instruction after `BLXNS`.

6.3 Alternatives to the Arm TrustZone

The Arm TrustZone is one of the multiple hardware-based security solutions in use. Besides security solutions, such as Aegis [187], which found no application in productive environments to the best of our knowledge, several other solutions exist that are used in productive environments. The most notable three security extensions will be presented in the following paragraphs.

AMD Secure Encrypted Virtualization (SEV). AMD's Secure Encrypted Virtualization [4] is a security solution that uses memory encryption to prevent information leakage of data residing in the memory. AMD SEV encrypts the memory used by a virtual machine that is operating on a system with a unique key for every guest machine. This is done to prevent the host from accessing the data that is processed by a guest.

A coprocessor integrated into the CPU, the AMD Platform Security Processor (PSP), handles the cryptographic material. Alongside the ability to store the cryptographic material, the PSP also contains a hardware-based cryptographic accelerator to accelerate the encryption and decryption of data.

Google Titan M. Google Titan M [123] is a security coprocessor that provides security functionality, such as secure boot, a feature that is similarly offered by the Arm TrustZone. Further functionalities of the chip include attack detection, tamper detection, and memory protection.

Initially, Google developed and used their custom security chip for servers running in their data centres [220]. The first mobile device to use the Google Titan M is the Google Pixel 3 (XL), which was released in 2018. To the best of our knowledge, the only mobile platform to include Google Titan M coprocessors is Google's own Pixel smartphone series. Every Pixel smartphone, after the Pixel 3 includes the coprocessor.

Intel Software Guard Extensions (SGX). The Intel Software Guard Extensions [40] is an approach that is similar to the one of AMD SEV. Intel SGX creates encrypted memory areas that are decrypted on-the-fly. The sensitive data stored in these secure memory areas is only accessible via special instructions. . A major drawback, compared

to AMD SEV, is that initially only small portions of a system's memory, 128 MB per secure enclave, are useable [59]. As a workaround, the Linux kernel driver for Intel SGX offers paging support to increase the size of enclaves. This workaround is only available on Linux, though, but Intel removes this limit with the 2.0 release of the SGX [79].

Except Intel SGX, every presented security solution is implemented using a coprocessor [149]. Thus, these mechanisms works in one of the lower levels of the protection ring model, compared to Intel SGX, which works on the application level.

The lower the level, the more privileged access a component has to a system, which represents a two-edged sword. Intel SGX is working at level 3 and has a security advantage compared to the security coprocessors in a special regard:

A successful attack on Intel SGX may leak sensitive data but does not compromise the rest of a system, unlike a successful attack on AMD PSP or Google Titan M would.

CHAPTER 7

Fundamentals of the Linux Kernel

Although the terms *Linux* and *Linux Kernel* are synonyms, they are commonly used to describe different things.

Linus Torvalds has initially developed the *Linux Kernel* in 1991 [111], and the project has been under development as an open-source project ever since. Most parts of the source code are C code, and architecture-specific code is implemented in the respective assembly programming language. Starting with version 6.1 the Linux kernel also introduced the first infrastructure for Rust development [61]; initiated by the *Rust for Linux* project. Additionally, helper scripts in scripting languages such as Python, Shell script, and Perl exist to facilitate various tasks, such as preparing and building the project. Over the years, the Linux kernel has amassed a size of about 27 million lines of code (counted with cloc [2] using the Linux release v6.8.7 [9]).

Besides the Linux foundation, other entities are actively developing the Linux kernel. For example, companies such as Google (Android [63]), Qualcomm (Aurora [156]), or Linaro have set up their respective repositories to make their changes publicly available. The respective companies use these repositories to test and integrate code into the Linux kernel for new devices. For example, when Qualcomm releases a new processor, they usually need to add changes to the Linux kernel. Qualcomm's developers push the finished code to Qualcomm's Aurora code repositories. Google then clones Qualcomm's repositories and uses it as its base for new Android Open Source Project releases.

7.1 The Linux Kernel Architecture

One reason the Linux kernel grew to 27 millions lines of code is due to being a monolithic kernel [191]. There are three kinds of kernel architectures:

1. monolithic kernels

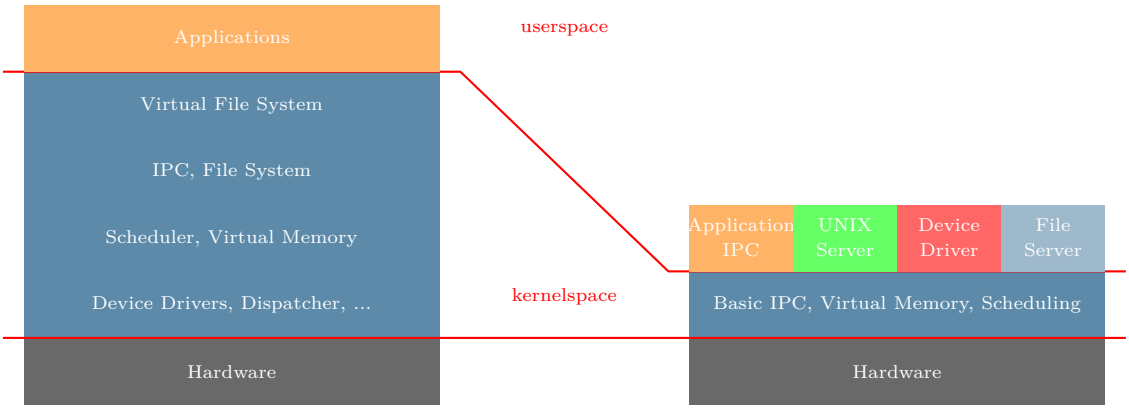

Figure 7.1: An overview of the two kernel types. A monolithic kernel on the left, and a micro-kernel on the right.

2. micro-kernels
3. hybrid kernels (a combination of the former two types)

The first two types are depicted in Figure 7.1. A monolithic kernel, like the Linux kernel, contains, e.g. drivers for the underlying hardware, code for process scheduling and virtual memory. Every time such a kernel supports, e.g. a new feature, a new filesystem, or a new processor, the size of the kernel's source code increases.

The advantage of a monolithic kernel is its performance compared to a micro-kernel. Due to outsourcing functionalities to the userland, a lot more context switches between the user- and kernelspace take place in a micro-kernel. A drawback of a monolithic kernel is that if a single component in the kernel crashes, it will most likely result in a system crash. On the other hand, if a microkernel's device driver crashes, it most likely does not take down the complete system.

On the other hand, a micro-kernel only provides the most fundamental functionalities in the kernel mode (see Figure 7.1). The idea behind this approach is to keep the kernel's source code as tidy and bug-free as possible [191]. The work of Ostrand & Weyuker [141] and Zhang [226] have shown that bigger projects tend to contain more bugs. Furthermore, smaller code bases are easier to formally verify. For example, the micro-kernel *seL4* was formally proven to be secure by Klein et al. [91]. This was possible because the seL4 kernel contains in total about 8300 lines of C and assembly code. The Linux kernel contains around 2500 times as much code, making its formal verification, due to its vast size and complexity, almost impossible.

The Linux kernel contains code to schedule processes, interface and manage devices and filesystems, manage memory, networking, etc. To provide additional extensibility to the

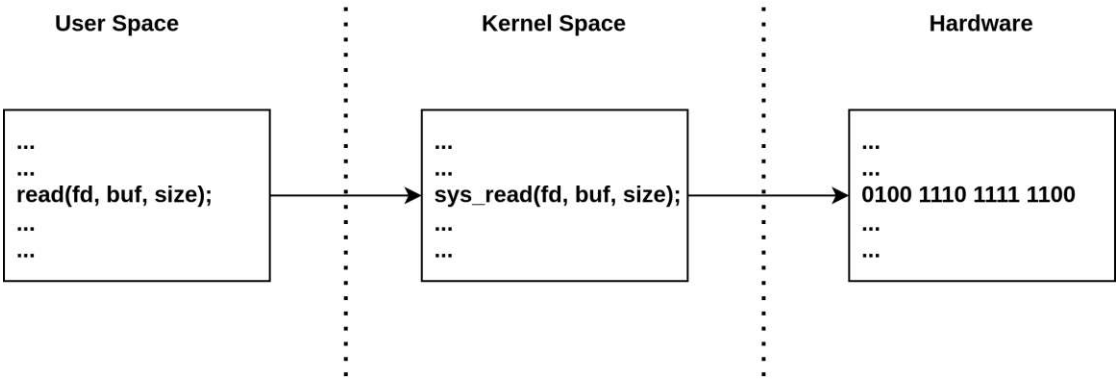

Figure 7.2: Overview of the call hierarchy during the execution of a system call.

Linux kernel, Linux offers support for extensions in the form of modules. By design, the Linux kernel allows third parties to develop parts of the kernel on their own to add, extend, or replace functionality provided by the stock kernel. This property simplifies the process of adding customised drivers for devices or implementing new technologies and algorithms.

7.2 Linux Kernel Internals

The Linux kernel is responsible for tasks such as process scheduling, memory management, and inter-process communications. Applications in the userland can access this functionality using system calls.

In general, a system call works as depicted in Figure 7.2. For example, a program needs to read from a file. An application in the userspace calls the function `read`, triggering a context switch to the kernel, as the system call `sys_read` needs to be called. The kernel does some processing and eventually forwards the call to the hard drive to retrieve the requested data.

In some cases, the Linux kernel can handle system calls completely internally. This happens when a user, e.g., tries to access the `proc` directory. This directory is not a real existing directory, but rather generated by the Linux kernel on request [206]. It is used to share information handled in the kernelspace with the userspace. In this virtual directory, data found in kernel structures, such as `task_struct`, is made available via files that are generated on request.

For example, to retrieve a list of processes, programs such as `ps` [153] need to query the `proc` directory.

In addition, the `/dev` directory may also contain entries that are not backed by an existing hardware device. For example, devices such as `/dev/null`, `/dev/urandom`

and `/dev/binder`, an important device to Android, which will be discussed more thoroughly in Chapter 8, only exist in memory.

7.3 The Linux Kernel's Memory Management

Memory management is one of the most significant tasks of a kernel. One reason being, that memory tends to be a limited resource on every system. Thus, it needs to be properly distributed to ensure the proper functioning of a system. The following sections give a brief introduction of two important concepts of Linux's memory management:

1. virtual memory
2. and user and kernelspace memory separation

7.3.1 Virtual Memory

Over the years, operating systems supported running multiple processes and at the same time processes also got bigger. Thus, a system required more memory to run the processes concurrently. For example, in 1991, the year of first Linux's first release, computer OSes had 4 MB of memory [146]. In comparison, nowadays, the smallest Linux kernels for embedded systems, such as routers, have a size of about 2-4 MB [136]. As for regular computers, according to Steam's hardware survey in March 2024 [181], 99,4% of users have more than 4 GB of memory.

To allow processes to access all the memory they need, even if it does not physically exist, the Linux kernel introduced support for virtual memory [44, 82]. The virtual memory mechanism generates a virtual memory space for each process, where it can address more memory than a system potentially has. Before the use of virtual memory, there was a one-to-one mapping between the memory address a process uses and the physical address of a system's memory.

The introduction of virtual memory mapping results in the necessity of translation tables, as the virtual memory address cannot be mapped one-to-one to physical memory addresses anymore. There is somewhat of an exception, though, as can be seen in Figure 7.3. The memory area where the Linux kernel resides plus some additional memory has an almost one-to-one mapping to the physical memory. To get the physical address of one of these addresses, it is enough to subtract a constant value that slightly varies across architectures and build configurations. All memory addresses outside this region need to be resolved by using the translation tables. Embedded systems are another execption, as they sometimes also have a direct address mapping due to the lack of hardware support. For various reasons, the CPUs in these systems lack a memory management unit that is fundamental for virtual memory management.

The high-level structure of the translation tables is similar across all processor architectures. The actual implementation relies on the available hardware in the respective

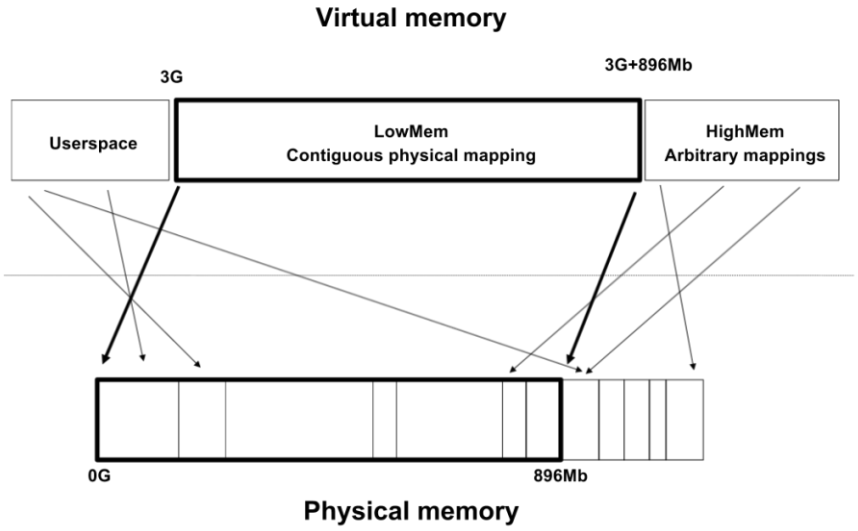

Figure 7.3: Example of memory mapping in a 32-bit system [99].

processor architecture. For example, on Arm-based systems, the TTBRs store the addresses for the translation tables, which are not available on other architectures. Through the number of translation tables, the memory page size can be changed. For example, the 3-level translation shown in Figure 7.4 is used to address memory pages with the size of 4 kilobytes. Armv8-based processors support up to 64 kilobyte pages by adding a 4th translation table.

As shown in Figure 7.4, the first bits of a virtual memory address are necessary for the address translation. Each of these bit groups represents an index for a table, where the value of an index is the pointer to another table. Eventually, after a few steps, a translation table contains the physical address.

7.3.2 User- and Kernelspace Memory

Linux splits the accessible memory into userspace and kernelspace [71]. Not considering interfaces, such as `/dev/kmem` [158], which offered privileged users direct access to the kernel memory, memory in the kernelspace cannot be directly accessed by users.

To let the Linux kernel access userspace memory, multiple functions and macros exist, which are defined in the `uaccess.h` [203] header file. The four most important functions and macros are:

1. `copy_from_user` (function),
2. `copy_to_user` (function),

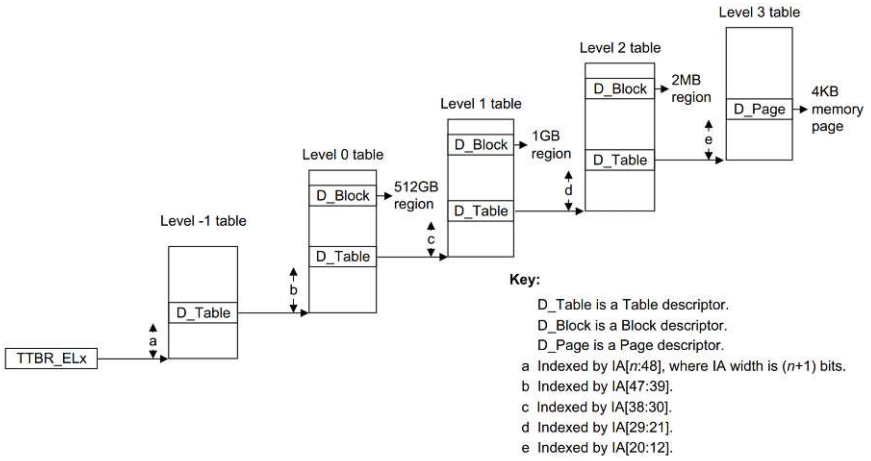

Figure 7.4: Example of a 3 level memory address translation [11].

