DocuSign Envelope ID: E4E568A1-8978-433E-A15E-38A89FE5B794

**Budapest University of Technology and Economics**

Faculty of Electrical Engineering and Informatics

Department of Networked Systems and Services

# Detection of persistent rootkit components on embedded IoT devices

Bachelor's Thesis

*Author*

Krisztián Németh

*Advisors*

Dr. Levente Buttyán

Dorottya Futóné Papp

December 8, 2020

DocuSign Envelope ID: E4E568A1-8978-433E-A15E-38A89FE5B794

# Contents

**Kivonat** **i**

**Abstract** **ii**

**1 Introduction** **1**

1.1 On the subject of IoT security . . . . . . . . . . . . . . . . . . . . . . . . 1

1.2 Our contribution to this field . . . . . . . . . . . . . . . . . . . . . . . . . 4

**2 Rootkits and their detection** **6**

2.1 Rootkit classification and techniques . . . . . . . . . . . . . . . . . . . . 7

2.2 Persistence techniques . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9

2.3 Hiding persistent components . . . . . . . . . . . . . . . . . . . . . . . . 11

2.4 Rootkit detection . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13

2.5 Existing solutions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15

**3 Technical background** **17**

3.1 Addressing the fundamental issues of rootkit detection . . . . . . . . . 17

3.2 Trusted Execution Environments . . . . . . . . . . . . . . . . . . . . . . 19

3.2.1 ARM TrustZone . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20

3.2.2 OP-TEE . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22

**4 The scope of our work** **24**

DocuSign Envelope ID: E4E568A1-8978-433E-A15E-38A89FE5B794

**5 Design** **27**

5.1 The architecture of the detection software . . . . . 27

5.1.1 The client application . . . . . 27

5.1.2 The trusted application . . . . . 28

5.2 Possible approaches to File Integrity Monitoring . . . . . 30

5.2.1 The dynamic approach . . . . . 30

5.2.2 The static approach . . . . . 33

5.3 Secure storage of the baseline . . . . . 34

5.4 Attack surface and security measures . . . . . 35

**6 Implementation** **38**

6.1 Development for OP-TEE . . . . . 38

6.2 The hashing process . . . . . 39

6.3 Accessing REE file systems from the TEE . . . . . 40

6.4 Updating the baseline . . . . . 42

6.5 Securing the detection software . . . . . 43

**7 Evaluation** **45**

7.1 Verification and validation . . . . . 45

7.2 Performance measurements . . . . . 47

7.3 Limitations and possible improvements . . . . . 48

**8 Conclusion** **51**

**Acknowledgements** **53**

**List of Figures** **55**

**Bibliography** **55**

DocuSign Envelope ID: E4E568A1-8978-433E-A15E-38A89FE5B794

# HALLGATÓI NYILATKOZAT

Alulírott *Németh Krisztián*, szigorló hallgató kijelentem, hogy ezt a szakdolgozatot meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy autentikált felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Budapest, 2020. december 8.

Németh Krisztián

*Németh Krisztián*

hallgató

DocuSign Envelope ID: E4E568A1-8978-433E-A15E-38A89FE5B794

# Kivonat

Az IoT rendszerek gyakran célpontjai kibertámadásoknak, és különösen sebezhetőek a kifinomultabb fenyegetésekkel, például rootkitekkel szemben. A rootkitek olyan kártékony szoftverek, amelyek magas jogosultságokkal futnak, és különböző technikákat alkalmaznak annak érdekében, hogy rejtve maradjanak. A memóriában lévő komponenseik mellett a rootkitek rendelkezhetnek a háttértáron megbúvó perzisztens komponensekkel is, melyek segítségével a rendszer újraindítását követően is aktívak tudnak maradni, illetve képesek lehetnek kikerülni a víruskereső szoftverek ellenőrzéseit. A rootkitek és perisztens komponenseik detekciója azért jelent kihívást, mert a rootkitek képesek megfertőzni a kernelt, vagy akár magát az ellenőrző szoftvert is. Ezt figyelembe véve az általunk készített implementáció kihasználja a megbízható végrehajtási környezetek által nyújtott lehetőségeket, hogy megakadályozza a rootkit beavatkozását az ellenőrzési folyamatba. Dolgozatunkban a perzisztens rootkit komponensek detekciójával foglalkozunk, amelyet az IoT eszköz fájlrendszerén lévő fájlok integritásának periodikus ellenőrzésével valósítunk meg. Az elkészült szoftver mellett bemutatjuk a tervezés és az implementáció főbb részleteit, továbbá igazoljuk a megoldásunk helyességét egy saját fejleszétsű rootkittel való tesztelés keretében.

DocuSign Envelope ID: E4E568A1-8978-433E-A15E-38A89FE5B794

# Abstract

IoT systems are subject to cyber attacks, and are especially defenseless against sophisticated threats, such as rootkits. Rootkits are malicious software that run with elevated privileges, and employ a wide variety of techniques to remain hidden. Aside from having components which are loaded into memory, rootkits also have disk resident, persistent components, which they may use to survive system reboots, and to evade detection software. Detecting rootkits and their persistent components is challenging, because rootkits may compromise the kernel, or even the detection software itself. For this reason, we take advantage of a Trusted Execution Environment (TEE), which prevents the rootkit from interfering with our detection process. In this thesis, we address the challenge of persistent rootkit component detection on embedded IoT devices by using File Integrity Monitoring (FIM) to search for changes on the file system of the device, that may indicate the presence of a rootkit. We provide a detailed description of the design and implementation of our software, and the evaluation of our finished application, which is based on testing we performed using a custom rootkit developed for this purpose.

DocuSign Envelope ID: E4E568A1-8978-433E-A15E-38A89FE5B794

# Chapter 1

# Introduction

The Internet has gone through significant transformation in the past decade. The wide availability of low-cost, network-connected system-on-chip devices gave rise to the Internet of Things (IoT) paradigm, which opened the possibility of many new, exciting applications.

The IoT has brought with itself large changes in industry and manufacturing, which has since been named the Fourth Industrial Revolution, or Industry 4.0. It has integrated into transportation and delivery, healthcare, and transformed many of our homes into smart homes. The popularity of the IoT is still on the rise, and the number of connected devices is already measured in billions, and is estimated to double in the next five years, as shown in Figure 1.1.

Unfortunately, the IoT technology has not matured yet, and these devices gained a notoriety for their many issues regarding privacy, and security.

## 1.1 On the subject of IoT security

As the IoT expands, becomes more prevalent in our everyday lives, and also becomes an increasingly important component of critical infrastructure, securing its systems becomes vital. While insecure home IoT devices may expose the owner's personal information, or become the targets of ransomware, the risks of insecure

DocuSign Envelope ID: E4E568A1-8978-433E-A15E-38A89FE5B794

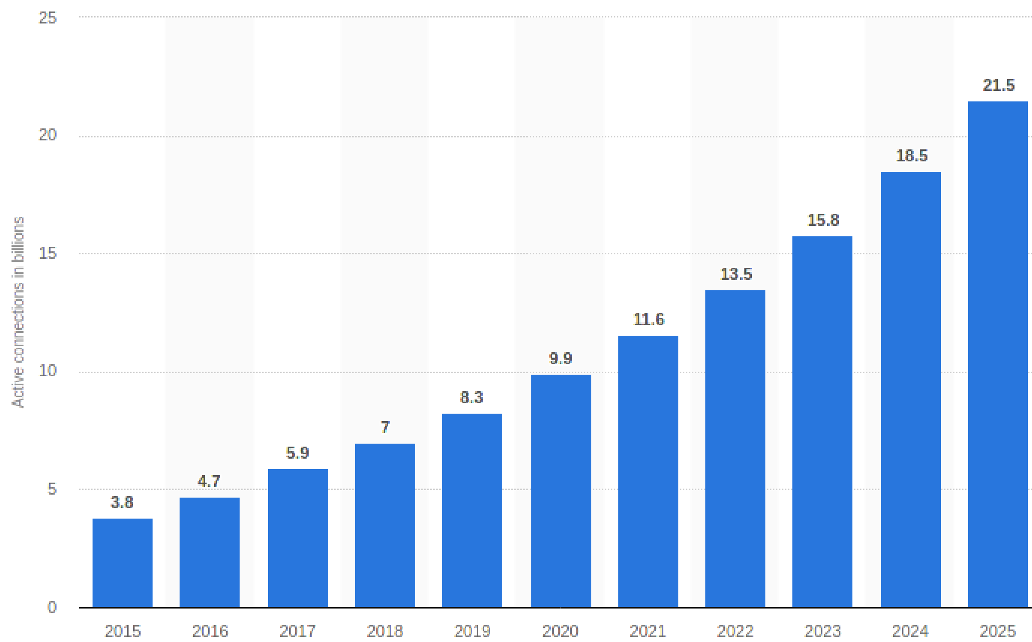


**Figure 1.1:** Number of connected IoT devices worldwide.[1]

cyber-physical devices are much greater. In 2014, attackers were able to remotely take control over a German steel mill's PLCs, and cause physical damage to the facility's blast furnaces[2]. Another, similar incident happened in 2015, when hackers managed to compromise and take offline about 30 electrical substations across Ukraine, causing hours long power outages across the country[3], but the most widely known example of such an incident is the attack against the nuclear program of Iran, where the Stuxnet worm caused the fast-spinning gas centrifuges used to separate nuclear material to tear themselves apart [15]. Based on these past incidents, we can see how security failures may lead to substantial monetary loss, or even loss of human life.

The reason behind the widespread security issues of the IoT is likely the lack of standardisation and regulation. IoT has evolved using a wide range of different technologies and protocols, often with proprietary solutions for security [13]. The growth of the field was rapid, and first-to-market mentality was, and still is, very common. For these reasons, security risk analysis, risk assessment, and countermea-

[1] https://www.statista.com/statistics/1101442/iot-number-of-connected-devices-worldwide/, Last visited: 31.10.2020.