3. `get_user` (macro),

4. and `put_user` (macro)

Unlike the macros, which should be used to copy only primitive values from/to userspace, the functions can be used to copy an arbitrary amount of data. Accessing userspace memory from the kernel without properly transferring the data into the kernelspace would cause a fault, potentially crashing the system.

7.4 Extensions to the Linux Kernel for Android

The Linux Kernel used in Android differs from the mainline Linux kernel in various points. In comparison to home computers, servers, or embedded systems, Android has different requirements regarding performance and security, requiring extensions to the Linux kernel.

Over the years, Android developers added new functionality to the Linux kernel or improved existing functionality to better support the needs of Android devices.

`ashmem`, short for *Anonymous Shared Memory* [119], is a custom allocator for memory that can be shared between different processes. Between 2011 [112] and 2022, `ashmem` was part of the Linux kernel. In 2022, `memfd` [77] succeeded the `ashmem` memory allocator implementation.

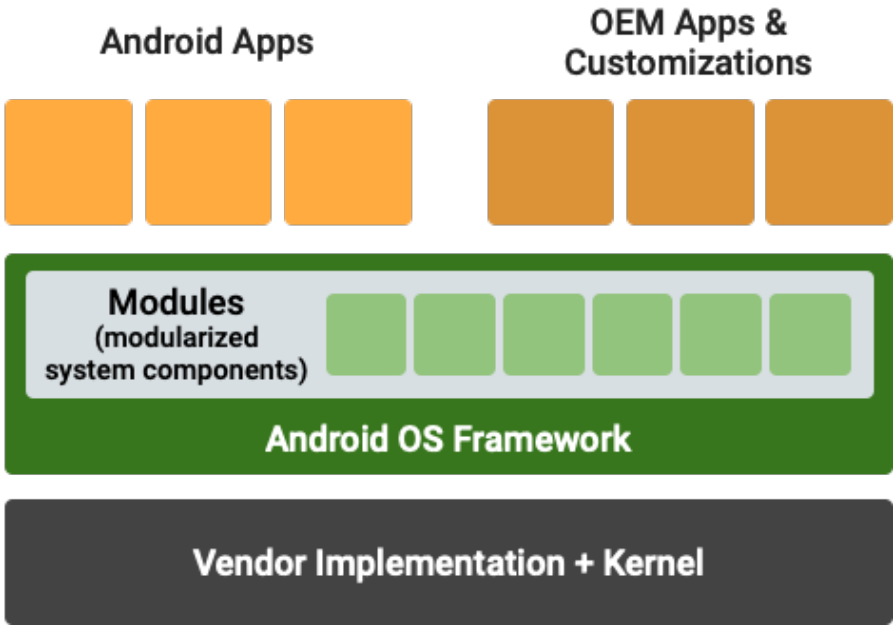

Figure 7.5: The modular concept currently used by Android [7].

Further additions include `wakelocks`, a mechanism to prevent a device from going completely to sleep, which got its last update in 2022 [96] or the Android Binder that will be discussed separately in Chapter 8.

Currently, Android is modularised as shown in Figure 7.5. Vendors provide components, such as vendor apps in the userspace, and make adaptions to the Linux kernel to ensure that their hardware works as intended.

One of Google's goals is to make the additional modifications to the kernel redundant [192]. As of Linux kernel version 5.9 [192], most of the changes required by Android are part of the mainline Linux kernel. Thus, devices supported by the Android Open Source Project can run on a (minimally adapted) mainline Linux kernel.

Ideally, this results in a single kernel image for all devices at one point. Besides *less work* for all involved parties, a single kernel image would result in a faster roll-out of an updated kernel to, e.g. address security issues or add new functionality.

CHAPTER 8

The Android Binder

An important part of every operating system is the Interprocess Communication (IPC). The Linux kernel offers multiple variants of IPC. *Linux Standard Base Core Specification* [53] and the *POSIX specification* [85], which the former specification widely mirrors, specify these mechanisms. Namely, these IPC mechanisms include signals, pipes, message queues, semaphores, and shared memory.

These mechanisms do not suffice for application on Android smartphones [49]. Errors in these low-level inter-process mechanisms provided by Linux, unintended or not, can lead to the subversion of a system by an attacker. Error causes can include the carelessness of developers, unintended side effects, or attacks caused by malicious actors, inadvertently leading to security problems. For example, the vulnerability CVE-2022-0847 [128], which is nicknamed *Dirty Pipe* because of the exploited IPC mechanism, enabled an attacker to take over a Linux system. Due to the vulnerability, unprivileged users can read and write files that should normally be inaccessible to them.

To harden the system, Android uses the *Binder* [65], an inter-process mechanism initially developed as Open Binder [70]. To further harden the IPC, Google rewrites the `libbinder` [64] in Rust. As part of the *Rust for Linux* project, the work on a Rust rewrite of the Binder driver [106] has started. The idea behind a rewrite in Rust is that the programming langauge is less prone to memory vulnerabilities [87], improving the security of the Binder components.

The Binder, also known as the Android Binder, features a client-server architecture that allows a process to request or provide service for another process. In the Binder's client-server architecture, the client and the server are processes in the userland. The component in the kernel handles most of the work involved in the data exchange between two processes. The kernel module contains a lot of the involved program logic. In comparison, the `libbinder` is mostly responsible for transforming data and communicating with the Binder driver.

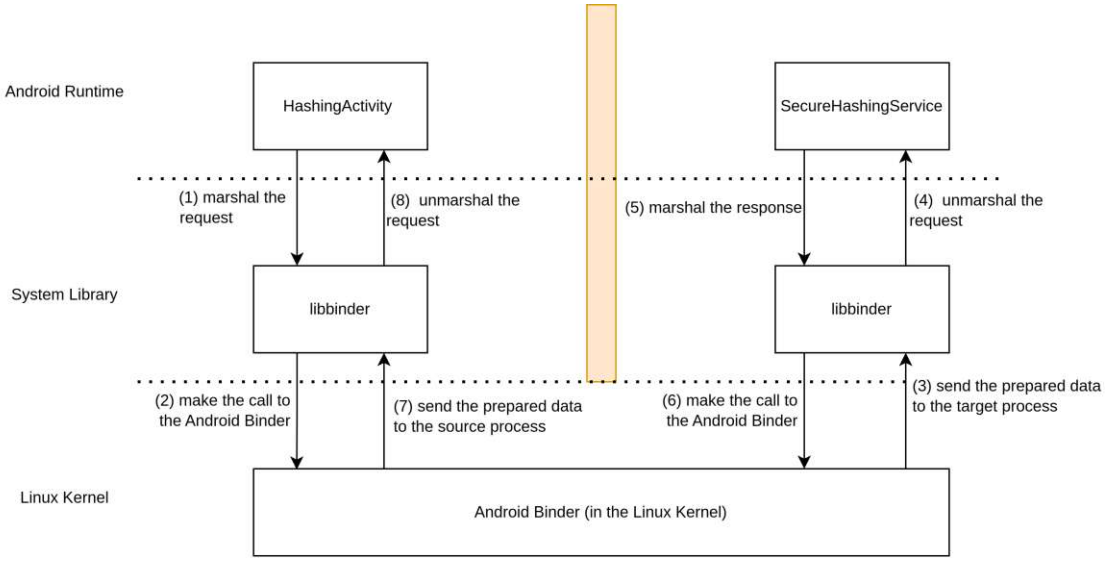

Figure 8.1: An example of the data and control flow of the Android Binder.

Figure 8.1 exemplifies the Android Binder flow. The graphic shows an imaginary `SecureHashingService`. In the example, the `SecureHashingService` is called by the `HashingActivity` that is part of another app. To issue a request to the `SecureHashingService`, the `HashingActivity` needs to send data to the Binder component in the kernel. Before the transmission, the client process marshals the data into so-called `Parcels` (1) that the server will unmarshal later on (4). In both cases, the `libbinder` is responsible for the conversion of the data and also sends the marshalled data via the `ioctl` syscall to the Binder driver (2).

The kernel module performs several checks [26] during the transmission of a Binder packet to ensure that, e.g.

- a process has the privileges to send/request data from another process,
- the correct metadata is present,
- and the server answers with the correct type of data

After the checks succeeded, the kernel module forwards the data to the server process (3). There it gets unmarshalled and processed so that the `SecureHashingService` can use it. Once the service finishes its computations, the `libbinder` marshals the data (5) and sends it back to the kernel (6).

The kernel driver processes the response of the service and prepares to sent it back to the client process (7). In the client, the response needs to be unmarshalled one last time so that the `HashingActivity` can use it.

Android's Binder protocol supports a total of 19 different types of commands. An excerpt of the available commands is shown in Listing 5.

As can be seen in the listing 5, Binder commands can contain data packets attached to them. The first two types of binder commands, `BC_TRANSACTION` and `BC_REPLY`, are part of the control flow that transfers data between two processes. The `binder_transaction_data` that is sent from a client to a server process. A server process processes the data stored in the structure and puts into another `binder_transaction_data` struct as part of a `BC_REPLY` command. The client process receives the reply and can access, e.g. the computation results.

Besides complex structures, single primitive types are also part of data embedded in commands. For example, `binder_uintptr_t`, which is a 32-bit unsigned integer. A process sends the `BC_DEAD_BINDER_DONE` command to acknowledge that an observed process died.

Lastly, there are also commands, such as `BC_REGISTER_LOOPER`, which registers a looper thread, which do not need to carry extra information.

```
enum binder_driver_command_protocol {
  BC_TRANSACTION = _IOW('c', 0, struct
   ↪  binder_transaction_data),
  BC_REPLY = _IOW('c', 1, struct binder_transaction_data),
  /*
   * binder_transaction_data: the sent command.
   */

    ...

  BC_REGISTER_LOOPER = _IO('c', 11),
  /*
   * No parameters.
   * Register a spawned looper thread with the device.
   */

    ...

  BC_DEAD_BINDER_DONE = _IOW('c', 16, binder_uintptr_t),
  /*
   * void *: cookie
   */

  ...
};
```

Listing 5: Excerpt of the Binder commands [26].

CHAPTER 9

Arm TrustZone-based Rootkit

An Arm TrustZone-based rootkit is another type of rootkit. Similarily to user- and kernelspace rootkits, it can rely on the interfaces provided by an underlying (secure) operating system. The proof-of-concept uses the OP-TEE and its interfaces to implement and run the rootkit.

The following sections discuss the basics and design of an Arm TrustZone-based rootkit.

9.1 Basics of an Arm TrustZone-based Rootkit

As discussed in Chapter 7, the Arm TrustZone commonly runs a custom operating system that is as lightweight as possible. We think this is mainly due to preventing too much performance overhead and keeping the number of programming mistakes low, which are the result of a bigger code base. Zhang [226] as well as Ostrand and Weyuker [141] have shown that there is a relation between the lines of code and programming mistakes.

Thus, a secure operating system such as the OP-TEE highly differs from a Rich Execution Environment (REE), such as Linux, regarding the availability of device drivers, memory management, etc. Subsequently, the OP-TEE is not context-aware of the properties of a REE. It does not understand the relation between the memory mappings done by the REE and the context of the bytes written to the memory. Techniques commonly deployed by kernel and user rootkits, such as hooking functions in the REE, are not as easy to implement, as there is no context of kernel functions.

For example, a kernel rootkit commonly acts as pictured in Figure 9.1 to intercept data. On an unmodified system, a simplified execution would proceed as follows:

A function calls the exposed `read` system call which reads data from file descriptors, copies the data into a buffer provided by the caller, and returns the number of bytes copied to the buffer.

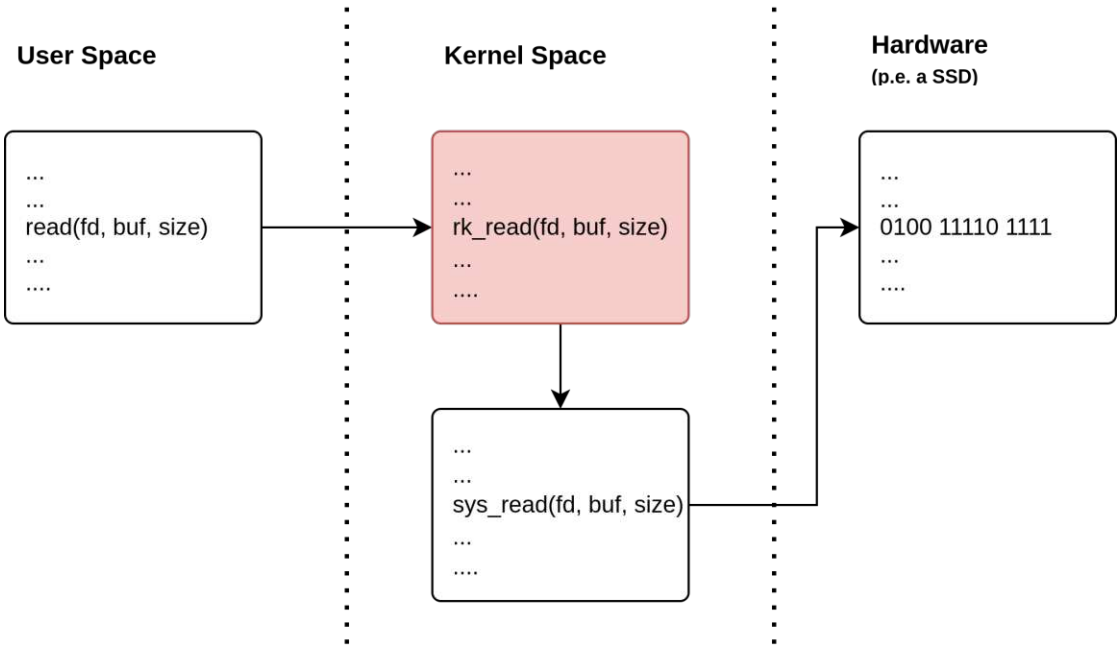

Figure 9.1: Overview of the call hierarchy when an intercepted system call.

This normal execution might get modified by a kernel rootkit by the control flow shown in Figure 9.1. The user calls `sys_read`, but the rootkit replaced the original function with a malicious one, `rk_read`, without the function's caller knowing it. `rk_read` can intercept data before it is copied from/to the user by proxying the call to `sys_read`. This approach allows an attacker to manipulate the data returned to the caller or intercept the data and copy it elsewhere for further processing.

The above-mentioned method would not work the same way if executed from the OP-TEE, due to missing information that is only available to the REE. Commonly, a kernel rootkit redirects a system call by iterating over the memory and searching for the system call table, which contains the address pointing to the `read` system call. Before the Linux kernel version 2.6.18 [168], the system calls `sys_exit` and `sys_close` were exported. Rootkits can use the addresses of these system calls to find the memory area that contains the system call table when iterating over a system's memory.

When working from the OP-TEE, it is not possible to obtain such data without an extensive parsing of the memory and puzzling together the obtained information. Marth et al. [118] have shown that it is possible to identify `task_struct` structs in the memory that are used by the Linux kernel to handle process scheduling. The search algorithm devised by Marth et al. works well for `task_struct` structs due to some unique properties of this struct, among other things. For example, the `task_struct` struct contains multiple pointers to itself on several offsets. This allows to search for certain patterns, which are 8 bytes long (the size of an address in 64-bit systems) and are prefixed

with `0xffff`, which all kernel memory commonly has [117].

On the other hand, the system call table does not provide as many restrictions, making system call hooking a much more difficult task.

As shown by this example, it is necessary to work around the constraints that are imposed when working outside a REE. An algorithm needs to be devised to find artefacts in the raw memory to make the changes to the control flow of the kernel, which will be discussed in the following section.

9.2 Design of an Arm TrustZone-based Rootkit

The design of an Arm TrustZone-based rootkit differs from the design of types of rootkits such as userspace and kernelspace rootkits. Besides the restriction that a trusted application does not have a context of what all the 0s and 1s in the memory are referring to, there are further limitations that make it necessary to adapt mechanisms commonly found in rootkits. In the TrustZone, only a minimum of devices can be accessed as easily as in the normal world, due to the lack of drivers.

9.2.1 Intercept Binder Data

The constant iteration over a system's memory to identify binder data not only causes a big performance overhead but also likely results in a large amount of false positives. To minimise the performance overhead and rate of false positives, the preferable approach is to intercept data once it is processed.

A function that handles Binder data is `binder_ioctl_write_read` [25]. In addition to processing the struct that is used to transfer data via the Binder, this function contains code that is only relevant for optional debugging purposes. The space used by the instructions to output the debug data can be replaced to contain instructions to interact with the rootkit.

As can be seen in listing Listing 7, the function `binder_ioctl_write_read` calls the debug function `binder_debug`. This function is a wrapper for one of the Linux kernel's `printk` functions and does a preliminary check if debug data should be printed. Thus, it can be assumed that replacing this debugging functionality will not affect the proper execution of the Binder.

When calling the function `binder_write_read` struct, shown in listing Listing 6, is passed as part of an argument to this function.

Inside the `binder_write_read` struct, the variable `write_buffer` is defined, which is a pointer to the data destined for another process. The first few bytes of the buffer indicate the type of data transmitted to the Android Binder. For the rootkit, only data containing the enum `BC_TRANSACTION` is relevant, allowing to filter out any data irrelevant to the rootkit component in the secure world.

```
1
2  /** On 64-bit platforms where user code may run in 32-bits the
       driver must
3   *  translate the buffer (and local binder) addresses
       appropriately.
4   */
5
6  struct binder_write_read {
7    binder_size_t    write_size;    /* bytes to write */
8    binder_size_t    write_consumed;  /* bytes consumed by driver
       */
9    binder_uintptr_t  write_buffer;
10   binder_size_t    read_size;    /* bytes to read */
11   binder_size_t    read_consumed;  /* bytes consumed by driver
       */
12   binder_uintptr_t  read_buffer;
13 };
```

Listing 6: `binder_write_read` struct as seen in kernel version 4.14 [26].

During the compilation, the debugging function will be inlined, as can be seen in Listing 8 and Listing 9. The generated assembly contains a check if data should be printed (line 2), the preparation for the function call (line 6-15 and line 6-16), and the instruction to eventually call `printk` (line 18 and line 19). Depending on the compiler and the used optimisations, the instructions responsible for outputting the debug data consist of 11 to 12 instructions, which are 44 to 48 bytes of memory. This space suffices to pass the pointer of a `binder_write_read` struct to another location.

After identifying the needed data and where to redirect the control flow, it is necessary to determine the location in the memory where the patch for the function redirection is placed. To identify the location, we devised a search algorithm that allows to identify the location of the 11 to 12 instructions that will be replaced. Besides the compiler version and the used optimisations, the kernel version also affects the resulting machine code, as offsets and values may differ between different kernel versions. This makes it very difficult to find a search pattern that can be uniquely identified in all available Linux kernels. Instead of an exact byte matching, the algorithm checks for the operations codes of certain instructions that highly likely match the sought instructions.

Listing 8 and Listing 9 show an extract of the disassembled `binder_ioctl_write_read` function. The assembly code shown in Listing 8 was extracted from a stock kernel, compiled by Victor Chong [38] as part of an official pre-built image for the Hikey 960. In Listing 9 the disassembly of a custom kernel, containing additions made to provide the Linux kernel interfaces for the OP-TEE, is shown. Both code extracts eventually call `printk`, but the set-up before the call differs in the number of statements and the

```
1  static int binder_ioctl_write_read(struct file *filp,
2              unsigned int cmd, unsigned long arg,
3              struct binder_thread *thread)
4  {
5  int ret = 0;
6  struct binder_proc *proc = filp->private_data;
7  unsigned int size = _IOC_SIZE(cmd);
8  void __user *ubuf = (void __user *)arg;
9  struct binder_write_read bwr;
10
11 if (size != sizeof(struct binder_write_read)) {
12     ret = -EINVAL;
13     goto out;
14 }
15 if (copy_from_user(&bwr, ubuf, sizeof(bwr))) {
16     ret = -EFAULT;
17     goto out;
18 }
19 binder_debug(BINDER_DEBUG_READ_WRITE,
20              "%d:%d write %lld at %016llx, read %lld at
               ↪  %016llx\n",
21              proc->pid, thread->pid,
22              (u64)bwr.write_size, (u64)bwr.write_buffer,
23              (u64)bwr.read_size, (u64)bwr.read_buffer);
24
25 if (bwr.write_size > 0) {
26     ret = binder_thread_write(proc, thread,
27                     bwr.write_buffer,
28                     bwr.write_size,
29                     &bwr.write_consumed);
30     trace_binder_write_done(ret);
31     if (ret < 0) {
32         bwr.read_consumed = 0;
33         if (copy_to_user(ubuf, &bwr, sizeof(bwr)))
34             ret = -EFAULT;
35         goto out;
36     }
37 }
38 ...
39 ...
40 ...
```

Listing 7: The `binder_ioctl_write_read` function as seen in kernel version 4.14 [25].

```
1  # check if the debugging function should be called
2  cbnz x0,LAB_00a56c3c
3
4  # load all the required arguments for function call into the
5  # respective registers
6  adrp x8,0x162e000
7  ldrb w8,[x8,#0x1f8]=>DAT_0162e1f8 = 07h
8  tbz  w8,#0x6,LAB_00a53800
9  ldr  x8,[sp, #local_100]
10 ldr  w2,[x20, #0x30]
11 ldr  x3,[sp, #local_f0]
12 ldp  x4,x5,[sp,#local_e0]
13 ldr  w1,[x8, #0x40]
14 ldr  x6,[sp, #local_c8]
15 adrp x0,0x1181000
16 add  x0=>DAT_011813e4,x0,#0x3e4 = 01h
17
18 # call printk
19 bl   printk
```

Listing 8: Prelude of the debug function call in an unmodified Linux kernel.

```
1  # check if the debugging function should be called
2  cbnz x0,LAB_00a3d79c
3
4  # load all the required arguments for function call into the
5  # respective registers
6  adrp x8,0x15c9000
7  ldrb w8,[x8,#0xde0]=>DAT_015c9de0 = 07h
8  tbz  w8,#0x6,LAB_00a3a060
9  ldr  w1,[x20, #0x40]
10 ldr  w2,[x27, #0x30]
11 ldp  x4,x5,[sp, #local_e0]
12 ldr  x3,[sp,#local_f0]
13 ldr  x6,[sp, #local_c8]
14 adrp x0,0x114a000
15 add  x0=>DAT_0114a842,x0,#0x842 = 01h
16
17 # call printk
18 bl   printk
```

Listing 9: Prelude of the debug function call in a modified Linux kernel.

```
# prepare the instructions to call vmalloc_exec
mov  w0, #4096

# prepare the address of the vmalloc_exec function
movk x3  0x<byte 1 and 2 of vmalloc_exec address>
movz x3  0x<byte 3 and 4 of vmalloc_exec address>
movz x3  0x<byte 5 and 6 of vmalloc_exec address>
movz x3  0x<byte 7 and 8 of vmalloc_exec address>

# execute the function call
blr x3

# add the pattern to search for
stp x0, x0, [x0]
stp x0, x0, [x0, #16]
stp x0, x0, [x0, #32]
stp x0, x0, [x0, #64]
stp x0, x0, [x0, #256]

# nop sled to prevent execution of invalid instructions
nop
```

Listing 10: Instructions needed to allocate an executable memory page and write a pattern to it.

types of statements used. Besides a difference in the number of instructions used to prepare the execution of `printk`, the order and operation codes slightly differ. Despite the differences, similarities exist that make it possible to derive a search pattern for the used instruction.

9.2.2 Architecture of the Arm TrustZone Rootkit

The Arm TrustZone rootkit consists of 2 components:

- the rootkit running in the secure world
- shellcode placed in the memory range of the Linux kernel

The components both reside in EL1 in their respective worlds, as can be seen in Figure 9.2. Shellcode resides in the normal world's EL1, as part of *Guest OS 1*. On the other hand, the secure world rootkit is part of *Trusted OS* in the secure world.

Both components will be explained thoroughly in the following sections.

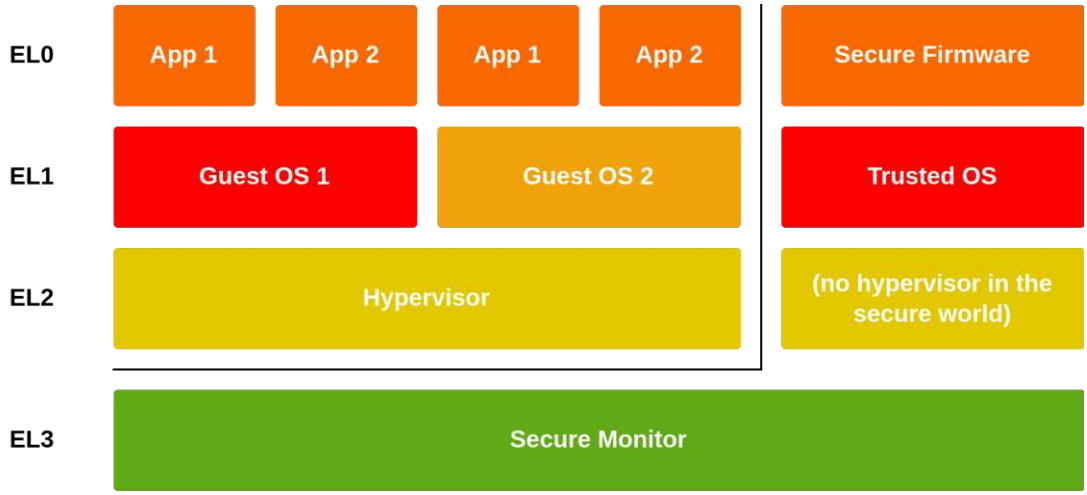

Figure 9.2: The Arm-A processor's security model [17] with subverted components (shown in red).

9.3 Preparing the Interception of Data in the Android Binder

To access applications running in the OP-TEE, it is necessary to have a client running in the normal world. The first few versions of the OP-TEE mainly relied on a userland client that interacted with the OP-TEE via a kernel driver. Starting with Linux 4.12-rc1 [215], the Linux kernel module for the OP-TEE contains the APIs to allow also Linux kernel modules to interface the OP-TEE. As the Linux kernel used by the Hikey 960 did not include this API for an unknown reason, the API was ported from a stock Linux kernel of the same version (4.14).

The first step of the Arm TrustZone rootkit is its invocation, depicted as (1) in Figure 9.3. In this case, a Linux kernel module invokes the rootkit.

Once the LKM makes the initial call, the trusted application takes over and prepares the necessary steps to forward Binder data to itself. In total 5 steps have to be taken to intercept data in the Android Binder:

1. find the memory region that contains the `binder_ioctl_write_read` function
2. allocate a memory page to fit all the instructions necessary to forward data to the TA
3. identify the address of the memory page just allocated
4. prepare the shellcode and copy its instructions to the allocated memory page
5. place the instructions that redirect the control flow to the allocated memory page

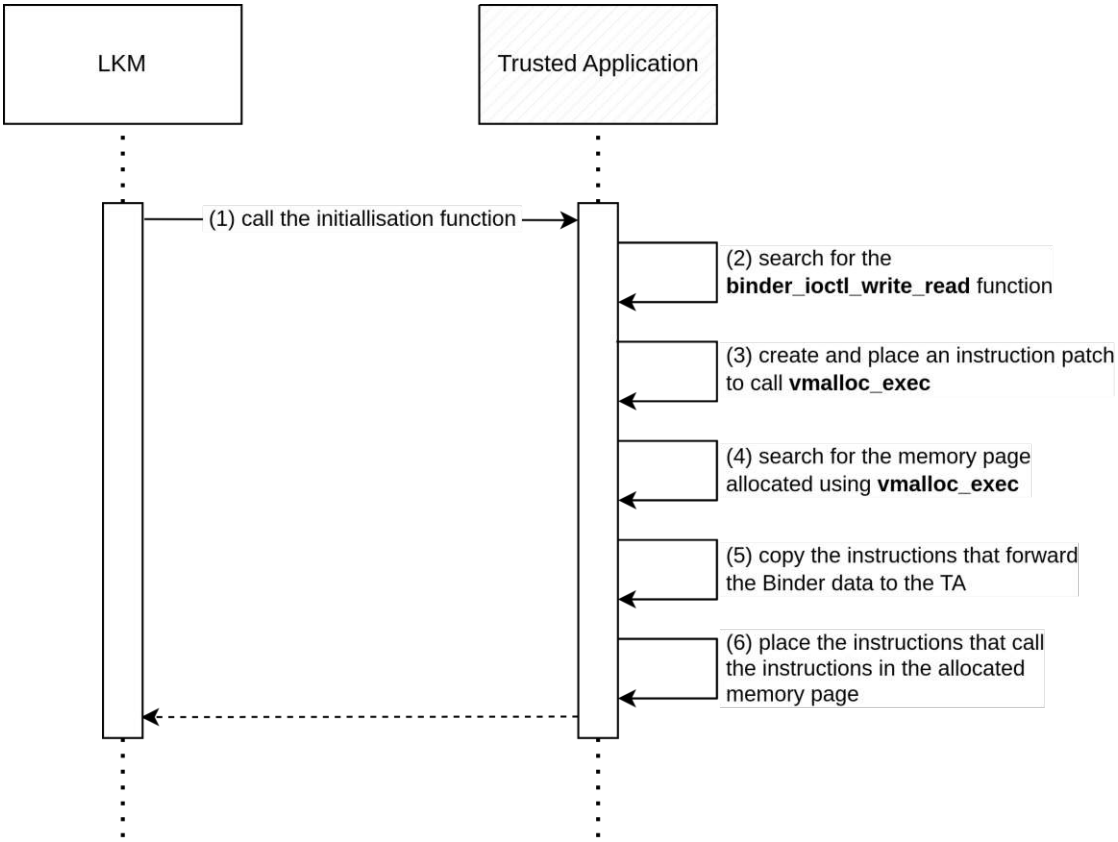

Figure 9.3: The initial call from the LKM to the TA.