[2] https://ics.sans.org/media/ICS-CPPE-case-Study-2-German-Steelworks_Facility.pdf, Last visited: 30.11.2020.

[3] https://ics.sans.org/media/E-ISAC_SANS_Ukraine_DUC_5.pdf, Last visited: 30.11.2020.

DocuSign Envelope ID: E4E568A1-8978-433E-A15E-38A89FE5B794

sure implementation was unable to keep up with application development, resulting in reliance on 'security through obscurity'. Devices were released using weak, guessable, or hard-coded passwords, and no secure update mechanism while running old, unpatched operating systems and software. The customers, meanwhile, were unaware of these insecurities and the possible risks.

On October 20, 2016, a large-scale distributed denial of service (DDoS) attack was launched against the North American DNS provider Dyn, which caused several websites and services from companies like Amazon, Twitter, GitHub and PayPal to become unreachable for multiple hours throughout the day. The attack was carried out through a massive IoT botnet. A sample of the Mirai malware that has been used to build the botnet was retrieved and analyzed by the research group MalwareMustDie![4] in August 2016, and the source code of the malware has also been published some time later[5]. As demonstrated by the Mirai botnet, hundreds of thousands of Linux based, Internet-connected home-routers, IP cameras and printers with easily brute-forceable passwords can be discovered and compromised through basic techniques [2, 25], and privilege escalation, if needed, can also be easily achieved because of the outdated, unpatched, or poorly-written software running on these devices.

Unfortunately, all this is only one part of the issue. Another part is that once these devices have been compromised, often they have no ability to detect, diagnose and report the infection, leaving backdoors open and malicious software running unimpeded. Even rudimentary malware are capable of achieving persistence on the infected system, surviving system reboots, but their components could be discovered by scanning for suspicious running processes and files.

A more sophisticated form of malware are rootkits. Rootkits run with the highest (root) privilege level, and are capable of modifying system programs and low-level data structures, making themselves invisible for the diagnostic and monitoring tools provided by the operating system (OS). Detecting rootkits and their persistent components is challenging, because any detection program running at the same or lower privilege levels than the rootkit may also be compromised or may be misled by the

[4] `https://web.archive.org/web/20160905023500/http://blog.malwaremustdie.org/2016/08/mmd-0056-2016-linuxmirai-just.html`, Last visited: 08.11.2020.

[5] `https://github.com/jgamblin/Mirai-Source-Code`, Last visited: 08.11.2020.

DocuSign Envelope ID: E4E568A1-8978-433E-A15E-38A89FE5B794

tricks and techniques used by the rootkit to hide itself. What is more, a sufficiently advanced rootkit could avoid detection software designed to find hidden processes and anomalies in the kernel by terminating, restoring the integrity of the memory, but saving itself on the persistent storage for later execution.

## 1.2 Our contribution to this field

In this work, we address the challenges of persistent rootkit component detection on Linux-based embedded IoT devices by leveraging the Trusted Execution Environment (TEE) technology. The TEE is an isolated area of the main processor, which provides a tamper-resistant processing environment with its own, separate system partition and separate kernel. The TEE can guarantee the authenticity of the executed code, and the integrity of the runtime states (e.g CPU registers and memory) [24]. Consequently, even if a rootkit manages to gain root privileges on the main OS, it will not be able to interfere with its operation.

An example for technologies that support TEEs is the ARM TrustZone technology, available on many modern ARM Cortex-A and Cortex-M processors, which are widely deployed in embedded systems[6]. TrustZone is based on the idea of partitioning all of the system's hardware and software into two worlds: the secure world and the normal world. The two worlds implement concurrent secure and non-secure operating systems. The non-secure, general OS running in the normal world (e.g. Linux, Android, etc.) is prevented from accessing certain resources, such as the designated secure regions of the physical memory, while the secure OS, such as OP-TEE[7], running in the secure world, has access to all resources. The isolated and protected software running in the secure OS are known as Trusted Applications (TAs). Implementing our rootkit detection software's components as trusted applications provides two main advantages: it protects the detection software's integrity, preventing malicious modification of the executable binary or the code segment in

[6] `https://developer.arm.com/ip-products/security-ip/trustzone` Last visited: 23.11.2020.

[7] `https://www.op-tee.org/`, Last visited: 22.11.2020.

DocuSign Envelope ID: E4E568A1-8978-433E-A15E-38A89FE5B794

memory, and it eliminates the reliance on the normal world programs and OS, which could be compromised and providing false and misleading information.

Our work was performed as part of a larger project, the aim of which was to develop a fully integrated, TEE-based rootkit detection solution for IoT systems. Our part in this project was the detection of persistent rootkit components on the normal world file systems. To detect persistent components, we implemented a TA capable of automatic File Integrity Monitoring (FIM). FIM is a technology for monitoring and detecting changes in files, that may indicate the presence of malicious software on the device. This is achieved by calculating a secure baseline checksum for a selected group of important files in a controlled environment, and comparing these to the hashes of the same files calculated during operation. The advantages of this technique are its comprehensiveness, and lack of reliance on any malware signature databases, enabling us to potentially detect previously undiscovered or modified rootkits. The main challenges of using FIM lie in being able to provide secure storage for the trusted baseline, and keeping its potentially high false-positive detection rate to a minimum. In order to address these challenges, we leveraged the trusted storage functionalities of the TEE, and have cautiously designed a file system structuring and file monitoring policy.

The rest of this thesis is organized as follows. In Chapter 2, we first provide a more detailed description of rootkits, focusing on the ways they can use and hide persistent components, then we introduce some examples of existing rootkit detection software designed for Linux. In Chapter 3, we show what advantages using a TEE gives our solution in comparison to existing solutions. In Chapter 4, we give an overview of the larger rootkit detection software, and define the scope of our work in the project. Chapter 5 contains the detailed design of the detection software's architecture, and Chapter 6 describes the specifics of the implementation of all its components which relate to the detection of persistent rootkit components. In Chapter 7, we report on the evaluation of our persistent component detection mechanisms, which we performed with a custom rootkit that we developed for this specific purpose, and finally, we conclude this thesis in Chapter 8.

DocuSign Envelope ID: E4E568A1-8978-433E-A15E-38A89FE5B794

# Chapter 2

# Rootkits and their detection

A rootkit is a malicious software, often a collection of different malware components, designed to enable unauthorized access to the infected system's resources, while remaining hidden. The term *rootkit* originates from the name of the privileged user account in Unix-like operating systems, but this type of malicious software is widespread in all types of systems. The name is also an indication to one of the main characteristics of rootkits, which is that they can possess unrestricted, kernel-level access to the infected system.

It is important to mention, that the rootkit itself is only installed on the target system after the attacker has already gained root privileges. To do this, the attacker must find an exploitable vulnerability in the operating system or the installed software, which can be very easy, if the target system is badly secured and running outdated software, or it could require tremendous resources and the discovery of a zero-day exploit, but social engineering and blackmail have also been used for this purpose in the past. The prominent thing to note, is that the presence of a rootkit points to the presence of a vulnerability, which must be found and patched, lest reinfection will inevitably occur.

DocuSign Envelope ID: E4E568A1-8978-433E-A15E-38A89FE5B794

## 2.1 Rootkit classification and techniques

In addition to providing elevated privileges, the other characteristic feature of rootkits is that they are capable of employing a wide variety of techniques to remain hidden. These techniques can provide a basis for the classification of rootkits [23], dividing them into four main types.

*Type 1* rootkits use simple user space techniques and deception to remain hidden, often masquerading as disk-resident system programs (e.g. `login, ps` or `ls`). They are easily detectable via a comparison of their hashes to the hashes of the original system files, or via searching for suspicious running processes, since they were not designed to intercept calls that enumerate files or running programs. Modern rootkits often have no presence on the persistent storage at all, but the ability to temporarily unload from the memory and hide on the file system could be a technique an advanced rootkit could employ to avoid detection software.

*Type 2* rootkits may use hooking – the modification of code or function pointers in programs or dynamically loaded libraries – to divert the flow of code execution. This allows the rootkit to intercept legitimate operating system calls to filter return values, or skip certain checks. There are two main types of hooking techniques: user space hooking, and kernel space hooking. One example of a well-known user space hooking technique is the abuse of the `LD_PRELOAD` environment variable to load malicious libraries before benign ones, therefore overriding their provided functionalities. There exist a number of rootkits employing this technique, such as BEURK[1], Azazel[2], vlany[3] and Jynx2[4]. Kernel space hooking techniques include the replacement of pointers in the system call or interrupt descriptor table, and the use of malicious device drivers and kernel probes. In the past, modifying the kernel memory image by directly writing to `/dev/mem` and `/dev/kmem` was a widely used technique to remain hidden on Linux system. However, recent Linux kernel versions

[1] `https://github.com/unix-thrust/beurk`, Last visited: 25.11.2020.
[2] `https://github.com/chokepoint/azazel`, Last visited: 25.11.2020.
[3] `https://github.com/mempodippy/vlany`, Last visited: 25.11.2020.
[4] `https://github.com/chokepoint/Jynx2`, Last visited: 25.11.2020.

DocuSign Envelope ID: E4E568A1-8978-433E-A15E-38A89FE5B794

restrict access to these device files, mitigating this attack. Diamorphine[5], OSOM[6] and Suterusu[7] all make use of kernel space hooking techniques. Although rootkits that use such techniques are difficult to detect, their limitations lie in that the modification of function behaviour inherently leaves a detectable footprint, because it introduces malicious code, either in the user space or the kernel.

*Type 3* rootkits are able to use direct kernel object modification (DKOM) to subvert the integrity of the kernel by targeting dynamic kernel data structures. This technique can be used, for example, for process hiding. By exploiting that the task schedulers in the operating systems generally use different data structures to track processes than the data structures used for bookkeeping operations, a rootkit could use DKOM to remove its own task structure from the process list or process tree in the kernel, therefore no longer appearing as a running process, but still continuing execution. DKOM attacks are not detectable by user space anti-malware software, since they assume a trusted kernel and rely on it for information. DKOM attacks are also much harder to detect than kernel hooks, because they target dynamic data structures whose values change during normal runtime operation. This technique is used by rootkits such as Adore-ng[8], LilyOfTheValley[9] Reptile[10] and SucKIT [6]. The detection of DKOM often relies on the improper or incomplete implementation of the technique. A rootkit attempting process hiding through the use of DKOM might remove itself from one of the kernel structures, like the process list, but it might appear in another, like the process identifier namespace, meaning that cross-referencing these kernel structures could reveal the presence of a rootkit on the system. In addition, the incorrect implementation of DKOM could easily result in kernel crashes, which may draw attention from the system administrators.

*Type 4* rootkits have been developed in proof-of-concept settings, and have been first discovered in the wild in 2018[11]. They operate at the virtualization layer, in

[5] `https://github.com/m0nad/Diamorphine`, Last visited: 25.11.2020.

[6] `https://github.com/NinnOgTonic/Out-of-Sight-Out-of-Mind-Rootkit`, Last visited: 25.11.2020.

[7] `https://github.com/mncoppola/suterusu`, Last visited: 25.11.2020.

[8] `https://github.com/trimpsyw/adore-ng`, Last visited: 25.11.2020.

[9] `https://github.com/En14c/LilyOfTheValley`, Last visited: 25.11.2020.

[10] `https://github.com/f0rb1dd3n/Reptile`, Last visited: 25.11.2020.

[11] `https://www.welivesecurity.com/wp-content/uploads/2018/09/ESET-LoJax.pdf`, Last visited: 04.12.2020.

DocuSign Envelope ID: E4E568A1-8978-433E-A15E-38A89FE5B794

the BIOS, and in hardware [9] [7]. Such rootkits are called OS-independent rootkits since they reside at a lower level than the operating system, but they are still dependent on the type of BIOS version, instruction set, and hardware. Rootkits in the lowest-level components can survive reboots and re-installations, and they leave no traces on the disk. Their detection is particularly challenging because they do not make visible changes to the operating system.

## 2.2 Persistence techniques

Attackers install rootkits on compromised systems to maintain the privileged access they obtained, even after the system is updated and the original vulnerability that was used to gain the access is patched. A rootkit has gained persistence, when it is able to survive system reboots, and continue operation. Linux rootkits are generally implemented as either user space programs or Linux kernel modules (LKMs). Both require to be started or loaded after boot to operate.

There are several system programs on Linux that read and execute shell commands from files, for example, service managers and job schedulers, which could be used to start a rootkit process or load a kernel module. Service managers, like `init` or `systemd` execute shell commands directly after system boot. OSOM and EnyeLKM[12] are classic examples of this technique. Both rootkits modify run commands (rc) files such as `/etc/rc.local` or the files in `/etc/rc.d`, which are files `init` will read and execute shell commands from directly after boot, and load their kernel modules this way. Time-based job schedulers, for example, `cron`, `anacron`, or `at`, can be used for the same purpose. Cron parses crontab files from predefined directories each minute, gathers jobs that are due for execution, and executes the shell commands associated with these job entries. Another similar technique is inserting commands into user-specific startup files. An example for such a file is `.bashrc`, which runs certain shell commands whenever the user starts a bash session. BROOTKIT[13] implements all of the above mentioned techniques.

[12] `https://github.com/David-Reguera-Garcia-Dreg/enyelkm`, Last visited: 25.11.2020.
[13] `https://github.com/cloudsec/brootkit`, Last visited: 25.11.2020.

DocuSign Envelope ID: E4E568A1-8978-433E-A15E-38A89FE5B794

Kernel modules can be loaded during boot, without the use of shell commands, by placing them into certain directories and including their names in certain configuration files. The exact methods of loading kernel modules this way can differ between Linux distributions, which is something that the rootkits using this technique need to be prepared for. On Debian based distributions this could include but is not limited to adding the kernel module to `/etc/modules`, or placing a `.conf` file within `/etc/modules-load.d/`. On Red Hat based distributions this could include, but is not limited to placing a `.modules` executable script within `/etc/sysconfig/modules/`, adding the kernel module to `/etc/modules.conf`, or placing a `.conf` file within `/etc/modules-load.d/`. Distributions using `systemd` can also load kernel modules by placing a `.conf` file within `/etc/modprobe.d/`. Drovorub[14] and rkduck[15] are both rootkits that use this technique for loading their kernel modules.

We have already touched on `LD_PRELOAD` hijacking as a stealth technique in Section 2.1, but did not mention its uses when it comes to maintaining persistence. `LD_PRELOAD` can be set via the environment variable or `/etc/ld.so.preload` file. Attackers may set `LD_PRELOAD` to point to malicious libraries that match the names of legitimate libraries which are requested by a victim program, causing the operating system to load the malicious code upon execution of the victim program. Aside from stealth, these malicious library functions could also be used to launch an executable component of the rootkit, or load a kernel module, but most commonly they are used for connecting to remote services. BEURK, Azazel, vlany and Jynx2 all use `LD_PRELOAD` for establishing remote shell sessions.

Ramdisk based rootkits are the final category of persistent rootkits that is important to mention. Their main feature is that they are capable of activating even before the `init` process. The initial ramdisk, or `initrd`, is a compressed archive containing files needed by a Linux system early in the boot process. The `initrd` is dynamically generated on each system, so that kernel modules that are needed for a specific environment (e.g. hardware, files systems, etc.) can be included. The `initrd` is

[14] `https://media.defense.gov/2020/Aug/13/2002476465/-1/-1/0/CSA_DROVORUB_RUSSIAN_GRU_MALWARE_AUG_2020.PDF`, Last visited: 25.11.2020.

[15] `https://github.com/QuokkaLight/rkduck`, Last visited: 25.11.2020.

DocuSign Envelope ID: E4E568A1-8978-433E-A15E-38A89FE5B794

loaded into memory, alongside the compressed kernel image, by the bootloader (e.g. GRUB or Gummiboot). At boot, the kernel runs the init script from the initrd, which in turn executes the run-init binary from the initrd to start the initial user-mode process of the system. Horse Pill [17] is a proof-of-concept ramdisk based containerizing rootkit, that infects klibc, a minimalistic C library used in the early user space to start the init process. Horse Pill replaces run-init, gaining control over the system at boot time. It places the whole system into a newly created Linux kernel namespace, places a backdoor outside this namespace, and fakes some newly created processes to appear as kernel threads to make the system appear normal from within. Horse Pill can be detected by inspecting the fake kernel threads, since they are actually renamed user space processes, by auditing the namespace links in the /proc filesystem, or by performing regular integrity checks on the initrd.

## 2.3 Hiding persistent components

In general, gaining persistence requires writing some form of data on the persistent storage (i.e. the hard disk or solid state drive), the exception being the previously mentioned *Type 4* rootkits, which may, for example, reside in SPI flash ROM chips [10, 11]. These pieces of data are what we refer to as persistent rootkit components, and we have already listed several examples of them in Section 2.2. In addition to those examples, anything that allows backdoor access for the attacker (e.g. authorized ssh keys, newly created user accounts, etc.) and is stored on the disk can be considered a persistent component.

Early user space rootkits relied on obscure or deceptive naming and placement to avoid detection, but this is insufficient for avoiding any kind of rootkit detection software. Modern rootkits can easily hide their persistent components from the output of different system programs using hooking. User space rootkits using LD_PRELOAD hooking, for example, the previously mentioned Azazel and BEURK, hijack POSIX library functions such as readdir(3), opendir(3), open(3), or stat(3). When a hooked library function is called, they check the caller's shell environment variables, and unless the call came from their own shell, they remove their components

DocuSign Envelope ID: E4E568A1-8978-433E-A15E-38A89FE5B794

from the output of these functions. LKM rootkits that use kernel space hooking can hook the underlying system calls of library functions directly. OSOM hooks the `getdents(2)`, `open(2)` and `read(2)` system calls, and removes its kernel module, "osom.ko", from the output of these functions, or acts as if the file did not exist.

Another type of hooking technique that can be used for hiding files is Virtual File System (VFS) function pointer hooking. VFS is the software layer in the kernel that provides the file system interface to user space programs. It also provides an abstraction within the kernel which allows different file system implementations to coexist. VFS uses four different data structures to represent the file systems, as shown on Figure 2.1. File objects represent open files associated with a process. The superblock structure represents a mounted partition and stores metadata about the partition itself. An inode is the physical representation of a file or a directory stored on the device. An inode can be used by one or more directory entries, or dentries for short. For example, if we create a new file and a hard link pointing to it, then we will have one inode and two dentires referencing the inode.