The call to the rootkit's `binder_ioctl_write_read` function, where the patch will be applied to. The used algorithm can identify the sought memory region based on a pattern of certain instructions.

Once the rootkit identifies the memory, the instructions below the instruction `cbnz` shown in Listing 9 and Listing 8 are replaced by the instructions shown in Listing 10 (3). The execution of this code snippet causes a call to the function `vmalloc_exec`, which is used by the kernel to obtain an executable memory region, to allocate a single page of memory. The address of the function `vmalloc_exec` is hardcoded, a limitation that will be addressed in more detail in Section 9.5.

After the execution of `vmalloc_exec`, the register `r0` contains the address of the memory page. The execution of the remaining shellcode creates a searchable pattern by writing the value in register `r0` at the indices 0, 1, 2, 3, 4, 5, 8, 9, 32, and 33.

The rootkit repeatedly checks the memory for the appearance of the search pattern in a memory page until it is found (4). The checks need to be repeated, as there is no guarantee that the instructions, which allocate the memory page, are executed right

```
1  # save the binder_write_read struct's address in x0
2  add x0 , sp, <offset>
3
4  # prepare the address of the MITM function
5  movk x3  0x<byte 1 and 2 of the memory page address>
6  movz x3  0x<byte 3 and 4 of the memory page address>
7  movz x3  0x<byte 5 and 6 of the memory page address>
8  movz x3  0x<byte 7 and 8 of the memory page address>
9
10 # execute the function call
11 blr x3
12
13 # nop sled to prevent execution of invalid instructions
14 nop
15 nop
16 nop
17 nop
18 nop
19 nop
```

Listing 11: The instructions needed to forward the control flow to the allocated memory page.

after they have been written to the memory. Once the search process identifies the allocated memory page, the instructions calling `vmalloc_exec` are replaced by `nop` (*No Operation*) instructions. If the existing instructions were not to be replaced, further memory allocations could lead to a crash of the system, due to running out of memory.

The trusted application puts shellcode in the memory page that facilitates copying data from the Android Binder to the TA (5). The shellcode is discussed in detail in the following section (see Section 9.4).

Eventually, the rootkit replaces the `nop` instructions with instructions to execute the shellcode (6) shown in Listing 11.

Eventually, the control flow returns to the LKM, which initially invoked the rootkit, and subsequently exits. At the same time, the rootkit in the Arm TrustZone lies dormant until the shellcode invokes it.

9.4 The Shellcode in the Normal World

The rootkit contains shellcode that is a helper to facilitate moving data from the Binder to the rootkit. In the following sections, special properties and the functionality of the shellcode are explained.

```
...
# store the bytes 0 and 1 of the function address in register 2
mov  x2, #0x1414
...
# store the bytes 2 and 3 of the function address in register 2
movk  x2, #0x1414, lsl #16
...
# store the bytes 4 and 5 of the function address in register 2
movk  x2, #0x1414, lsl #32
# store the bytes 6 and 7 of the function address in register 2
movk  x2, #0x1414, lsl #48
# call the function address stored in register 2
blr  x2
...
```

Listing 12: An excerpt of the generated shellcode instructions.

9.4.1 Properties of the Shellcode

The shellcode is directly embedded in the Arm TrustZone rootkit and deployed by the rootkit to memory in the REE. The shellcode consists of three functions:

1. a 4 instruction function that is required for the OP-TEE setup
2. a function (further on referred to as `setup_ta_call`) that is responsible for setting up the environment needed to make a call to the TA and free allocated resources after operations have finished
3. a function (further on referred to as `make_mem_copy`) that handles the duplication of the Android Binder data. Parts of the data are in the userspace and require special handling to make them accessible. Without using the function `__arch_copy_from_user`, trying to access the memory region would result in a kernel panic.

In our case, the shellcode is based on the code of a Linux kernel module that works and is compiled differently to a normal LKM. The binary lacks stack protection to remove unneeded code and after the compilation the binary is also stripped. In addition, instead of calling functions by their name, functions calls invoke placeholder addresses like `0x1414141414141414`. This forces the compiler to generate the snippet shown in Listing 12.

Using placeholders allows the shellcode to adapt to different systems. During step (5) in the initialisation phase, the rootkit (see Figure 9.3) replaces the placeholders with a

function's real memory address. As in the case of `vmalloc_exec` function, the addresses of the functions are hardcoded. This limitation will be addressed in detail in Section 9.5.

9.4.2 Functionality of the Shellcode

As shown in Figure 9.4, the invocation of the shellcode is triggered by a call to the function `binder_ioctl_write_read` (1). The instructions that have been placed during the initialisation of the rootkit redirect the control flow to the function `setup_ta_call` (2), which is part of the shellcode.

The function `make_mem_copy` (3) repeatedly copies memory from the userspace to the kernelspace. Most data in the struct `binder_write_read` resides in the userspace. Thus, it is necessary to copy it to the kernelspace, as an access would cause an error otherwise. The first 4 copied bytes contain the type of Binder transaction. If those 4 bytes do not match the `0x40406300`, the function returns `NULL`, indicating that the function `setup_ta_call` could not identify relevant data.

Copying the data is necessary, as the sending, as well as the receiving end of the inter-process communication, are located in the userspace. As can be seen in step (3) of Figure 9.4, the data that is transferred via the Android Binder is deep copied to make the data accessible to the code placed in the kernelspace. Theoretically, it is feasible to forward the userspace memory pointers as-is to the trusted application and let it directly handle the memory. This causes a major overhead in complexity, though, as the virtual memory addresses need to be parsed and resolved to physical memory addresses.

The selectd approach is to copy the data from the userspace to the kernelspace and subsequently forwarding the data from the kernelspace to the trusted application. This approach allows reusing functions that are available in the Linux kernel.

Eventually, the shellcode invokes several functions of the OP-TEE driver to prepare the call to the TA (4). After the completion of the preparation steps, the shellcode invokes the rootkit with the data copied from the Binder as parameter (5).

9.5 Limitations of the Presented Approach

The current approach has a major shortcoming: Most Linux kernel symbol addresses are specific to a specific compiler and platform. The address of kernel functions, such as `kmalloc`, `kfree`, and the OP-TEE driver's API, are hardcoded in this proof-of-concept. Thus, the current implementation of the rootkit does not work on differing platforms and kernel versions. The execution of the rootkit causes an incompatible system to crash. The circumvention of this issue requires the ability to dynamically identify the address of kernel symbols. There have been for example the work of Zhang, Meng, & Wang [227], as well as a different variant shown in an online post by Igal [100].

The method of Zhang, Meng, & Wang [227] would theoretically work on the Linux kernel version that is used by the Hikey 960 (Linux 4.14). As the authors have neither provided

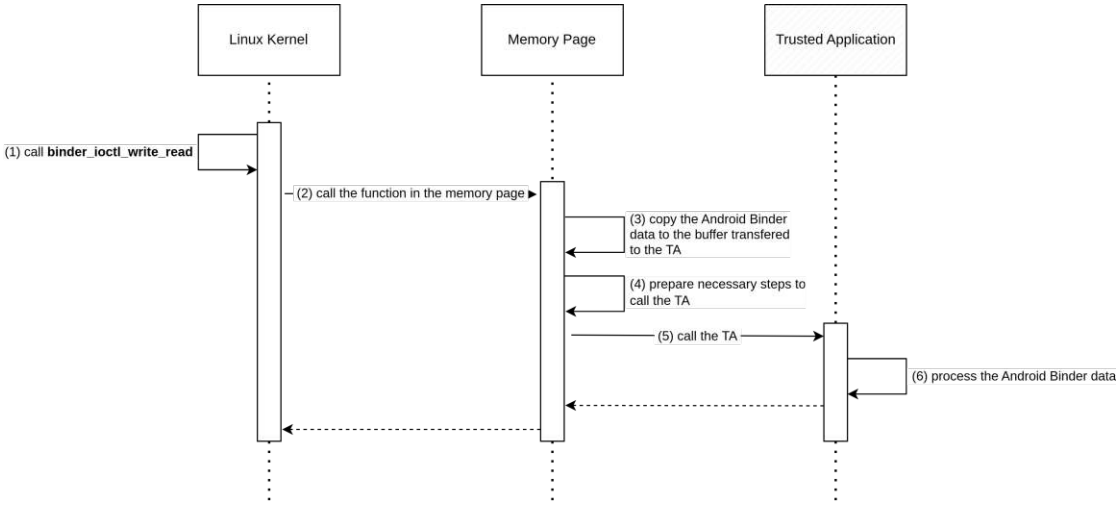

Figure 9.4: A sequence diagram showing how the shellcode invokes the rootkit.

their source code nor provided thorough documentation, it is not usable without extensive research. Attempts to get further information have been unsuccessful; there has been no response to sent emails.

On the other hand, Igal has documented his work on his blog [100] and has published the source code on GitHub [101], allowing to reproduce the result. The method used by Igal is not applicable anymore without major modifications or respectively a complete rewrite. Over the years, several things have changed in the Linux kernel's codebase. For example, smartphones commonly use a 64-bit Linux kernel, compared to the 32-bit kernel Igal used in the blog entry. Furthermore, several kernel structures have been replaced, rewritten, or have been removed entirely from the Linux kernel.

CHAPTER 10

Threats and Limitations of an Arm TrustZone-based Rootkit

In the following chapter, threats and limitations of an Arm TrustZone-based rootkits will be discussed. Furthermore, detection and prevention mechanisms for the rootkit will be shortly addressed, as it directly impacts the threat caused by this kind of rootkits.

10.1 Threats Posed by an Arm TrustZone-based Rootkit

The threats of a rootkit in the Arm TrustZone are similar to the ones in the user and kernelspace. In addition to the general threats posed by rootkits, which are discussed in chapter Chapter 4, additional threats exist due to the placement of the rootkit in the attacked system. Once an attacker can reside in the secure world, it is very unlikely that another component can verify and validate the instructions executed in the system. To the best of our knowledge, there is currently no coprocessor operating on level -3 in the protection ring model, which checks a system's memory for malware.

Due to this property, Arm TrustZone rootkits are harder to detect and harder to remove than other types of rootkits. In addition, Arm TrustZone rootkits can access all of a system's resources, granting it total control of a system.

Each of these threats will be discussed in the following sections.

10.1.1 Detection of Arm TrustZone Rootkits

The first threat mainly stems from the fact that the Arm TrustZone is so deeply integrated into a system. A layer in the protection ring model can directly access all resources by the layer above it, without relying on the interfaces provided by the layer above it. Commonly, the Arm TrustZone is the component with the lowest protection ring model

level. Only Google Pixel phones, such as the Google Pixel 3 [220], contain coprocessors that are positioned a level below the Arm TrustZone. This leads to a fundamental issue that the author Juvenal [212] has identified already in the first to second century:

"Quis custodiet ipsos custodes?",

which translates to

"Who will watch the watchmen?"

There is no intended mechanism to control what the trusted applications in the TEE are doing. Hardware-based intrusion detection mechanisms, such as the ones proposed by Singh et al. [176] and Zhou et al. [231] might be suited for this task. There are several issues, though:

1. Where are the detection mechanisms placed?
2. How are the machine learning algorithms trained?
3. What should be done once a rootkit is found?

The intrusion detection would need to be integrated into the firmware of a system, as the data could be easier forged if the intrusion detection systems would, e.g. run as part of the kernel, 2 levels above the Arm TrustZone.

The second issue is harder to address, as there is a rather small to non-existent sample size of Arm TrustZone rootkits. Considering the work of Marth et. al [118] and this thesis, the sample size is 2. Training an algorithm on such a small size likely leads to many false positives, potentially causing more harm than good.

The last issue is also rather difficult to resolve. While it might be feasible to shut down certain systems that execute malicious software, other systems might be too important to stop. For example, shutting down one of many sensors that measures humidity likely causes fewer issues than powering off a system that is integral for controlling power lines.

As detecting Arm TrustZone-based rootkits is currently hard, it is necessary to take any precautions necessary to prevent a rootkit from finding its way into the system in the first place.

10.1.2 Removing Malicious Software from the System

Compared to normal software, firmware is much harder to update. For example, updating fundamental firmware on smartphones is a rather tedious and complicated task. Without the option to update a device's firmware, a system is prone to attacks, exposing a huge user base to a security risk.

A good example of the problems faulty firmware causes is a jailbreak that affects Apple's iPhones. The jailbreak *checkra1n* [98] uses the *checkm8* [216] vulnerability to give users more privileges on iPhones. The checkm8 exploit uses various vulnerability in the *Device Firmware Upgrade* mode, an integral USB driver that is part of the secure boot chain. Apple is unable to patch the exploit in the firmware, exposing many iPhone users to a security risk.

10.1.3 Unrestricted Access to System Resources

The unrestricted access an Arm TrustZone-based rootkit has to a system's resources is a big threat. As discussed in Section 10.1.1, to the best of our knowledge, there are no components in a system that audit the behaviour of processes in the secure world. Thus, a rootkit in the OP-TEE has uncontrolled access to various components of a system.

Commonly, manufacturers use Arm processors in embedded devices, such as set-top boxes and small, handheld gaming systems (e.g. the Nintendo Switch). In addition to standard components such as memory and storage, embedded devices usually also have additional hardware incorporated, such as card readers. In some cases, the additional hardware allows processes to access to sensitive data, creating an additional attack surface. For example, recently, software like softPOS [195] allowed to use Android smartphones as banking terminals. Accessing the NFC interface would allow intercepting all kinds of banking information and in addition, e.g. manipulate a transaction to increase the paid amount.

The unrestricted access still comes with a big limitation: all the data needs to be parsed by the rootkit on its own to make sense of it. This limitation will be further discussed in the following section.

10.2 Limitations of an Arm TrustZone-based Rootkit

An Arm TrustZone-based rootkit has two major issues: the missing context of gathered data and finding a way to enter a system. Both issues will be discussed in the following sections.

10.2.1 Obtaining and Interpreting Data

The limited set of functionality a trusted operating system commonly provides, makes it difficult to easily aggregate data from sources apart from the memory. Scrapping the memory for data is simple, as a trusted application has direct access to it. Due to the

lack of drivers, trusted applications commonly cannot, e.g., obtain data from a hard drive or wireless interfaces, such as Wi-Fi or Bluetooth. To do so, the OP-TEE would need to incorporate a driver to interact with the hard drive and a driver that can interact with the hard drive's filesystem. Trusted execution environments hardly use other means of exchanging data with a rich execution environment other than pre-defined interfaces. Thus, it was never necessary to add functionality to directly communicate with other components of a system. To access devices, such as hard drives or network devices, it would be necessary to write custom drivers for every piece of interfaced hardware.