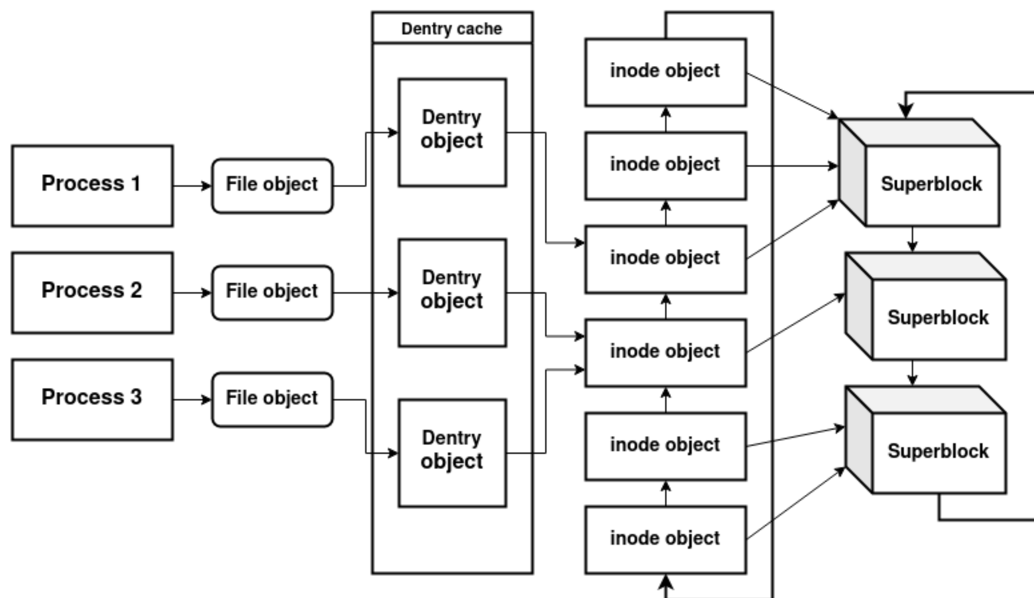


**Figure 2.1:** Components of the Linux Virtual File System (VFS).

At code level, VFS uses so called operation structures which contain function pointers. These operation structures appear in all 4 of the previously mentioned structures. The function pointers, like `*lookup`, `*create`, or `*rename` point to functions

DocuSign Envelope ID: E4E568A1-8978-433E-A15E-38A89FE5B794

of the underlying file system drivers. By hooking certain function pointers, rootkits can hide certain files or directories. For example, Liinux[16] hooks the `*lookup` pointer of the inode corresponding to the `/proc` directory, and excludes its own process id whenever process information is retrieved from `/proc`. The same technique can be applied for hiding any other file or directory. VFS operation structure hooks can be detected by examining the function pointers, since normally they should be pointing inside the kernel's code segment.

## 2.4 Rootkit detection

Rootkit detection methods can be divided into multiple categories [28]. *Signature-based* methods compare hashes of files or memory patterns to a database of signatures associated with known malware. Signatures cannot be directly used to discover new rootkits, but code reuse is very common in all types of malware, which means that certain characteristic byte patterns can appear even in undiscovered rootkits [1, 22]. The classification of such byte patterns using machine learning models has been widely studied and deployed as integral components of anti-malware systems [27].

*Behavioral/Heuristic* detection methods detect deviations from "normal" patterns of system behavior. Discrepancies in resource usage and timing can be used to detect virtual-machine based rootkits (VMBRs), which install a virtual-machine monitor underneath an existing operating system and place the main OS into a virtual machine [16, 14]. Network level traffic analysis can be used to detect the presence of even advanced rootkits. Information stealing rootkits, like Duqu and Flame connect to their command and control (C&C) server to send the gathered data from the system [4]. This can create periodic, anomalous network traffic, for example, when the rootkit is sending out screen captures from the infected system. This technique is especially useful in detecting rootkits that turn the hosts into parts of a botnet, or rootkits that are able to hide network connections from the host. The disadvantage of this technique is that it requires very complex implementations,

[16] `https://github.com/a7vinx/liinux`, Last visited: 26.11.2020.

DocuSign Envelope ID: E4E568A1-8978-433E-A15E-38A89FE5B794

and it generally has a high false-positive rate [26], but machine learning techniques were shown to be able to reduce the false-positive rate, and improve accuracy and performance [18].

*Cross-view-based* methods observe the same aspect of a system in multiple ways, with the assumption that there is no perfect rootkit which can perfectly emulate all aspects of the system. They enumerate OS objects (e.g. files, processes, open ports, etc.) in multiple, independent ways, and compare the results to each other. Depending on the implementation, cross-view-based methods may or may not assume a trusted kernel, and therefore may even be applied to detect DKOM. However, there are a limited number of API and system calls for requesting certain information, and the detection software may need to have its own implementation to be able to make an adequate number of comparisons. Also, the observed structures are often dynamically changing, which can lead to false-positive results. This can be circumvented by preventing certain operations and prohibiting the start of new processes, but this approach has an impact on system performance.

*Integrity-based* detection methods are based on invariant specification, which means that they define aspects of the system that should not change during normal behaviour, and periodically monitor these aspects, comparing them to a secure baseline. In the case of files, the trusted baseline can be the hash value of a file computed in a controlled environment. Kernel data structures (e.g. the system call table, the list of running processes, etc.) can also be monitored for malicious changes. This technique can also be used for performing integrity checks on running processes, but exploit code placed on the process memory's dynamic segments, such as the stack or the heap, and return-oriented rootkits [12] cannot be detected this way. The main challenges of integrity-based detection are the proper definition of invariant parameters and the secure storage of the baseline. Since the kernel is assumed to be compromised, no file or memory on the system can be considered secure, and no system call return value can be considered accurate. For this reason, this method is most effective for rootkit detection when the trustworthiness of the kernel can be ensured.

DocuSign Envelope ID: E4E568A1-8978-433E-A15E-38A89FE5B794

## 2.5 Existing solutions

Several different rootkit detection solutions are available for Linux systems, both open-source and licensed. In this section we name a few examples, ones that use different approaches, and evaluate their abilities and shortcomings.

*Chkrootkit*[17] is a signature-based, open-source rootkit detection software implemented as a shell script, with certain functions written in C. It can detect a fixed set of rootkits, since it implements the checks for each one separately. The current version supports 71 different rootkits and worms. The checks for user space rootkits are based on finding their characteristic components on the file system, for example, when checking for the SucKIT rootkit, it searches for the strings `.sniffer` and `FUCK` in `sbin/init`. It is also able to detect certain LKM rootkits that hide their processes and files, by applying some cross-view-based techniques. It can detect processes hidden from the output of `ps(1)` by attempting to open every possible sub-directory in `/proc` directly, where the sub-directory names are the possible Linux process ids, from 1 to 99999. It can also detect directories hidden from `readdir(3)` by invoking `lstat(3)` and comparing the number of links to the directory with the number of sub-directories returned by `readdir(3)`. Overall, *Chkrootkit* is a useful tool for detecting infections by the most well-known and well-documented rootkits and worms, but it lacks the ability to detect modified or new malware. Being a simple user space process, it is also defenseless against attacks specifically targeting detection software. *Rootkit Hunter*, also known as *rkhunter*[18], is another popular signature-based detection software, but due to its similarity to *Chkrootkit*, we will not provide further information about it.

*Lynis*[19] is a security auditing and hardening tool, not directly a rootkit scanner. It can perform a system audit that can reveal vulnerable configurations and anomalous behaviour. The audit extends to the file systems, kernel, firewall, networking settings, file permissions, and many more. While not designed to find rootkits present

[17] `https://github.com/Magentron/chkrootkit`, Last visited: 27.11.2020.
[18] `http://rkhunter.sourceforge.net/`, Last visited: 27.11.2020.
[19] `https://github.com/CISOfy/lynis`, Last visited: 27.11.2020.

DocuSign Envelope ID: E4E568A1-8978-433E-A15E-38A89FE5B794

on the system, the regular audits using *Lynis* can potentially have a bigger impact on the security of the system than a rootkit scanner.

*St. Michael*[20], also known as *St. Jude*, is an integrity-based rootkit detection tool implemented as a LKM. It provides protection against LKM rootkits by monitoring various portions of the kernel for modifications, and can backup and restore a copy of the kernel, in case a catastrophic kernel compromise is detected. It is able to generate and check the MD5, and optionally SHA1, checksums of various kernel data structures, such as the system call table, and file system call out structures. It can also generate and check the MD5 checksum of the kernel code segment. *St. Michael* noticeably shares some features with rootkits, such as hiding itself from the list of loaded kernel modules, and loading through `initrd`, but such techniques are warranted when real rootkits have unrestricted privileges on the system and employ more and more advanced methods to stay hidden. Unfortunately, just like any other host-based intrusion detection software (HIDS), *St. Michael* is unable to defend itself against targeted attacks, and a rootkit that took the presence of detection software on the target into consideration could easily disable it after gaining privileged access. Examples of non-LKM HIDS software with a similar feature set to *St. Michael* are *Samhain*[21] and *OSSEC HIDS*[22].

[20] `https://sourceforge.net/projects/stjude/`, Last visited: 27.11.2020.
[21] `https://la-samhna.de/samhain/`, Last visited: 27.11.2020.
[22] `https://github.com/ossec/ossec-hids`, Last visited: 27.11.2020.

DocuSign Envelope ID: E4E568A1-8978-433E-A15E-38A89FE5B794

# Chapter 3

# Technical background

As we have shown in Chapter 2, reliable detection of rootkits is not possible when the detection software runs at the same or lower privilege level than the rootkits themselves. A well-designed rootkit that has successfully compromised the kernel can return a false view of the memory, making detection attempts that rely on kernel-provided information ineffective, and even LKM-based security solutions are vulnerable to targeted attacks. In response to this issue, a number of detection methods were implemented, that completely circumvent the need for a trusted kernel.

## 3.1 Addressing the fundamental issues of rootkit detection

Rather than relying on the kernel to provide a correct view of memory, *hardware-based* solutions can use direct memory access (DMA) to read and analyze memory data on a separate system. *Copilot* [20] uses DMA via the PCI bus and uses a co-processor to perform fast hashes over static kernel memory, and reports violations to a supervisory machine. This technique can subverts any rootkit's ability to change the view of memory, with the exception of a rootkit implemented in the hardware itself. The main limitation of this technique is that DMA approaches cannot acquire locks on kernel data structures, so race conditions resulting in false-positive detection

DocuSign Envelope ID: E4E568A1-8978-433E-A15E-38A89FE5B794

results can arise when the kernel is updating a data structure during a DMA read. This problem was also documented with user space rootkit detection software when using asynchronous page fetching [3].