Once the data acquistion finishes, another issue awaits. The intercepted data has no meaning to the rootkit by default, as everything is just a bunch of ones and zeros due to the missing context a rich execution environment provides. To be able to distinguish a binary blob from a chunk of memory, which might contain valuable information, it is necessary to give the data a context. Depending on the sought content, more or less sophisticated approaches need to be taken to identify data. For example, private keys in the PEM format are straightforward to look for. They start with the header `-----BEGIN PRIVATE KEY-----` and end with the footer `-----END PRIVATE KEY-----`. On the other hand, encrypted data might just be a binary blob, requiring a lot more effort to reliably identify as there are no headers to identify the binary.

10.2.2 Persistent Storage and Exfiltration of Data

Sometimes the goal of an attacker is not only the manipulation of data, but also a way to either extract or persistently store data for further processing. The OP-TEE offers an API to securely store data by either creating a secure, encrypted partition that is only accessible to the secure world or by relying on the normal world.

The latter variant, called *REE FS Secure Storage*, is shown in Figure 10.1. The secure world encrypts the to-be-stored data and forwards it to the OP-TEE driver in the normal world. Once the data is received in the normal world, the OP-TEE driver stores it in the local file system.

A major issue in this regard is that only NXP's boards support either mechanism [138] in the OP-TEE version 3.13.0, which is used for the proof-of-concept. Subsequently, there is no default way to securely store data in many environments, requiring self-implemented workarounds to securely persist data.

10.2.3 Deploying an Arm TrustZone-based Rootkit

A big limitation of an Arm TrustZone-based Rootkit is the ways it can be deployed. Compared to other malware, there are only a few possibilities to enter a system.

The following paragraphs present 3 attack vectors to deploy malicious code. Each of these attack vectors is a viable way for the rootkit to enter the TEE, but the probability of any of these existing for a particular target system is rather small. This severely limits the threat caused by an Arm TrustZone-based Rootkit.

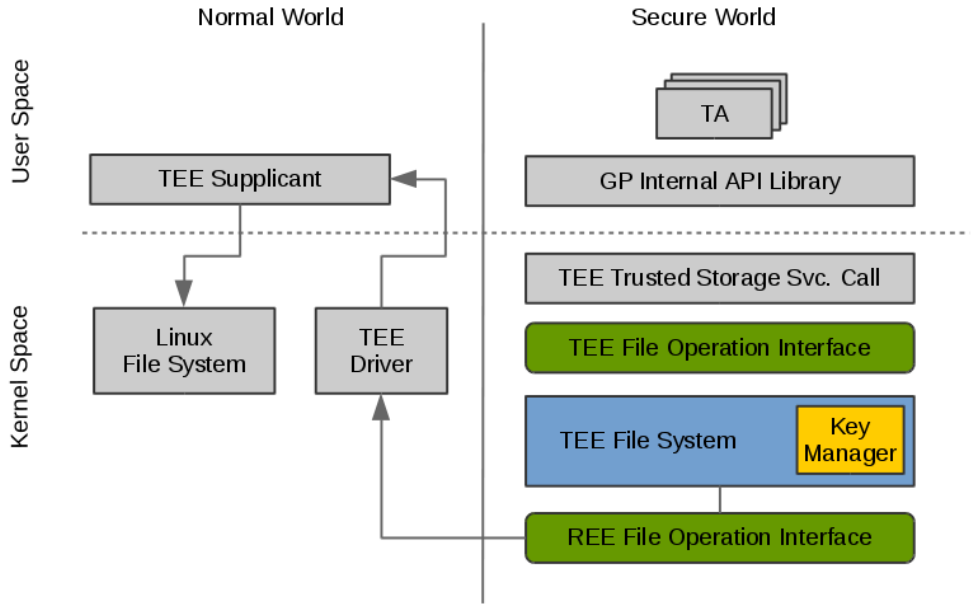

Figure 10.1: A depiction of the secure storage implementation relying on the REE [20].

Exploiting Code in the Secure World A common entry point for attacks is programming mistakes made by the developers, which can lead to the injection and execution of malicious code. Attackers commonly use memory-based attacks, such as buffer overflows, to execute malicious code which allows the manipulation of the current process' control flow. For example, the OP-TEE's CVEs found in 2019 [124, 125, 126] could be used to execute malicious code.

Over the years, developers added multiple mechanisms to compilers, such as `clang` [103] and GCC [66], to make it more difficult to exploit memory-based vulnerabilities in an executable. The compilers apply static analyses to find errors in a program or, e.g., ensure that a program follows a specific execution path using *Control Flow Integrity* checks. Although the OP-TEE supports many of the mechanisms, some are not widely available yet. For example, address space layout randomisation, which makes it harder to perform certain types of memory-based attacks, is only supported as of OP-TEE version 3.8 [214]. Another security mechanism used by the OP-TEE [23] is *Privileged Execute Never* [10]. This mechanism prevents user code from executing code in privileged memory regions.

Additionally, not all kinds of vulnerabilities can be fixed by the mechanisms applied during the compilation of C, C++, and assembly code. A drawback of these programming languages are memory issues [46] that, e.g., are the cause of OpenSSL's heartbleed [45] or a heap-based overflow in `sudo` [127], which allowed attackers to gain privileged access to a system.

In 2020, Wan et. al [52] presented the *Teaclave TrustZone SDK*. It builds upon the OP-TEE project and can be used on every platform that is supported by the OP-TEE. The important fact about the *Teaclave TrustZone SDK* is that this secure operating system is written in the Rust programming language. The compiler detects common memory problems, such as data races, dangling pointers, or null pointers at compile time, resulting in a failed compilation, requiring developers to resolve these issues for a successful build. Buffer overflows can also be detected, but only in debug build, as the checks lead to a performance penalty.

Leveraging compiler checks that apply additional sanity checks on the code, best-practice coding standards, and the use of memory-safe programming languages, such as Rust, help to mitigate issues. Eradicating these issues is not feasible, though, as there can be other error sources that cannot be addressed by a project, such as issues with the underlying hardware.

Manipulating Code Repositories. A second attack vector is to attack the code base for the trusted firmware. This approach has the added benefit that a rootkit can be directly added to a code base and is directly included in the firmware.

In the last couple of years, there have been multiple cases of attackers taking over code repositories, leading to the insertion of malicious code. Notable cases include the insertion of malicious code into the Gentoo Linux repository in 2018 [58]. An attacker included commands whose execution would have deleted a user's hard drive. Due to several safeguards, this code could never be executed, though.

In 2021, attackers added malicious code in *ua-parser-js* [51], a widely used JavaScript library. A more recent example is from the beginning of 2023, when attackers compromised Github [62] and stole two code-signing certificates. These certificates would have allowed attackers to redistribute malware that would be able to impersonate a legitimate application.

In 2024, a backdoor in the utility *xz* was found [33] that is used by security-critical applications, such as *SSH*. A maintainer added an obfuscated backdoor that would have allowed an attacker to gain access to a system without providing the proper credentials.

These examples show that it is not completely unlikely that code repositories can be used as starting points to take over systems.

Problems like these are difficult to prevent, as a single compromised account could be enough. Commonly, attacks such as phishing [89] are used to steal credentials that allow access to project repositories in this case. Mechanisms, such as multifactor authentication [131], which forces people to use of at least a second token, in addition to good password rules [67] can help to prevent a successful account take-over. As with the vulnerabilities in the software, the eradication of all issues is not feasible. Misconfigured servers, usage of deprecated software, etc. may all lead to attackers gaining access to an organisation's repository.

In this situation, another problem is how a project handles its dependencies. The trusted firmware consists of multiple projects that all provide different kinds of functions. If a single project gets compromised and malicious code is inserted, the complete build setup for the trusted firmware and the built artifact itself must be considered compromised.

Adding Malicious Code as Trusted Application. The OP-TEE provides the ability to add additional trusted applications that run in the secure world. Besides *Pseudo Trusted Applications*, commonly abbreviated as PTA, which is built directly into the trusted firmware, there are also *User Mode Trusted Applications*. The OP-TEE maps *User Mode Trusted Applications* directly into the memory of the secure world. These trusted applications are ELF files [194] that are signed and optionally encrypted. The correct signature verifies that the executable file is supposed to be loaded into the secure world. Flaws in the signature verification or leaked private keys can lead to attackers adding malicious software in this way. Depending on the attacked system, it could suffice to delete the executables. For example, Android mostly uses a read-only filesystem and remounting it is commonly not possible due to technical limitations [217]. Thus, it would not be possible to remove the malicious software without, e.g. applying a software update or formatting the system.

CHAPTER 11

Future Work

The current implementation of the rootkit poses the following research questions for further work:

1. What other functionality can be realised with an Arm TrustZone-based rootkit?
2. How can a rootkit in the OP-TEE dynamically adapt to a system?

One could argue that better hiding mechanisms should also be in the scope of future work. At the time of writing this thesis, there is, to the best of our knowledge, no mechanism that could properly identify Arm TrustZone rootkits. The approaches of Singh et al. [176] and Zhou et al. [231] might be successful in finding these types of rootkits. This would require adapting the respective machine-learning algorithms, and would further be complicated by the small sample size for the system's training.

11.1 Extending the Trusted Application

The rootkit introduced in this thesis does not extensively process the data in the secure world. The data that is sent to the TA is currently only printed and not further processed.

For an attacker, the data obtained by intercepting the binder might be valuable. As such, it needs to be further processed to make it usable.

11.1.1 Persisting the Intercepted Data

An issue for the current implementation is to persistently store the data. The current implementation can only store data in volatile memory. Thus, a reboot of the system would lead to a loss of the collected data. The OP-TEE [137] offers an API to securely store data. The secure storage API has currently two implementations:

1. one backed by the file system in the normal world
2. one backed by a separate location in the EMMC, only accessible to the secure world

In both cases, the support in the OP-TEE must be configured at compile time. The HiKey 960 development board currently supports neither implementation [139]. In general, in the OP-TEE version 3.13.0, which was used for this thesis, only boards of NXP Semiconductors had a configuration [138] to enable either implementation.

Thus, the best option would most likely be to find a separate solution that does not rely on the secure storage API, as it is most likely unavailable on the attacked system.

11.1.2 Transmitting the Intercepted Data

In addition to storing intercepted data on a compromised device, the data theoretically could also be extracted by using the network and Bluetooth capabilities of a device. The big issue is that the OP-TEE does not have drivers to access either communication technology. To transmit the intercepted data, it would be necessary to implement the drivers for wireless devices. The OP-TEE allows accessing the needed hardware interfaces, but the development of the drivers would be very time-consuming.

11.1.3 Modifying the Intercepted Data

A lot of data generated in an Android system passes the Binder. Modifying the input of the keyboard can be used to, e.g., redirect money to another account, change passwords to predefined values, or redirect users to malicious websites. Adding the support for data modification to the rootkit would require additional modifications in the assembly code. Currently, the shellcode copies data from the userspace to the kernelspace in the normal world, before it is eventually transferred to the secure world. To apply changes, it would be necessary to implement the data transfer in the reverse direction. Modifying data requires to update the length descriptors, offsets, and potential checksums that are related to data.

11.2 Dynamically Find the Kernel Symbols

A big drawback of the current implementation is that the rootkit is engineered to work on a certain combination of hardware and software. The deployment of the rootkit on a different platform or a device with an entirely different software version would fail. The addresses of kernel symbols differ due to a different target platform, compiler, and applied optimisations.

The rootkit uses several kernel functions to copy data from and to the kernelspace, allocate buffers, access the rootkit, etc. Without access to these functions, the rootkit cannot work as intended.

As Zhang, Meng & Wang [227] as well as Igal [100] have shown, it is possible to find the kernel symbols in the memory. The mechanisms in the respective works would need to be updated to work with current Linux kernels; if that is even possible. For example, the mechanism developed by Igal to determine the address of kernel symbols highly depends on a 32-bit Linux kernel and kernel structures, which were modified since the initial release of the work.

CHAPTER 12

Conclusion & Outlook

The Arm TrustZone is a security extension that allows to split a system into a normal world and a secure world. Resources allocated to the secure world can only be accessed by trusted applications in the secure world, preventing attackers from extracting sensitive data or manipulating security-sensitive computations. The Arm TrustZone is used by projects such as Android and Samsung's Knox to secure the respective system and cryptographic material among other things.

Usually, the Arm TrustZone is considered a trusted component. Marth et. al [118] and Roth [160] suggested the idea of a malicious actor attacking a system from the TEE. The former work showcases a proof-of-concept implementation of an Arm TrustZone-based rootkit, which can covertly grant elevated privileges to processes, among other things.

The rootkit presented in this thesis showcases an attack on a system using the Arm TrustZone to intercept data passing through the Binder. The rootkit in the Arm TrustZone searches and hooks functions in the normal world to redirect the data flow to itself, where it can be safely stored for an attacker.

What makes it so problematic is that Android extensively uses the Binder to transfer data from one process to another. For example, virtual keyboards on Android phones pass the user input via the Binder to the process that shows the input field. Thus, sensitive information, such as message contents, passwords, and bank data are all interceptable at a single point.

The proof-of-concept implemented as part of this thesis finds the `binder_ioctl_write_read` function, using a simple search algorithm. This function processes the `binder_write_read` struct, which is central to the Binder. By replacing the instructions in the identified function, a proxy function can be inserted. The proxy function forwards the control flow to shellcode that copies the `binder_write_read` struct and sends this copy to the rootkit residing in the Arm TrustZone. The rootkit is a pseudo-trusted application as part of the OP-TEE operating system. Once a rootkit

is able to intercept the transferred data, all kinds of modifications could be made to damage the system's user.

As the rootkit resides in the Arm TrustZone, it has access to all of a system's resources. In addition, it is very difficult to remove as it is a persistent part of a system's firmware. Apple's inability to patch the *checkm8*-vulnerability [216], has shown that potential backdoors in the firmware might not be removable at all under certain circumstances.

To find its way into the Arm TrustZone, a rootkit could either exploit a vulnerability in the code or be integrated as a part of a device's firmware, requiring an organisational vulnerability. Both vulnerabilities can be effectively addressed, but there is never a guarantee that they can be completely prevented.

Well written code can prevent vulnerabilities that may be used to load malicious code into the secure world. As C is still a prevalent language for low-level components, there needs to be extra caution taken. Besides static analysis, tests, etc., a switch to Rust could also be a good choice. The Apache Teaclave project [52] aims to provide the same functionality as the OP-TEE but uses memory-safe language Rust instead of C. Using Rust can prevent issues, such as OP-TEE's CVEs found in 2019 [124, 125, 126] that attackers could have used to deploy the rootkit in the secure world.

Organisational security is not easy to address. As various big security incidents caused by, e.g. the Lapsus$ group [39] that broke into Nvidia, Microsoft, and Samsung, among other companies, to extort them with the data obtained during the hacks, have exemplified, it is possible to gain access almost everywhere. In addition, researchers from Trend Micro [222] have shown that a criminal group succeeded in deploying malware during the production of devices. If malware gets deployed during the production of a device, it is almost impossible to remove it. This circumstance highlights the importance of proper organisational rules and a secure development process.

List of Figures

Acronyms

CISC Complex Instruction Set Computer. 25, 26, 27, 28, 30, 31, 32

IoT Internet of Things. 7, 31, 33, 37

IPC Interprocess Communication. 51

ISA Instruction Set Architecture. 25, 26, 29, 36, 37

LKM Linux Kernel Module. 19, 62, 63, 64, 65, 83

MCU Microcontroller Unit. 31, 33

PMU Performance Monitor Unit. 20

PSP Platform Security Processor. 41

REE Rich Execution Environment. 55, 56, 57, 65, 73, 83

RISC Reduced Instruction Set Computer. 25, 26, 27, 28, 29, 30, 31

RPC Remote Procedure Call. 8

SEV Secure Encrypted Virtualization. 41

SGX Software Guard Extensions. 42

SIMD Single Instruction, Multiple Data. 33, 34

TA Trusted Application. 2, 64, 65, 66, 77

TEE Trusted Execution Environment. 1, 2, 3, 7, 8, 35, 37, 70, 72, 81

TTBR Translation Table Base Register. 39, 47

TTBR0 Translation Table Base Register 0. 39

TTBR1 Translation Table Base Register 1. 39

VM Virtual Machine. 6, 7, 20, 28

Bibliography

Print Resources

[22] Ahmed M. Azab et al. "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*. ACM New York, NY, USA, 2014.

[24] Bruno Bierbaumer et al. "Smashing the Stack Protector for Fun and Profit". In: *ICT Systems Security and Privacy Protection: 33rd IFIP TC 11 International Conference, SEC 2018, Held at the 24th IFIP World Computer Congress, WCC 2018, Poznan, Poland, September 18-20, 2018, Proceedings 33*. Springer, 2018.

[27] E. Blem, J. Menon, and K. 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 (HPCA)*. IEEE, 2013.

[28] David Brash. "Extensions to the ARMv7-A architecture". In: *2010 IEEE Hot Chips 22 Symposium (HCS)*. IEEE, 2010.

[29] Rory Bray, Daniel Cid, and Andrew Hay. *OSSEC host-based intrusion detection guide*. Syngress, 2008. ISBN: 159749240X.

[30] Tiago Brito, Nuno O. Duarte, and Nuno Santos. "Arm trustzone for secure image processing on the cloud". In: *2016 IEEE 35th Symposium on Reliable Distributed Systems Workshops (SRDSW)*. IEEE, 2016.

[32] J. Burton Browning and Bruce Sutherland. *C++ 20 Recipes: A Problem-Solution Approach*. Springer, 2020. ISBN: 148425712X.

[34] Kenneth H. Chan, Matthew Pasco, and Betty HC Cheng. "Towards a blockchain framework for autonomous vehicle system integrity". In: *SAE International Journal of Transportation Cybersecurity and Privacy* (2021).

[35] Yue Chen, Mustakimur Khandaker, and Zhi Wang. "Pinpointing vulnerabilities". In: *Proceedings of the 2017 ACM on Asia conference on computer and communications security*. ACM New York, NY, USA, 2017.

[37] Binlin Cheng et al. “Towards paving the way for large-scale windows malware analysis: Generic binary unpacking with orders-of-magnitude performance boost”. In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM New York, NY, USA, 2018.

[39] Michael Conklin, Brian Elzweig, and Lawrence J. Trautman. “Legal Recourse for Victims of Blockchain and Cyber Breach Attacks”. In: *UC Davis Bus. LJ* (2023).

[41] Emanuele Cozzi et al. “Understanding linux malware”. In: *2018 IEEE symposium on security and privacy (SP)*. IEEE, 2018.

[43] Francis M. David et al. “Cloaker: Hardware supported rootkit concealment”. In: *2008 IEEE Symposium on Security and Privacy (sp 2008)*. IEEE, 2008.

[44] Peter J. Denning. “Virtual memory”. In: *ACM Computing Surveys (CSUR)* (1970).

[45] Zakir Durumeric et al. “The matter of heartbleed”. In: *Proceedings of the 2014 conference on internet measurement conference*. ACM New York, NY, USA, 2014.

[46] Mehmet Emre et al. “Translating C to safer Rust”. In: *Proceedings of the ACM on Programming Languages* (2021).

[47] Robert C. Fannon. “An analysis of hardware-assisted virtual machine based rootkits”. PhD thesis. Monterey, California: Naval Postgraduate School, 2014.

[49] Huan Feng and Kang G. Shin. “Understanding and defending the binder attack surface in android”. In: *Proceedings of the 32nd Annual Conference on Computer Security Applications*. ACM New York, NY, USA, 2016.

[50] Xinyue Feng et al. “BehaviorKI: Behavior Pattern Based Runtime Integrity Checking for Operating System Kernel”. In: *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 2018.

[55] Jessie Frazelle. “Securing the boot process”. In: *Communications of the ACM* (2020).

[56] Praveen Gauravaram et al. “On hash functions using checksums”. In: *International Journal of Information Security* (2010).

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

[59] Anders T. Gjerdrum et al. “Performance of Trusted Computing in Cloud Infrastructures with Intel SGX.” In: *CLOSER*. ACM New York, NY, USA, 2017.

[61] Amélie Gonzalez, Djob Mvondo, and Yérom-David Bromberg. “Takeaways of Implementing a Native Rust UDP Tunneling Network Driver in the Linux Kernel”. In: *Proceedings of the 12th Workshop on Programming Languages and Operating Systems*. ACM New York, NY, USA, 2023.

[66] Brian Gough and Richard M. Stallman. “An Introduction to GCC for the GNU Compilers gcc and g++”. In: (1994).

[67] Paul Grassi, Michael Garcia, James Fenton, et al. “NIST Digital Identity Guidelines”. In: *https://csrc. nist. gov/publications/detail/sp/800-63/3/final* (2020).

[68] Thomas Grechenig et al. *Softwaretechnik: mit Fallbeispielen aus realen Entwicklungsprojekten*. Pearson Deutschland GmbH, 2010. ISBN: 3868940073.

[69] Seon Ha et al. “Kernel code integrity protection at the physical address level on RISC-V”. In: *IEEE Access* (2023).

[71] Andreas Haeberlen and Kevin Elphinstone. “User-level management of kernel memory”. In: *Advances in Computer Systems Architecture: 8th Asia-Pacific Conference, ACSAC 2003, Aizu-Wakamatsu, Japan, September 23-26, 2003. Proceedings 8*. Springer, 2003.

[72] Jakob Hagl, Oliver Mann, and Martin Pirker. “Securing the Linux Boot Process: From Start to Finish.” In: *ICISSP 2021*. SciTePress, 2021.

[73] Xueyuan Han et al. “UNICORN: Runtime Provenance-Based Detector for Advanced Persistent Threats”. In: *27th Annual Network and Distributed System Security Symposium, NDSS 2020*. Internet Society, 2020.

[76] Anders Hejlsberg et al. *C# Programming language*. Addison-Wesley Professional, 2010. ISBN: 0321334434.

[78] Ingoo Heo et al. “Efficient Kernel Integrity Monitor Design for Commodity Mobile Application Processors”. In: *Journal of Semiconductor Technology and Science* (2015).

[79] Muhammad El-Hindi et al. “Benchmarking the Second Generation of Intel SGX Hardware”. In: *Data Management on New Hardware*. Association for Computing Machinery, 2022. ISBN: 3319561103.

[80] Sandra Höltervennhoff et al. “{“I} wouldn’t want my unsafe code to run my {pacemaker”}: An Interview Study on the Use, Comprehension, and Perceived Risks of Unsafe Rust”. In: *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, 2023.

[81] Zhichao Hua et al. “vTZ: Virtualizing ARM TrustZone”. In: *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, 2017.

[82] Jian Huang, Moinuddin K. Qureshi, and Karsten Schwan. “An evolutionary study of linux memory management for fun and profit”. In: *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. USENIX Association, 2016.

[83] Wan Huzaini Wan Hussin, Reuben Edwards, and Paul Coulton. “E-pass using drm in symbian v8 os and trustzone: Securing vital data on mobile devices”. In: *2006 International Conference on Mobile Business*. IEEE, 2006.

[86] Xabier Iturbe, Balaji Venu, and Emre Ozer. “Soft error vulnerability assessment of the real-time safety-related ARM Cortex-R5 CPU”. In: *2016 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*. IEEE, 2016.

[87] Ralf Jung et al. “RustBelt: Securing the foundations of the Rust programming language”. In: *Proceedings of the ACM on Programming Languages* (2017).

[89] Mahmoud Khonji, Youssef Iraqi, and Andrew Jones. “Phishing detection: a literature survey”. In: *IEEE Communications Surveys & Tutorials* (2013).

[90] Samuel T. King and Peter M. Chen. “SubVirt: Implementing malware with virtual machines”. In: *2006 IEEE Symposium on Security and Privacy (S&P'06)*. IEEE, 2006.

[91] Gerwin Klein et al. “seL4: Formal verification of an OS kernel”. In: *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. USENIX Association, 2009.

[92] Paul Kocher et al. “Spectre Attacks: Exploiting Speculative Execution”. In: *40th IEEE Symposium on Security and Privacy (S&P'19)*. IEEE, 2019.

[93] Jagadish B. Kotra and John Kalamatianos. “Improving the utilization of micro-operation caches in x86 processors”. In: *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020.

[95] Iggy Krajci et al. “The Intel Mobile Processor”. In: *Android on x86: An Introduction to Optimizing for Intel® Architecture* (2013).

[97] Donghyun Kwon et al. “PrOS: Light-weight Privatized Secure OSes in ARM TrustZone”. In: *IEEE Transactions on Mobile Computing* (2019).

[98] Muhammad Rakha Laayu et al. “Comparison of Acquisition Results on iPhone 7 Plus (iOS 14.8. 1) between Jailbreaking vs Non-Jailbreaking Device”. In: *2022 10th International Conference on Information and Communication Technology (ICoICT)*. IEEE, 2022.

[102] Ralph Langner. “Stuxnet: Dissecting a cyberwarfare weapon”. In: *IEEE Security & Privacy* (2011).

[104] Rodrigo G. Lemos et al. “Inspecting Binder transactions to detect anomalies in Android”. In: *2023 IEEE International Systems Conference (SysCon)*. IEEE, 2023.

[105] Valentina Lenarduzzi et al. “Are sonarqube rules inducing bugs?” In: *2020 IEEE 27th international conference on software analysis, evolution and reengineering (SANER)*. IEEE, 2020.

[107] Moritz Lipp et al. “Meltdown: Reading Kernel Memory from User Space”. In: *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, 2018.

[108] Baozheng Liu et al. “{FANS}: Fuzzing android native system services via automated interface analysis”. In: *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 2020.

[109] Beichen Liu, Pierre Olivier, and Binoy Ravindran. “Slimguard: A secure and memory-efficient heap allocator”. In: *Proceedings of the 20th International Middleware Conference*. ACM New York, NY, USA, 2019.

[110] Mykhaylo Lobur, Serhiy Shcherbovskykh, and Tetyana Stefanovych. “Availability Audit of IoT System Data Reserved by 3-2-1 Backup Strategy based on Fault Tree and State Transition Diagram Analysis”. In: *2020 IEEE 15th International Conference on Computer Sciences and Information Technologies (CSIT)*. IEEE, 2020.

[111] Rafael Lotufo et al. “Evolution of the Linux kernel variability model”. In: *Software Product Lines: Going Beyond: 14th International Conference, SPLC 2010, Jeju Island, South Korea, September 13-17, 2010. Proceedings 14*. Springer, 2010.

[113] Björn Lundgren and Niklas Möller. “Defining information security”. In: *Science and engineering ethics* (2019).

[114] Aravind Machiry et al. “BOOMERANG: Exploiting the Semantic Gap in Trusted Execution Environments.” In: *24th Annual Network and Distributed System Security Symposium, NDSS 2017*. NDSS, 2017.

[115] Jochen Mades et al. “TLS-level security for low power industrial IoT network infrastructures”. In: *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2020.

[116] Takero Magara and Nobuyuki Yamasaki. “Design of Decoded Instruction Cache”. In: *2023 Eleventh International Symposium on Computing and Networking Workshops (CANDARW)*. IEEE, 2023.

[118] Daniel Marth et al. “Abusing Trust: Mobile Kernel Subversion via TrustZone Rootkits”. In: *2022 IEEE Security and Privacy Workshops (SPW)*. IEEE, 2022.

[120] Roger C. Mayer, James H. Davis, and F. David Schoorman. “An Integrative Model of Organizational Trust”. In: *The Academy of Management Review* (1995).

[121] J. Todd McDonald et al. “Phase space power analysis for PC-based rootkit detection”. In: *Proceedings of the 2022 ACM Southeast Conference*. ACM New York, NY, USA, 2022.

[122] Gary McGraw. “Software security”. In: *IEEE Security & Privacy* (2004).

[123] Damiano Melotti, Maxime Rossi-Bellom, and Andrea Continella. “Reversing and Fuzzing the Google Titan M Chip”. In: *Reversing and Offensive-oriented Trends Symposium*. ACM New York, NY, USA, 2021.

[129] Logan Moody et al. “Speculative Code Compaction: Eliminating Dead Code via Speculative Microcode Transformations”. In: *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2022.

[130] Noel Malcolm Morris. *Microelectronic and Microprocessor-based Systems*. Macmillan International Higher Education, 1985. ISBN: 0333361903.

[131] Katelin A. Moul. “Avoid phishing traps”. In: *Proceedings of the 2019 ACM SIGUCCS Annual Conference*. ACM New York, NY, USA, 2019.

[132] Michael Myers and Stephen Youndt. “An introduction to hardware-assisted virtual machine (HVM) rootkits”. In: *Mega Security* (2007).

[133] Anand Nayyar. “An encyclopedia coverage of compiler’s, programmer’s & simulator’s for 8051, pic, avr, arm, arduino embedded technologies”. In: *International Journal of Reconfigurable and Embedded Systems* (2016).

[134] Gerard O’Regan. “Java Programming Language”. In: *The Innovation in Computing Companion*. Springer, 2018. ISBN: 9783030026196.

[140] Hilarie Orman. “The Morris worm: A fifteen-year perspective”. In: *IEEE Security & Privacy* (2003).

[141] Thomas J. Ostrand and Elaine J. Weyuker. “The distribution of faults in a large industrial software system”. In: *ACM SIGSOFT Software Engineering Notes* (2002).

[142] Donn B. Parker. *Fighting computer crime: a new framework for protecting information*. John Wiley & Sons, Inc., 1998. ISBN: 0471163783.

[143] Yale N. Patt et al. “Run-time generation of HPS microinstructions from a VAX instruction stream”. In: *ACM SIGMICRO Newsletter* (1986).

[144] David A. Patterson. “Reduced Instruction Set Computers”. In: *Communications of the ACM* (1985).

[145] Andrea Pellegrini et al. “The arm neoverse n1 platform: Building blocks for the next-gen cloud-to-edge infrastructure soc”. In: *IEEE Micro* (2020).

[147] Fabien AP Petitcolas. “Kerckhoffs’ principle”. In: *Encyclopedia of Cryptography, Security and Privacy*. Springer, 2023. ISBN: 3030715213.