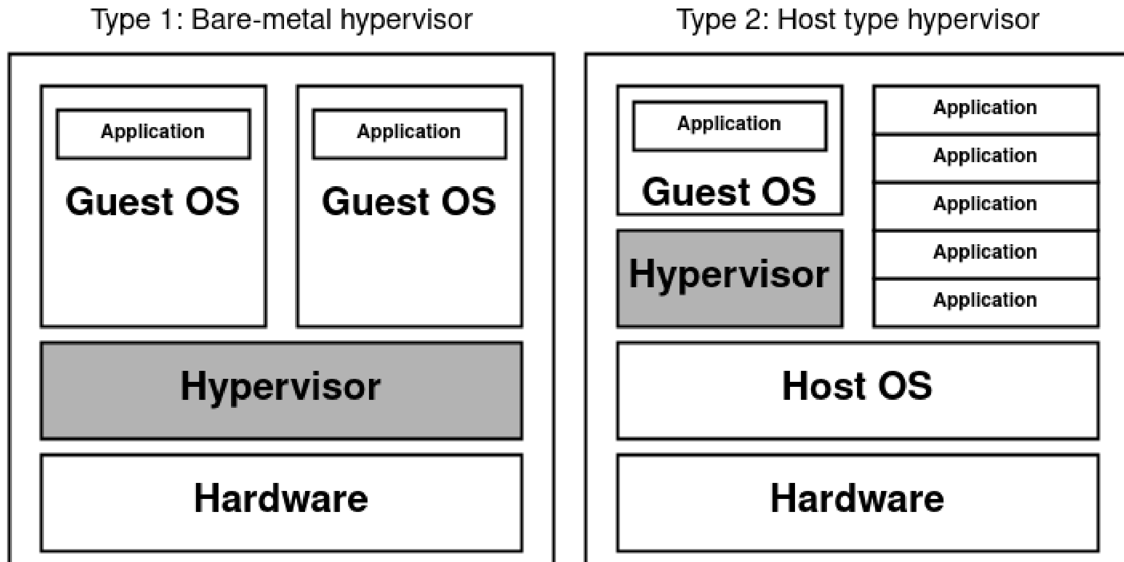


**Figure 3.1:** The two types of hypervisors.

*Virtualization-based* approaches involve the virtual machine monitor (VMM or hypervisor) in the inspection of system resources. The hypervisor operates at a higher privilege level than the guest OS, either directly on the hardware (bare-metal hypervisor), or simulated in software (host type hypervisor), and manages the resource allocation between the guest systems. Since the hypervisor holds control over the hardware resources, it can use DMA to read data from the kernel memory in a synchronous way [8], and monitor the static or dynamic kernel objects of the guest OS. *Hooksafe* [29] is a tool, based on the Xen hypervisor, for detecting kernel hooks. It gathers kernel hooks (system call table pointers, inode operation function pointers, etc.) into a dedicated memory space, and directs all read or write access to these protected hooks through a monitored indirection layer. One of the issues with virtualization or using hypervisors is the performance overhead involved in the virtualization process, especially in the case of resource-constrained embedded IoT devices. The other issue is that hypervisors themselves may be vulnerable to a targeted attack, if there is a bug in the hypervisor that can be exploited from the guest system [5].

DocuSign Envelope ID: E4E568A1-8978-433E-A15E-38A89FE5B794

For reliable detection of rootkits on IoT devices, our proposed solution leverages a Trusted Execution Environment (TEE) to protect the detection software from targeted attacks through hardware level isolation, and to eliminate the need for a trusted kernel. Our software incorporates ideas from both signature-, cross-view-, and integrity-based detection methods, and uses the trusted storage functionality of the TEE to store baseline values (e.g. file hashes and memory image signatures) in a secure way.

## 3.2 Trusted Execution Environments

*Trusted Computing* can be considered one of the predecessors of TEEs. It was developed and promoted by the Trusted Computing Group, and was aimed to to achieve secure computation, privacy and protection of data through the use of hardware components known as Trusted Platform Modules (TPMs)[1]. TPMs can generate cryptographic keys securely, store them in a separate, tamper-evident hardware module, and perform regular integrity checks, detecting changes to hardware or software components. TPMs raised many privacy concerns because of their use of remote software validation and general lack of transparency[2], and they were also unable to satisfy the need for a local, secure, isolated execution environment, since their functionality was reduced to the predefined set of APIs provided by the manufacturer.

A new approach to address trusted computing is to allow the execution of arbitrary code within a confined environment, that can provide tamper-resistant execution to applications, without the need for additional special purpose hardware. As defined by Global Platform[3], a Trusted Execution Environment is a secure area of the main processor of a connected device that ensures sensitive data is stored, processed and protected in an isolated and trusted environment. In practice, TEEs enable the coexistence of separate systems with different levels of security on the same

[1] `https://trustedcomputinggroup.org/work-groups/trusted-platform-module/`, Last visited: 28.11.2020.

[2] `https://www.gnu.org/philosophy/can-you-trust.html`, Last visited: 28.11.2020.

[3] `https://globalplatform.org/wp-content/uploads/2018/05/Introduction-to-Trusted-Execution-Environment-15May2018.pdf`, Last visited: 28.11.2020.

DocuSign Envelope ID: E4E568A1-8978-433E-A15E-38A89FE5B794

platform. Basically, it divides the system into multiple partitions, and guarantees a strong isolation between them, except for the carefully controlled interface for inter-partition communication. These isolated system partitions possess their own separate kernel and operating system, that may have different levels of access to system resources.

TEE implementations have to fulfill a certain set of basic design criteria in order to be considered secure:

- They must implement a *Secure Boot* protocol, which can validate code integrity during the bootstrapping process.
- Scheduling between separate systems must be efficient, assuring that the tasks running in the TEE do not affect the responsiveness of the main OS.
- The communication between the TEE and the rest of the system must be protected, reliable, and work with a minimal overhead.
- The authenticity, and optionally confidentiality of I/O communication must be ensured.
- There must exist a *Secure Storage*, where the confidentiality, integrity, and freshness of stored data is guaranteed, and only authorized entities can access data.

### 3.2.1 ARM TrustZone

*TrustZone* [19, 21] is hardware and virtualization based implementation of the TEE design principles for ARM Cortex-A and Cortex-M processors. It defines two separate system partitions: the secure world (TEE) and the normal world, which is often referred to as the Rich Execution Environment (REE). The secure world possesses unrestricted access to system resources, while the normal world is restricted by hardware barriers from accessing the secure portions of the physical memory. The two worlds may house separate kernels and separate operating systems. Secure boot is implemented using ARM's own Secure Boot Sequence, which cryptographi-

DocuSign Envelope ID: E4E568A1-8978-433E-A15E-38A89FE5B794

cally verifies each step of the boot process, with the unique verification keys stored in One-Time Programmable memory (OTP-ROM).

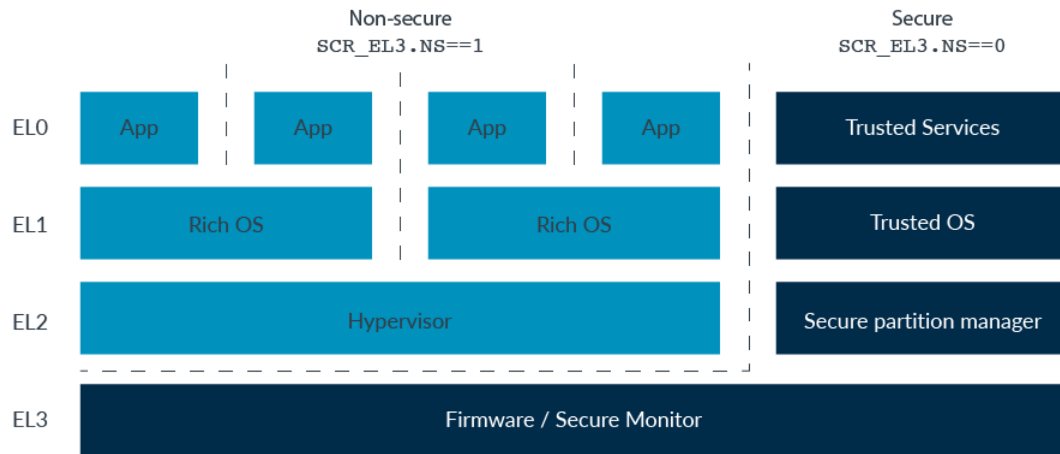


**Figure 3.2:** The arrangement of security states and exception levels in ARMv8-A architecture processors.[4]

At any moment, the processors of an ARM TrustZone enabled CPU may either be in secure state, executing secure world code, or non-secure state, executing normal world code. The current state of a processor is indicated in the Secure Configuration Register's (SCR) 0th, "Non-Secure" (NS) bit. Communication between the two worlds, processor context switching, and the setting/clearing of the NS bit is managed by the Secure Monitor, as shown in Figure 3.2. Context switching is invoked through a Secure Monitor Call (SMC) or an interrupt (IRQ or FIQ). With these mechanisms, it is possible run a trusted program inside the secure world without the need to trust the integrity of the normal world kernel, since attempts to access the secure world address space will result in access violations, that will be trapped by the secure monitor.

[4] `https://developer.arm.com/architectures/learn-the-architecture/trustzone-for-aarch64/trustzone-in-the-processor`, Last visited: 28.11.2020.