[148] Duy-Phuc Pham, Damien Marion, and Annelie Heuser. “ULTRA: Ultimate Rootkit Detection over the Air”. In: *Proceedings of the 25th International Symposium on Research in Attacks, Intrusions and Defenses*. ACM New York, NY, USA, 2022.

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

[151] Andrei Poenaru et al. “An evaluation of the Fujitsu A64FX for HPC applications”. In: *Presentation in AHUG ISC 21 Workshop*. ACM New York, NY, USA, 2021.

[152] Georgios Portokalidis and Angelos D. Keromytis. “Fast and practical instruction-set randomization for commodity systems”. In: *Proceedings of the 26th Annual Computer Security Applications Conference*. ACM New York, NY, USA, 2010.

[157] Randall C. Reid and Arthur H. Gilbert. “Using the Parkerian Hexad to introduce security in an information literacy class”. In: *2010 Information Security Curriculum Development Conference*. ACM New York, NY, USA, 2010.

[158] Junghwan Rhee et al. “Kernel malware analysis with un-tampered and temporal views of dynamic kernel memory”. In: *Recent Advances in Intrusion Detection: 13th International Symposium, RAID 2010, Ottawa, Ontario, Canada, September 15-17, 2010. Proceedings 13*. Springer, 2010.

[161] Alessandro Rubini. “Kernel Korner: The" Virtual File System" in Linux”. In: *Linux Journal* (1997).

[162] Zella G. Ruthberg and Robert G. McKenzie. “Audit and evaluation of computer security”. In: (1977). URL: https://www.nist.gov/publications/audit-and-evaluation-computer-security?pub_id=900276 (visited on 9/5/2024).

[166] Nuno Santos et al. “Using ARM TrustZone to build a trusted language runtime for mobile applications”. In: *ACM SIGARCH Computer Architecture News*. ACM New York, NY, USA, 2014.

[167] Simone Scalco et al. “On the feasibility of detecting injections in malicious npm packages”. In: *Proceedings of the 17th International Conference on Availability, Reliability and Security*. ACM New York, NY, USA, 2022.

[169] Adriana Sejfia and Max Schäfer. “Practical automated detection of malicious npm packages”. In: *Proceedings of the 44th International Conference on Software Engineering*. ACM New York, NY, USA, 2022.

[170] Alon Shakevsky, Eyal Ronen, and Avishai Wool. “Trust Dies in Darkness: Shedding Light on Samsung’s {TrustZone} Keymaster Design”. In: *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, 2022.

[172] Dong Shen et al. “H-binder: A hardened binder framework on android systems”. In: *Security and Privacy in Communication Networks: 12th International Conference, SecureComm 2016, Guangzhou, China, October 10-12, 2016, Proceedings 12*. Springer, 2017.

[174] Nikolay A. Simakov et al. “Are we ready for broader adoption of ARM in the HPC community: Performance and Energy Efficiency Analysis of Benchmarks and Applications Executed on High-End ARM Systems”. In: *Proceedings of the HPC Asia 2023 Workshops*. ACM New York, NY, USA, 2023.

[175] Richard O. Simpson and Phillip D. Hester. “The IBM RT PC ROMP processor and memory management unit architecture”. In: *IBM Systems Journal* (1987).

[176] Baljit Singh et al. “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*. ACM, 2017.

[177] Liantao Song et al. “TZ-IMA: Supporting Integrity Measurement for Applications with ARM TrustZone”. In: *International Conference on Information and Communications Security*. Springer, 2022.

[178] Wonjun Song et al. “{PIkit}: A New {Kernel-Independent}{Processor-Interconnect} Rootkit”. In: *25th USENIX Security Symposium (USENIX Security 16)*. USENIX Association, 2016.

[180] Matt Spisak. “Hardware-Assisted Rootkits: Abusing Performance Counters on the ARM and x86 Architectures”. In: *10th USENIX Workshop on Offensive Technologies (WOOT 16)*. USENIX Association, 2016.

[182] Marc Stevens et al. “The first collision for full SHA-1”. In: *Advances in Cryptology–CRYPTO 2017: 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20–24, 2017, Proceedings, Part I 37*. Springer, 2017.

[183] Alen Stojanov et al. “SIMD intrinsics on managed language runtimes”. In: *Proceedings of the 2018 International Symposium on Code Generation and Optimization*. ACM New York, NY, USA, 2018.

[184] Jacob Stringer et al. “Technical lag of dependencies in major package managers”. In: *2020 27th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2020.

[185] Fredrik Strupe and Rakesh Kumar. “Uncovering hidden instructions in Armv8-A implementations”. In: *Hardware and Architectural Support for Security and Privacy*. ACM New York, NY, USA, 2020. ISBN: 9781450388986.

[186] Mahesh Subramon, David Kramer, and Indrani Paul. “AMD Ryzen™ 7040 Series: Technology Overview”. In: *2023 IEEE Hot Chips 35 Symposium (HCS)*. IEEE Computer Society, 2023.

[187] G. Edward Suh, Charles W. O’Donnell, and Srinivas Devadas. “AEGIS: A single-chip secure processor”. In: *Information Security Technical Report* (2005).

[188] Hema Karnam Surendrababu. “System Integrity–A Cautionary Tale”. In: *2022 IEEE Physical Assurance and Inspection of Electronics (PAINE)*. IEEE, 2022.

[189] Kuniyasu Suzaki et al. “Library implementation and performance analysis of GlobalPlatform TEE Internal API for Intel SGX and RISC-V Keystone”. In: *2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*. IEEE, 2020.

[191] Andrew S. Tanenbaum. *Modern operating systems*. Pearson India, 2016. ISBN: 9789332575776.

[195] Romain Thomas. “DroidGuard: A deep dive into SafetyNet”. In: *Symposium sur la sécurité des technologies de l’information et des communications (SSTIC)*. SSTIC, 2022.

[196] Ken Thompson. “Reflections on trusting trust”. In: *Communications of the ACM* (1984).

[197] Donghai Tian et al. “A Kernel Rootkit Detection Approach Based on Virtualization and Machine Learning”. In: *IEEE Access* (2019).

[204] Duc-Ly Vu et al. “Typosquatting and combosquatting attacks on the python ecosystem”. In: *2020 ieee european symposium on security and privacy workshops (euros&pw)*. IEEE, 2020.

[206] Bei Wang, Bo Wang, and Qingqing Xiong. “The comparison of communication methods between user and Kernel space in embedded Linux”. In: *International Conference on Computational Problem-Solving*. IEEE, 2010.

[207] Haoyu Wang et al. “Characterizing android app signing issues”. In: *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019.

[208] Jiang Wang, Angelos Stavrou, and Anup Ghosh. “Hypercheck: A hardware-assisted integrity monitor”. In: *International Workshop on Recent Advances in Intrusion Detection*. Springer, 2010.

[209] KC Wang. “ARMv8 Architecture and Programming”. In: *Embedded and Real-Time Operating Systems*. Springer, 2023. ISBN: 9783031287008.

[211] Xueyang Wang and Ramesh Karri. “Numchecker: Detecting kernel control-flow modifying rootkits by using hardware performance counters”. In: *2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, 2013.

[212] Lindsay Watson, Patricia Watson, et al. *Juvenal: Satire 6*. Cambridge University Press, 2014. ISBN: 0521854911.

[213] Ahmad Samer Wazan et al. “RootAsRole: Towards a Secure Alternative to sudo/su Commands for Home Users and SME Administrators”. In: *IFIP International Conference on ICT Systems Security and Privacy Protection*. Springer, 2021.

[216] Jiajian Wu et al. “A research of digital forensic method based on the Checkm8 heap vulnerability”. In: *2021 IEEE 2nd International Conference on Information Technology, Big Data and Artificial Intelligence (ICIBA)*. IEEE, 2021.

[218] Xiaobo Xiang et al. “Ghost in the binder: Binder transaction redirection attacks in Android system services”. In: *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. ACM New York, NY, USA, 2021.

[219] Xiongwei Xie and Weichao Wang. “Rootkit detection on virtual machines through deep information extraction at hypervisor-level”. In: *2013 IEEE Conference on Communications and Network Security (CNS)*. IEEE, 2013.

[221] Weitian Xing, Yuanhui Cheng, and Werner Dietl. “Ensuring correct cryptographic algorithm and provider usage at compile time”. In: *Proceedings of the 23rd ACM International Workshop on Formal Techniques for Java-like Programs*. ACM New York, NY, USA, 2021.

[223] Kenichi Yasukata et al. “zpoline: a system call hook mechanism based on binary rewriting”. In: *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. USENIX Association, 2023.

[224] Dong-Hoon You and Bong-Nam Noh. “Android platform based linux kernel rootkit”. In: *2011 6th International Conference on Malicious and Unwanted Software*. IEEE, 2011.

[225] Nezer Jacob Zaidenberg. “Hardware rooted security in industry 4.0 systems”. In: *Cyber Defence in Industry 4.0 Systems and Related Logistics and IT Infrastructures* (2018).

[226] Hongyu Zhang. “On the distribution of software faults”. In: *Transactions on Software Engineering* (2008).

[227] Shuhui Zhang, Xiangxu Meng, and Lianhai Wang. “An adaptive approach for Linux memory analysis based on kernel code reconstruction”. In: *EURASIP Journal on Information Security* (2016).

[228] Zixuan Zhang. “Analysis of the Advantages of the M1 CPU and Its Impact on the Future Development of Apple”. In: *2021 2nd International Conference on Big Data & Artificial Intelligence & Software Engineering (ICBASE)*. IEEE, 2021.

[229] Kuo Zhao et al. “Design and implementation of secure auditing system in linux kernel”. In: *2007 International Workshop on Anti-Counterfeiting, Security and Identification (ASID)*. IEEE, 2007.

[230] Lei Zhou et al. “Hardware-assisted Live Kernel Function Updating on Intel Platforms”. In: *IEEE Transactions on Dependable and Secure Computing* (2023).

[231] 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)*. IEEE, 2018.

[232] Markus Zimmermann et al. “Small world with high risks: A study of security threats in the npm ecosystem”. In: *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, 2019.

[233] Nikola Zlatanov. “ARM Architecture and RISC Applications”. In: *IEEE Computer Society* (2016).

Online Resources

[1] Marat Akhin and Mikhail Belyaev. *Kotlin language specification*. 2021. URL: https://kotlinlang.org/spec/introduction.html (visited on 9/5/2024).

[2] AlDanial. *cloc*. 2021. URL: https://github.com/AlDanial/cloc (visited on 5/5/2024).

[3] Tiago Alves and Don Felton. *Trustzone: Integrated hardware and software security*. 2004. URL: https://web.archive.org/web/20070415022456/http://www.arm.com/pdfs/TZ%20Whitepaper.pdf (visited on 9/5/2024).

[4] AMD. *AMD64 Architecture Programmer's Manual, Volume 3: General-Purpose and System Instructions*. 2020. URL: https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/programmer-references/24594.pdf (visited on 5/5/2024).

[5] Android. *96boards*. 2023. URL: https://www.96boards.org/product/hikey960/ (visited on 5/5/2024).

[6] Android. *DRM*. 2023. URL: https://source.android.com/docs/core/media/drm (visited on 5/5/2024).

[7] Android. *Modular system components*. 2023. URL: https://source.android.com/docs/core/ota/modular-system (visited on 5/5/2024).

[8] *Android keystore system*. 2020. URL: https://developer.android.com/training/articles/keystore (visited on 5/5/2024).

[9] The Linux Kernel Archives. *Linux Kernel*. 2021. URL: https://www.kernel.org/ (visited on 5/5/2024).

[10] Arm. *Access permissions*. 2023. URL: https://developer.arm.com/documentation/den0024/a/BABCEADG (visited on 5/5/2024).

[11] Arm. *ARM architecture reference manual for A-profile architecture*. 2022. URL: https://developer.arm.com/documentation/ddi0487/latest/ (visited on 9/5/2024).

[12] Arm. *Arm Compiler armasm User Guide*. 2014. URL: https://developer.arm.com/documentation/100069/latest/ (visited on 9/5/2024).

[13] Arm. *ARM Cortex-A Series Programmer's Guide for ARMv8-A*. 2015. URL: https://developer.arm.com/documentation/den0024/latest/Memory-Ordering (visited on 9/5/2024).

[14] Arm. *Arm Cortex-A78 Core Software Optimization Guide*. 2024. URL: https://documentation-service.arm.com/static/6238b1b98804d00769e9deec?token= (visited on 5/5/2024).

[15] Arm. *Arm Trusted Firmware*. 2021. URL: https://github.com/ARM-software/arm-trusted-firmware (visited on 5/5/2024).

[16] Arm. *Firmware Update*. 2024. URL: https://trustedfirmware-a.readthedocs.io/en/latest/components/firmware-update.html#firmware-update-fwu (visited on 5/5/2024).

[17] Arm. *Security in ARMv8-A systems*. 2022. URL: https://developer.arm.com/documentation/100935/latest/ (visited on 9/5/2024).

[18] Arm. *The Cortex-M33 Instruction Set*. 2016. URL: https://developer.arm.com/documentation/100235/0004/the-cortex-m33-instruction-set?lang=en (visited on 5/5/2024).

[19] Arm. *TrustZone technology for ARMv8-M Architecture*. 2020. URL: https://developer.arm.com/documentation/100690/0200/ARM-TrustZone-technology?lang=en (visited on 5/5/2024).

[20] *Arm Types*. 2021. URL: https://community.arm.com/resized-image/__size/1040x0/__key/communityserver-blogs-components-weblogfiles/00-00-00-21-42/6521.7360.A_2B00_R_2B00_and_2B00_M.png (visited on 5/5/2024).

[21] Nitay Artenstein and Idan Revivo. *Man in the binder: He who controls ipc, controls the droid*. 2014. URL: https://www.blackhat.com/docs/eu-14/materials/eu-14-Artenstein-Man-In-The-Binder-He-Who-Controls-IPC-Controls-The-Droid-wp.pdf (visited on 9/5/2024).

[23] Joakim Bech. *What security features does optee support?* 2019. URL: https://github.com/OP-TEE/optee_os/issues/2865 (visited on 5/5/2024).

[25] *binder.c*. 2017. URL: https://elixir.bootlin.com/linux/v4.14/source/drivers/android/binder.c (visited on 5/5/2024).

[26] *binder.h*. 2017. URL: https://elixir.bootlin.com/linux/v4.14/source/include/uapi/linux/android/binder.h (visited on 5/5/2024).

[31] David Brown. *Android Widevine on OP-TEE*. 2016. URL: http://static.linaro.org/connect/las16/Presentations/Thursday/LAS16-406%20-%20Android%20Widevine%20on%20OP-TEE.pdf (visited on 5/5/2024).

[33] BSI. *Kritische Backdoor in XZ für Linux*. 2024. URL: https://www.bsi.bund.de/SharedDocs/Cybersicherheitswarnungen/DE/2024/2024-223608-1032.pdf?__blob=publicationFile&v=3 (visited on 5/5/2024).

[36] Yue Chen et al. *Downgrade Attack on TrustZone*. 2017. URL: https://arxiv.org/pdf/1707.05082 (visited on 5/5/2024).

[38] Victor Chong. *Pre-Compiled AOSP with OP-TEE*. 2021. URL: https://people.linaro.org/~victor.chong/prebuilt/pie/3130/hikey960/ (visited on 18/11/2023).

[40] Victor Costan and Srinivas Devadas. *Intel SGX explained*. 2016. URL: https://eprint.iacr.org/2016/086 (visited on 9/5/2024).

[42] Dino Dai Zovi. *Kernel rootkits*. 2001. URL: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=f5d4c37e9d22536e9869d3adf2a05306b96d3116 (visited on 9/5/2024).

[48] Huan Feng and Kang G. Shin. *BinderCracker: Assessing the robustness of android system services*. 2016. URL: https://arxiv.org/pdf/1604.06964 (visited on 5/5/2024).

[51] Ryan Flowers. *Supply Chain Attack: NPM Library Used By Facebook And Others Was Compromised*. 2021. URL: https://hackaday.com/2021/10/22/supply-chain-attack-npm-library-used-by-facebook-and-others-was-compromised/ (visited on 5/5/2024).

[52] Apache Foundation. *Apache Teaclave TrustZone SDK*. 2023. URL: https://github.com/apache/incubator-teaclave-trustzone-sdk (visited on 5/5/2024).

[53] Linux Foundation. *Linux Standard Base Core Specification for X86-64*. 2015. URL: https://refspecs.linuxfoundation.org/LSB_5.0.0/LSB-Core-generic/LSB-Core-generic.pdf (visited on 5/5/2024).

[54] Rust Foundation. *Build Scripts*. 2023. URL: https://doc.rust-lang.org/cargo/reference/build-scripts.html (visited on 5/5/2024).

[58] Gentoo. *Infrastructure/Incident Reports/2018-06-28 Github*. 2018. URL: https://wiki.gentoo.org/wiki/Project:Infrastructure/Incident_Reports/2018-06-28_Github (visited on 5/5/2024).

[60] *Go Programming Language*. 2024. URL: https://go.dev (visited on 5/5/2024).

[62] Dan Goodin. *BEWARE ... CERTIFICATE REVOCATIONS AHEAD*. 2023. URL: https://arstechnica.com/information-technology/2023/01/github-says-hackers-cloned-code-signing-certificates-in-breached-repository/ (visited on 5/5/2024).

[63] Google. *Common Android Kernel Tree*. 2021. URL: https://android.googlesource.com/kernel/common/ (visited on 5/5/2024).

[64] Google. *libbinder*. 2023. URL: https://android.googlesource.com/platform/frameworks/native/+/refs/heads/main/libs/binder/rust/ (visited on 5/5/2024).

[65] Google. *Using Binder IPC*. 2024. URL: https://source.android.com/docs/core/architecture/hidl/binder-ipc (visited on 5/5/2024).

[70] Dianne Hackborn. *Re: [PATCH 1/6] staging: android: binder: Remove some funny && usage*. 2009. URL: https://lkml.org/lkml/2009/6/25/3 (visited on 5/5/2024).

[74] hasherezade. *Simple userland rootkit - a case study*. 2011. URL: https://blog.malwarebytes.com/threat-analysis/2016/12/simple-userland-rootkit-a-case-study/ (visited on 5/5/2024).

[75] John Heasman. *Implementing and detecting an ACPI BIOS rootkit*. 2006. URL: https://www.blackhat.com/presentations/bh-europe-06/bh-eu-06-Heasman.pdf (visited on 9/5/2024).

[77] Christoph Hellwig. *staging: remove ashmem*. 2022. URL: https://github.com/torvalds/linux/commit/721412ed3d819e767cac2b06646bf03aa158aaec (visited on 5/5/2024).

[84] ISO. *ISO/IEC 9899:2018*. 2018. URL: https://www.iso.org/standard/74528.html (visited on 5/5/2024).

[85] ISO/IEC/IEEE. *IEEE/ISO/IEC International Standard - Information technology Portable Operating System Interface (POSIX(TM)) Base Specifications, Issue 7*. 2009. URL: https://ieeexplore.ieee.org/document/5393893 (visited on 9/5/2024).

[88] Kdm. *NTIllusion: A portable Win32 userland rootkit*. 2004. URL: http://phrack.org/issues/62/12.html (visited on 5/5/2024).

[94] Xeno Kovah and Corey Kallenberg. *How Many Million BIOSes Would you Like to Infect?* 2015. URL: https://web.archive.org/web/20230326202719/http://www.legbacore.com/Research_files/HowManyMillionBIOSesWouldYouLikeToInfect_Whitepaper_v1.pdf (visited on 9/5/2024).

[96] Greg Kroah-Hartman. *PM: wakeup: simplify the output logic of pm_show_wakelocks()*. 2022. URL: https://github.com/torvalds/linux/commit/c9d967b2ce40d71e968eb839f36c936b8a9cf1ea (visited on 5/5/2024).

[99] Linux Kernel Labs. *Linux Kernel Teaching*. 2022. URL: https://linux-kernel-labs.github.io/ (visited on 5/5/2024).

[100] laginimaineb. *Effectively bypassing kptr_restrict on Android*. 2015. URL: https://bits-please.blogspot.com/2015/08/effectively-bypassing-kptrrestrict-on.html (visited on 5/5/2024).

[101] laginimaineb. *static_kallsyms*. 2017. URL: https://github.com/laginimaineb/static_kallsyms (visited on 5/5/2024).

[103] Chris Lattner. *LLVM and Clang: Next generation compiler technology*. 2008. URL: https://www.llvm.org/pubs/2008-05-17-BSDCan-LLVMIntro.pdf (visited on 9/5/2024).

[106] Rust for Linux. *Android Binder Driver*. 2024. URL: https://rust-for-linux.com/android-binder-driver (visited on 5/5/2024).

[112] Robert Love. *ashmem: Anonymous shared memory subsystem*. 2015. URL: https://github.com/torvalds/linux/commit/11980c2ac4ccfad21a5f8ee9e12059f1e687bb40 (visited on 5/5/2024).

[117] Catalin Marinas. *Memory Layout on AArch64 Linux*. 2024. URL: https://www.kernel.org/doc/html/v5.3/arm64/memory.html (visited on 5/5/2024).

[119] Wolfgang Mauerer et al. *Real-Time Android: Deterministic Ease of Use*. 2012. URL: https://lfdr.de/Publications/2014/EWC-Mauerer.pdf (visited on 9/5/2024).

[124] MITRE. *CVE-2019-1010296*. 2019. URL: https://nvd.nist.gov/vuln/detail/CVE-2019-1010296 (visited on 5/5/2024).

[125] MITRE. *CVE-2019-1010297*. 2019. URL: https://nvd.nist.gov/vuln/detail/CVE-2019-1010297 (visited on 5/5/2024).

[126] MITRE. *CVE-2019-1010298*. 2019. URL: https://nvd.nist.gov/vuln/detail/CVE-2019-1010298 (visited on 5/5/2024).

[127] MITRE. *CVE-2021-3156*. 2021. URL: https://nvd.nist.gov/vuln/detail/CVE-2021-3156 (visited on 5/5/2024).

[128] MITRE. *CVE-2022-0847*. 2022. URL: https://nvd.nist.gov/vuln/detail/CVE-2022-0847 (visited on 5/5/2024).

[135] *OP-TEE*. 2019. URL: https://github.com/OP-TEE/optee_os (visited on 5/5/2024).

[136] Michael Opdenacker. *Embedded Linux size reduction techniques*. 2017. URL: http://events17.linuxfoundation.org/sites/events/files/slides/opdenacker-embedded-linux-size-reduction-techniques_0.pdf (visited on 9/5/2024).

[137] *Open Portable Trusted Execution Environment*. 2021. URL: https://www.op-tee.org/ (visited on 5/5/2024).

[138] *optee_os*. 2021. URL: https://github.com/OP-TEE/optee_os/blob/30c13f9e2ff178c9a299e409de75d50529cf5064/core/arch/arm/plat-imx/conf.mk#L370 (visited on 5/5/2024).

[139] *optee_os*. 2024. URL: https://github.com/OP-TEE/optee_os/blob/16fbd46d245d634778b9df729e3909d6bfd9a79b/core/arch/arm/plat-hikey/conf.mk (visited on 5/5/2024).

[146] Jason Perlow. *1991's PC technology was unbelievable*. 2021. URL: https://www.zdnet.com/article/1991s-pc-technology-was-unbelievable/ (visited on 5/5/2024).

[150] plaguez. *Weakening the Linux Kernel*. 1998. URL: http://phrack.org/issues/52/18.html (visited on 5/5/2024).

[153] procps-ng. *procps*. 2021. URL: https://gitlab.com/procps-ng/procps (visited on 5/5/2024).

[154] *Programming ARM TrustZone Architecture on the Xilinx Zynq-7000 All Programmable SoC*. 2014. URL: https://www.xilinx.com/support/documentation/user_guides/ug1019-zynq-trustzone.pdf (visited on 5/5/2024).

[155] T. Ptacek, Nate Lawson, and P. Ferrie. *Dont tell joanna, the virtualized rootkit is dead*. 2007. URL: https://www.blackhat.com/docs/us-15/materials/us-15-Shen-Attacking-Your-Trusted-Core-Exploiting-Trustzone-On-Android-wp.pdf (visited on 9/5/2024).

[156] Qualcomm. *Code Aurora git repositories*. 2021. URL: https://source.codeaurora.org/quic/kernel (visited on 5/5/2024).

[159] Dan Rosenberg. *Qsee trustzone kernel integer over flow vulnerability*. 2014. URL: https://www.blackhat.com/docs/us-14/materials/us-14-Rosenberg-Reflections-On-Trusting-TrustZone-WP.pdf (visited on 5/5/2024).

[160] Thomas Roth. *Next generation mobile rootkits*. 2013. URL: https://hacinparis.com/data/slides/2013/Slidesthomasroth.pdf (visited on 5/5/2024).

[163] Joanna Rutkowska. *Introducing Blue Pill*. 2006. URL: https://blog.invisiblethings.org/2006/06/22/introducing-blue-pill.html (visited on 5/5/2024).

[164] Spyridon Samonas and David Coss. *The CIA strikes back: Redefining confidentiality, integrity and availability in security*. 2014. URL: https://www.proso.com/dl/Samonas.pdf (visited on 9/5/2024).

[165] *SAMSUNG Knox*. 2021. URL: https://www.samsungknox.com/en (visited on 5/5/2024).

[168] Martin Schwidefsky. *[S390] remove export of sys_call_table*. 2006. URL: https://github.com/torvalds/linux/commit/8f27766a883149926e7c1f69d9f1d8f68efcd65f (visited on 5/5/2024).

[171] Di Shen. *Exploiting TrustZone on android*. 2015. URL: https://www.blackhat.com/docs/us-15/materials/us-15-Shen-Attacking-Your-Trusted-Core-Exploiting-Trustzone-On-Android-wp.pdf (visited on 9/5/2024).

[173] *SierraTEE Trusted Execution Environment*. 2021. URL: https://web.archive.org/web/20230204123939/https://sierraware.com/open-source-ARM-TrustZone.html (visited on 5/5/2024).

[179] Philip Sparks. *The route to a trillion devices*. 2017. URL: https://community.arm.com/cfs-file/__key/telligent-evolution-components-attachments/01-1996-00-00-00-01-30-09/Arm-_2D00_-The-route-to-a-trillion-devices-_2D00_-June-2017.pdf (visited on 9/5/2024).

[181] *Steam Hardware & Software Survey: March 2024*. 2024. URL: https://web.archive.org/web/20240415003003/https://store.steampowered.com/hwsurvey/ (visited on 5/5/2024).

[190] *System Debug User and Reference Guide*. 2020. URL: https://web.archive.org/web/20200811195610/https://software.intel.com/content/www/us/en/develop/documentation/system-debug-user-guide/top.html (visited on 5/5/2024).

[192] Satya Tangirala and Sumit Semwal. *State of Android on Mainline Kernels*. 2021. URL: https://linuxplumbersconf.org/event/7/contributions/785/attachments/532/946/State_of_Android_on_Mainline_Kernels__LPC.pdf (visited on 5/5/2024).

[193] OP-TEE. *Secure storage*. 2021. URL: https://optee.readthedocs.io/en/latest/architecture/secure_storage.html (visited on 5/5/2024).

[194] OP-TEE. *Trusted Applications*. 2021. URL: https://optee.readthedocs.io/en/latest/architecture/trusted_applications.html (visited on 5/5/2024).

[198] tlynn. *Avoid generating metadata in pip download –no-deps ...* 2014. URL: https://github.com/pypa/pip/issues/1884 (visited on 5/5/2024).

[199] Linus Torvalds. *EFI*. 2006. URL: https://yarchive.net/comp/linux/efi.html (visited on 5/5/2024).

[200] *Trusty Tee*. 2020. URL: https://source.android.com/docs/security/features/trusty (visited on 5/5/2024).

[201] Orange Tsai. *Breaking Parser Logic: Take Your Path Normalization off and Pop 0days Out! Black Hat USA, 2018*. 2018. URL: https://i.blackhat.com/us-18/Wed-August-8/us-18-Orange-Tsai-Breaking-Parser-Logic-Take-Your-Path-Normalization-Off-And-Pop-0days-Out-2.pdf (visited on 5/5/2024).

[202] Hannes Tschofenig et al. *Arm's Platform Security Architecture (PSA) Attestation Token*. 2020. URL: https://www.ietf.org/archive/id/draft-tschofenig-rats-psa-token-22.html (visited on 9/5/2024).

[203] *uaccess.h*. 2017. URL: https://elixir.bootlin.com/linux/latest/source/include/linux/uaccess.h (visited on 5/5/2024).

[205] Ben Walshe. *A Brief History of Arm: Part 1*. 1999. URL: https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/a-brief-history-of-arm-part-1 (visited on 5/5/2024).

[210] Xiao Wang et al. *TKRD: Trusted kernel rootkit detection for cybersecurity of VMs based on machine learning and memory forensic analysis*. 2019. URL: https://www.aimspress.com/article/10.3934/mbe.2019132 (visited on 9/5/2024).

[214] Jens Wiklander. *ASLR in OP-TEE*. 2021. URL: https://static.linaro.org/connect/lvc21/presentations/lvc21-118.pdf (visited on 5/5/2024).

[215] Jens Wiklander. *tee: generic TEE subsystem*. 2017. URL: https://github.com/torvalds/linux/commit/967c9cca2cc50569efc65945325c173cecba83bd (visited on 5/5/2024).

[217] John Wu. *Remount Android ext4*. 2019. URL: https://twitter.com/topjohnwu/status/1170404631865778177 (visited on 5/5/2024).

[220] Xiaowen Xin. *Titan M makes Pixel 3 our most secure phone yet*. 2018. URL: https://blog.google/products/pixel/titan-m-makes-pixel-3-our-most-secure-phone-yet/ (visited on 5/5/2024).

[222] Fyodor Yarochkin et al. *Behind the Scenes: How Criminal Enterprises Pre-infect Millions of Mobile Devices*. 2021. URL: https://www.blackhat.com/asia-23/briefings/schedule/index.html#behind-the-scenes-how-criminal-enterprises-pre-infect-millions-of-mobile-devices-31235 (visited on 5/5/2024).