DocuSign Envelope ID: E4E568A1-8978-433E-A15E-38A89FE5B794

### 3.2.2 OP-TEE

*OP-TEE*[5] stands for Open Portable Trusted Execution Environment. It was originally developed by ST-Ericsson, but is now an open-source project hosted by Linaro[6]. OP-TEE is based on ARM TrustZone, but implements the TEE Internal Core API[7] and TEE Client API[8] defined by Global Platform, which were aimed to standardize TEE functionality, making OP-TEE potentially compatible with other future technologies.

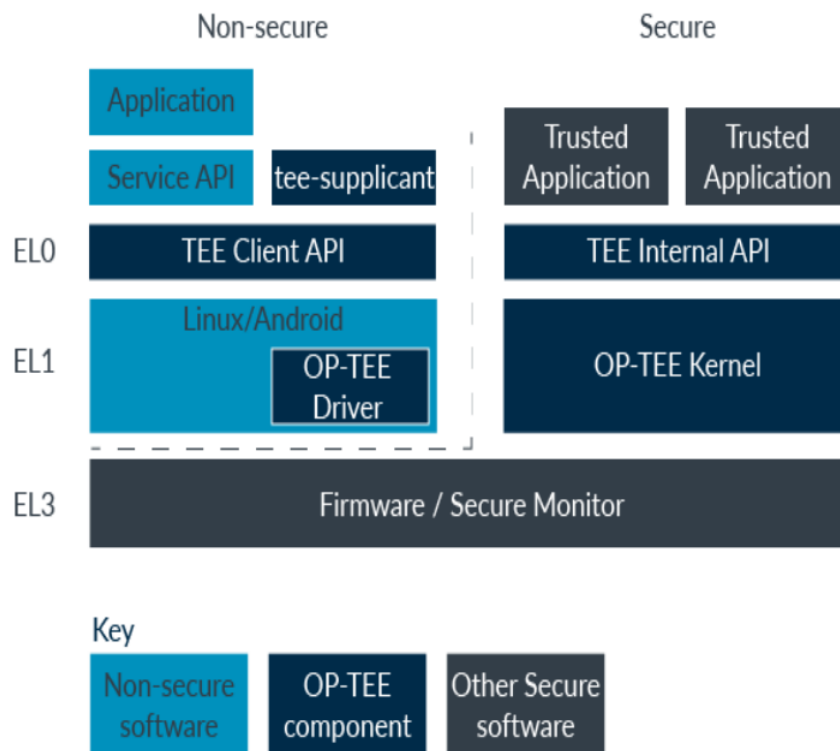


**Figure 3.3:** The structure of OP-TEE.[9]

OP-TEE has components in both the normal world and the secure world, as shown in Figure 3.3. There are four components to OP-TEE:

[5] https://www.op-tee.org/, Last visited: 28.11.2020.
[6] https://github.com/OP-TEE/optee_os, Last visited: 28.11.2020.
[7] https://globalplatform.org/wp-content/uploads/2018/06/GPD_TEE_Internal_Core_API_Specification_v1.1.2.50_PublicReview.pdf, Last visited: 28.11.2020.
[8] https://globalplatform.org/wp-content/uploads/2010/07/TEE_Client_API_Specification-V1.0.pdf, Last visited: 28.11.2020.

DocuSign Envelope ID: E4E568A1-8978-433E-A15E-38A89FE5B794

- The **TEE Client API** is what Client Applications (CAs) use to invoke a Trusted Application (TA).
- The **OP-TEE driver** is integrated into Linaro's own version of the Linux kernel, an is responsible for shared memory allocation between the two worlds, and provides the Remote Procedure Call (RPC) functionality through which the CA and TA can communicate.
- The **tee-supplicant** is a user space daemon, that handles services that are supported by OP-TEE and require some level of normal world OS interaction. It communicates with the OP-TEE driver through a bidirectional channel, meaning that it can both send and receive requests for an action.
- In the secure world, OP-TEE implements its own, minimal **OS**, which communicates with TAs through the TEE internal API.

There are two ways to implement TAs: user mode TAs and pseudo TAs. User mode TAs are loaded by OP-TEE in the secure world when a CA invokes it from the normal world using their universally unique identifier (UUID). They run at a lower privilege level than OP-TEE core code, just like how normal world user space applications run at a lower privilege level than the kernel.

Pseudo TAs (PTAs) are used to extend the functionality of the OP-TEE kernel, just like kernel modules in the normal world. These applications must be compiled into the OP-TEE OS and they are capable of exposing core functionality to TAs or CAs.

OP-TEE was chosen as the basis for our TEE-based persistent rootkit detection software due to its open source nature and continuously expanding feature set. Since OP-TEE OS is based on Linux, and the core is designed around the ARM TrustZone technology, it is an ideal test-bench for the development of software intended to run on embedded IoT devices.

[9] `https://developer.arm.com/architectures/learn-the-architecture/trustzone-for-aarch64/software-architecture`, Last visited: 28.11.2020.

DocuSign Envelope ID: E4E568A1-8978-433E-A15E-38A89FE5B794

# Chapter 4

# The scope of our work

As mentioned previously, our approach to the issue of reliable detection of persistent rootkits is to run the detection software in a Trusted Execution Environment (TEE), specifically, in OP-TEE. By using OP-TEE, our detection software is protected from targeted attacks originating from the normal world, however, all efforts must be made to avoid introducing vulnerabilities into the detection software itself.

We can create a snapshot of the Rich Execution Environment (REE) OS memory when execution is transferred to our Trusted Application (TA), and perform integrity and consistency checks on the kernel code and the kernel data structures. In order to make the detection process complete, we have to account for a rootkit that tries to evade our checks by removing its traces from memory before the invocation of our rootkit detection function, and hides itself on the persistent storage in such a way that it can execute again later when the memory has been checked. This can be done by performing File Integrity Monitoring (FIM) on chosen files and directories, the list of which designed to counter the persistence techniques, which were discussed in Section 2.2. The main focus of our work is to create a secure and reliable implementation of FIM, as part of a Trusted Application designed to detect rootkits.

Unfortunately, OP-TEE does not provide a way to access the REE file system by default. While we managed to implement this functionality through a PTA and the tee-supplicant, reliance on components of the REE OS could not be entirely negated

DocuSign Envelope ID: E4E568A1-8978-433E-A15E-38A89FE5B794

for file system access. In our case, this is not an issue, since we can assume the kernel to be reliable after the memory checks are complete. What has to be addressed, is that if a rootkit stored itself on the file system to evade the memory checks, it should not be allowed to activate and clean itself from the persistent storage before we finish our scan of the file systems.

While the detection of stealth rootkits in the memory is not the main topic of this thesis, it is necessary to understand the capabilities and limitations of the larger detection software in order to contextualize the functionalities of the FIM component, to understand the interactions between the software modules, and to be able to design the required protection measures.

When invoked, the TA performs a series of verification steps aiming at detecting inconsistencies in the data held by certain kernel data structures or modifications of kernel code. These checks counter the most common techniques employed by rootkits for the purpose of process hiding. The following checks are performed:

- The integrity of the system call table, kernel text segment, and system programs currently executing is checked through hash comparison.
- The kernel data structures, which could be modified through DKOM to achieve process hiding, are checked in a cross-view-based way.
- The integrity of the normal world components that are required for the FIM process, such as the tee-supplicant, is checked through text segment hash comparison.
- The operation structures in the Linux Virtual File Systems (VFS) are checked for modified function pointers. Since these checks have to be performed recursively, they can become very time consuming when applied for multiple directories. For this reason, only the operation structures relating to */proc* are checked, which is enough to counter this method of process hiding.

Since only the VFS objects that are available from the superblock of the */proc* file system are checked, file hiding is still possible through the modification of the operation structure function pointers. In practice, this is only an issue if the rootkit

DocuSign Envelope ID: E4E568A1-8978-433E-A15E-38A89FE5B794

can avoid all other checks, while keeping the malicious function hook in the memory. The addressing of this issue is outside the scope of this thesis.

DocuSign Envelope ID: E4E568A1-8978-433E-A15E-38A89FE5B794

# Chapter 5

# Design

## 5.1 The architecture of the detection software

Our rootkit detection solution has two main software components: a Trusted Application (TA) running in the TEE and a Client Application (CA) running as a user space process on top of the normal world OS in the REE. Our CA should be started when the system is booted and then it should run continuously. The main role of the CA is to invoke the TA periodically and to pass certain data to the TA collected from the REE (e.g. the list of running processes, as seen by the `ps(1)` program when executed on the normal world OS). The TA performs rootkit detection by executing different integrity and consistency verification functions. In order to ensure that the TA is indeed invoked periodically, a watchdog timer can be started during the boot process that can only be reset by the TA; therefore, if the TA is not invoked, the timer expires and the device reboots itself. When the TA finishes its execution, control is returned to the CA, which runs concurrently with other applications and services in the REE.

### 5.1.1 The client application

Figure 5.1 gives an overview of the tasks performed by the CA, and its interactions with the TA. As it can be seen, the CA is started at boot time, during the `init` process. It runs continuously, and invokes the TA at random time intervals.

DocuSign Envelope ID: E4E568A1-8978-433E-A15E-38A89FE5B794

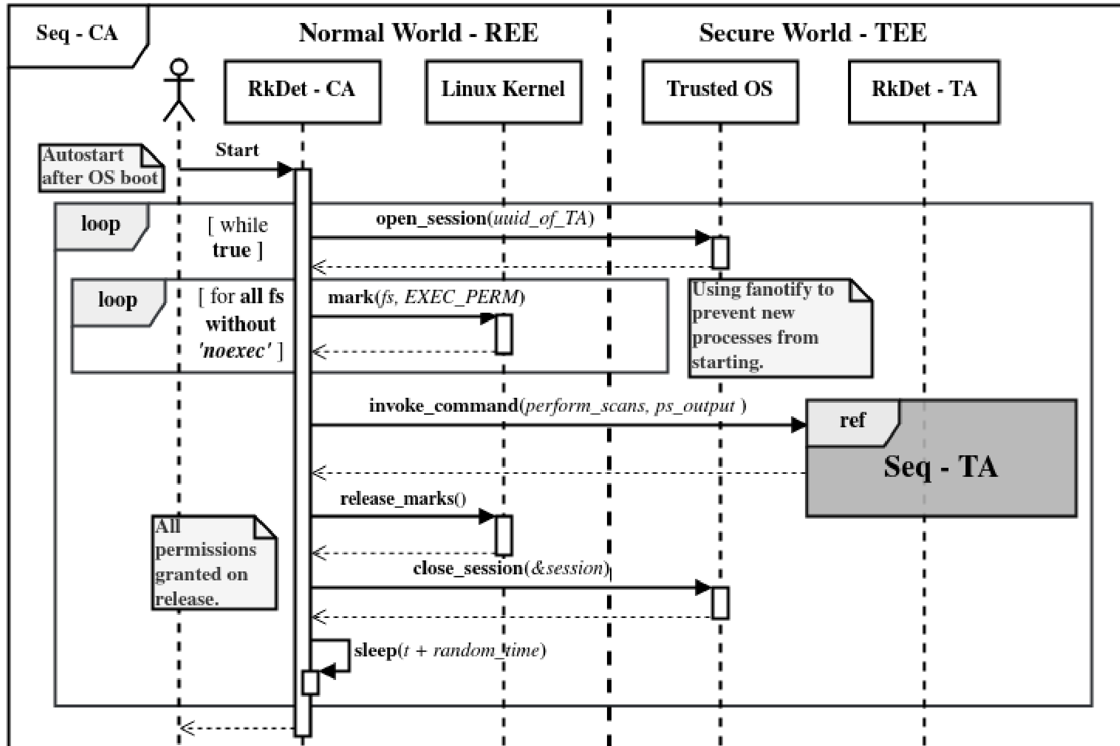


**Figure 5.1:** The functionality of the client application.

The CA establishes the communication channel with the TA by requesting the TEE to open a session between the two applications. During this process, the CA refers to the TA by its universally unique identifier (UUID).

Before invoking the TA, the CA disables execution type access to files. Consequently, new programs cannot be started until the rootkit detection function finishes, and the control is transferred back to the CA, but processes that were started before that point can continue to run normally. We provide a more detailed description of this functionality in Sections 5.4 and 6.5.

After the checks are finished, the CA re-enables execution, closes the session with the TA, and sleeps for a randomized length of time before invoking the detection function again.

### 5.1.2 The trusted application

After the detection function of the TA is invoked, it performs a series of consistency and integrity checks on the kernel data structures, the text segment of the kernel,

DocuSign Envelope ID: E4E568A1-8978-433E-A15E-38A89FE5B794

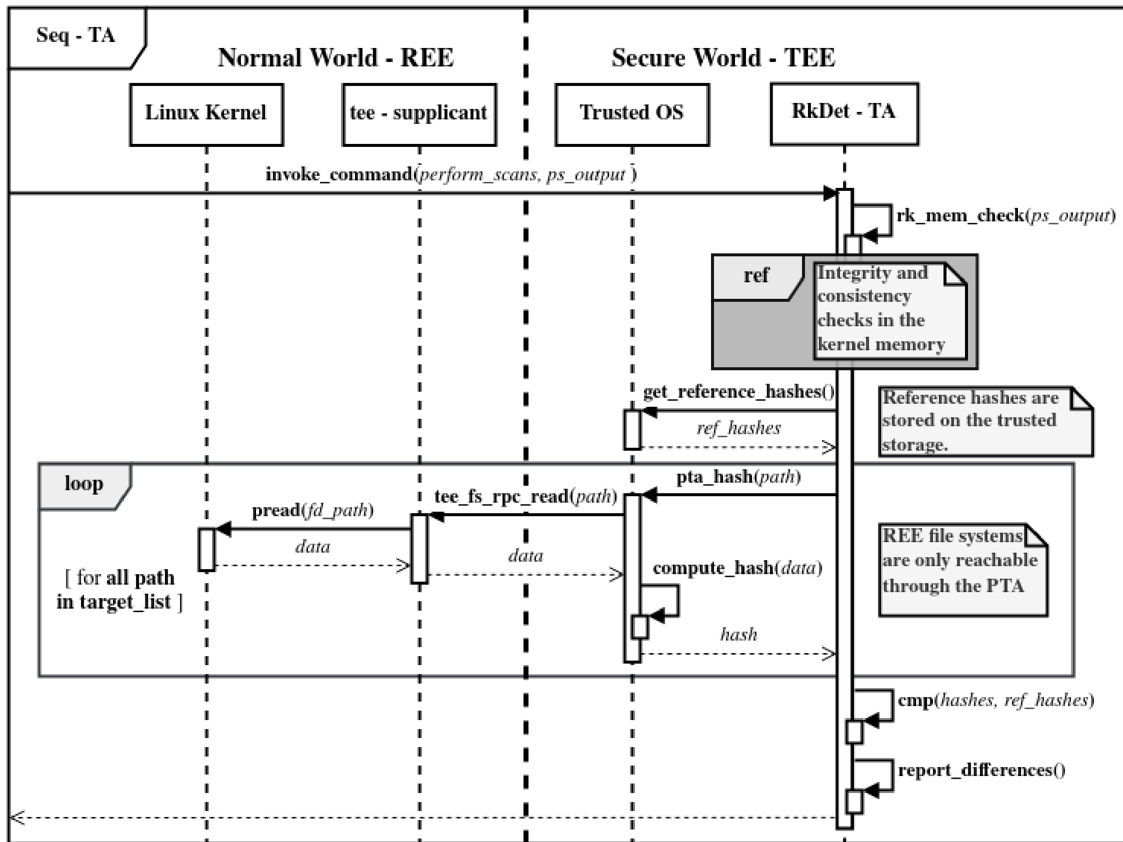


**Figure 5.2:** The functionality of the trusted application.

the running system programs, the tee-supplicant, and our CA. It is important to note, that we compiled the tee-supplicant and our CA as static binaries, so all code they use during execution can be found in their text segment, and they do not dynamically load any libraries.

If integrity violations or inconsistencies are found, then they are reported to the operator of the device via a remote attestation protocol, but this is outside of the scope of this thesis.

If the verification of the memory is successful, then the TA can proceed with the integrity checks on the files systems, since at that point the tee-supplicant, and the normal world kernel is assumed to be clean. The tee-supplicant uses the functions of the REE kernel, which is why the integrity of both of these components has to be verified. Before starting the process itself, the TA fetches the baseline hashes from OP-TEE's Secure Storage. The baseline is created during the initialization process, which we discuss in Section 6.4.

DocuSign Envelope ID: E4E568A1-8978-433E-A15E-38A89FE5B794

After the baseline has been successfully retrieved, the TA proceeds with calculating the SHA-256 hashes for the predefined list of target directories and files. The hashing itself is done by our PTA, which accesses the contents of the REE file systems through the tee-supplicant. We provide more information about this process in Section 6.3. Once all hashes have been calculated, the TA compares them to their baseline values, reports all integrity violations through the previously mentioned remote attestation protocol, and returns control to the CA.

## 5.2 Possible approaches to File Integrity Monitoring

As mentioned previously, one of the main challenges of File Integrity Monitoring (FIM) lies in keeping the false positive detection rate to a minimum, while providing ample coverage for the files and directories at risk. Correctly choosing the list of monitored directories and files is crucial in addressing this challenge. There are two possible approaches to constructing this list. The static approach is to construct the list manually, according to our knowledge of possible rootkit techniques and the system itself. The dynamic approach is to gather information about the state of the system, and construct the list accordingly before each scan. In our case, the goal of both approaches is to cover all the binaries and scripts that could legitimately execute on the device. The reason behind this is that potential rootkits are assumed to have been forced to unload from the memory at this point, and can only reinfect the system if a program or user executes their component from the persistent storage. For our implementation, we chose to use a static, predefined list, but we experimented with the possibilities of the dynamic approach too.

### 5.2.1 The dynamic approach

There are a limited number of ways the previously mentioned rootkit components could reinfect the system, especially in the case of IoT devices, where user activity is rare or nonexistent. We already provided a list of the possible methods to reinfect

DocuSign Envelope ID: E4E568A1-8978-433E-A15E-38A89FE5B794

the system with the use of system programs in Section 2.2. The discussed system programs, like cron, execute commands from a fixed directory, file, or set of files. For this reason, we could limit our list of monitored files and directories to the potential locations of these command files. Unfortunately, this list would give us incomplete coverage, because the referenced legitimate programs could have been modified, so they all would have to be monitored too.

We created a prototype solution where we experimented with parsing the crontab files of the system. Our goal was to extract all file references from the scheduled shell commands, so we could dynamically modify the list of target entries of our FIM component. We used a static analysis approach based on the source code of the bash command parser and cron's crontab entry parser.

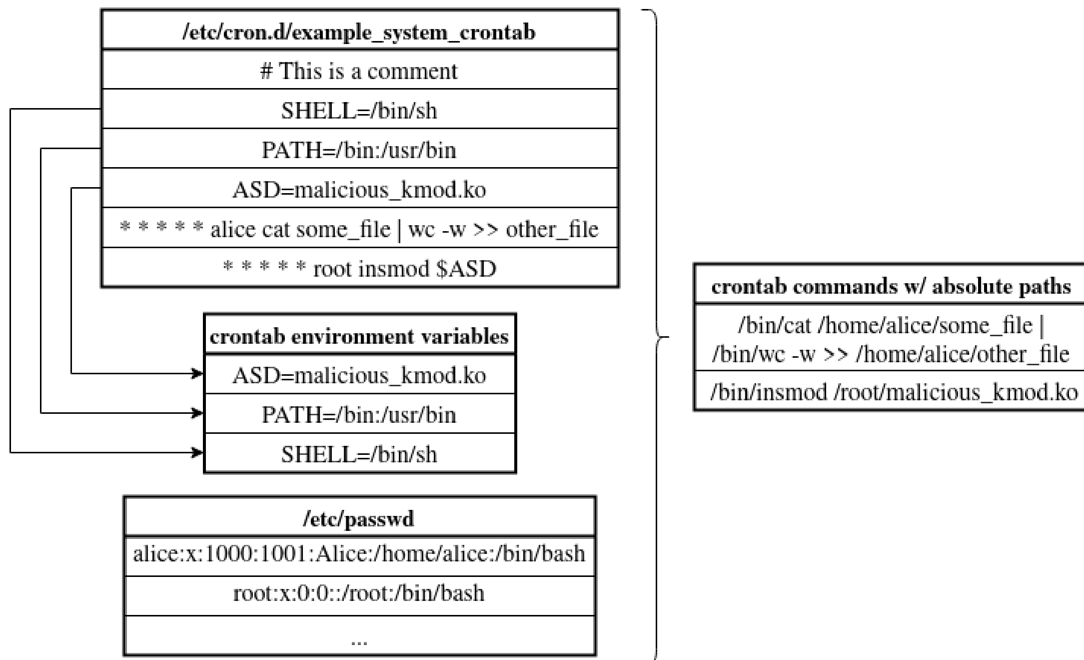


**Figure 5.3:** The extraction of shell commands from crontab files, and resolution of non-absolute paths.

The crontab format defines two types of valid lines. One is used to define an environment variable, the other, to schedule a job. We managed to extract all environment variables and all scheduled commands from the crontab files, using regular expressions, which we implemented as state machines, since no regex libraries were available in the TEE. The PATH and HOME environment variables are needed for

DocuSign Envelope ID: E4E568A1-8978-433E-A15E-38A89FE5B794

resolving non-absolute file paths, and the others could be referenced in any of the job entries.

Crontab files can belong to individual users, or the system itself. The entry format is slightly different between the two types of crontab files. System crontabs have a mandatory username field, and execute jobs as the specified user, while user crontabs have no username field, and execute jobs with the privileges of the their owners. This has to be accounted for, and all jobs have to be matched with their corresponding user profile and HOME directory in /etc/passwd, in order to be able to resolve non-absolute paths. Figure 5.3 shows an example of this process.

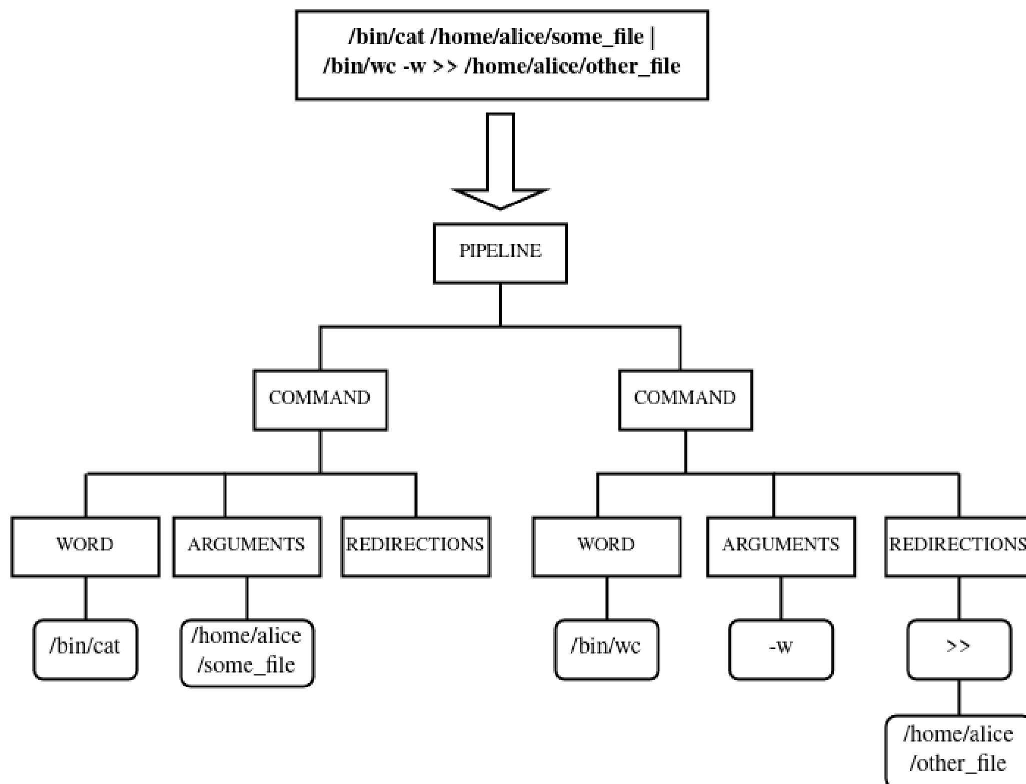


**Figure 5.4:** A simple shell command and its corresponding AST.

After we collected all environment variables and matched all entries with the correct user profiles, we could proceed to extract the shell commands from the job entries. We built an abstract syntax tree (AST) for each command in the crontab file. As shown on Figure 5.4, the AST representation helps with the tokenization of the shell commands. We expanded all valid paths into absolute paths using the PATH and

DocuSign Envelope ID: E4E568A1-8978-433E-A15E-38A89FE5B794

`HOME` variables, and extracted the possible file paths from the command's parameters (including the invoked command's own file).

However, this approach has serious limitations which may cause the target list to be incomplete. For example, if the crontab file includes a program which dynamically loads and executes another binary, our static analysis would miss that binary. What is more, our static analysis would need to be able to handle paths which are constructed dynamically (e.g. using loops). In order to overcome this challenge, either a "cron policy" is needed which limits how the users can define scheduled commands or the analysis should be extended with dynamic tools which run the commands in a controlled environment and extract the directories and files accessed during execution.

Another approach to the issue of having to monitor all legitimate programs, is to identify all executable files on the file system. However, we cannot identify such files using the `x` permission flag, because file permissions are easy to change. Recognizing ELF binaries by reading the first few bytes for the magic value is doable but the device may contain other types of executable files (e.g. different script files). Differentiating between script files for different interpreters, such as Python, Perl and Bash, and simple text files is challenging without executing them, so again, this would require dynamic analysis in a controlled environment.

### 5.2.2 The static approach

The static approach to covering all the binaries and scripts that could legitimately execute on the device is to monitor a predefined list of directories and files. A list like this will provide sufficient coverage, but it will either cause an increased run time, because it will cover binaries that are never actually executed by any system programs, or it will have to be meticulously, manually constructed for each system.

As our method is designed for embedded IoT devices, we assume a small local file system. For this reason, we can expect to perform a thorough scan in a reasonably short amount of time. Therefore, we compiled a general list of file entries to hash which consists of many of the well-known top-level Linux directories, like `/bin`, `/lib`

DocuSign Envelope ID: E4E568A1-8978-433E-A15E-38A89FE5B794

and /etc. There are directories with highly volatile contents, such as /tmp, /var and /dev, which we do not include in our scanning process.

To support this approach, we recommend a certain organization of the files on the file system. Files (including programs) that should not change should be separated from those that have variable content (e.g. files used mainly for data storage). This kind of separation is also useful for many other reasons related to the maintenance and troubleshooting of the device.

## 5.3 Secure storage of the baseline

In the previous sections we addressed one of the main challenges of FIM, namely the issue of balancing the false positive rate with the coverage of the scans, but we have not yet addressed the other main challenge, which is the secure storage of the baseline. OP-TEE provides an implementation of the Global Platform trusted storage API[1], which is referred to as Secure Storage in the OP-TEE documentation.

We use the Secure Storage to store our baseline hashes in the format shown on Figure 5.5. In this subsection, we briefly discuss a few important features of the Secure Storage that are necessary for protecting the authenticity and integrity of our reference hashes.

| /path/to/file/or/directory | \t | 32 byte SHA-256 hash | \n |
|---|---|---|---|
| /path/to/file/or/directory | \t | 32 byte SHA-256 hash | \n |
| /path/to/file/or/directory | \t | 32 byte SHA-256 hash | \0 |

**Figure 5.5:** The format of the hashes stored on the Secure Storage.

It is important to mention that according to the specification of the trusted storage API, a single, designated Trusted Storage Space is provided for each TA. As a result,

[1] https://globalplatform.org/wp-content/uploads/2018/06/GPD_TEE_Internal_Core_API_Specification_v1.1.2.50_PublicReview.pdf, Last visited: 03.12.2020.

DocuSign Envelope ID: E4E568A1-8978-433E-A15E-38A89FE5B794

our stored reference hashes are accessible only by authorized TAs running on the same device in the same TEE as in which the data was created.

The inner workings of the trusted storage are highly dependent on the TEE implementation. The trusted storage may not be entirely separated from the REE file system. Consequently, file modifications from a root privileged user account could be a valid concern. The trusted storage is expected to provide confidentiality and authenticity through the use of authenticated encryption. This protects our reference hashes against targeted modification and replay attacks. For additional protection and separation from the REE file system, the reference files can be stored on a Replay Protected Memory Block (RPMB) partition[2].

An attacker could attempt to delete the reference hashes from the trusted storage, but according to the specification, the trusted storage is protected from such attacks because a stored object should not be accessible from outside the TA that created it. How this functionality is achieved in practice is, again, highly dependent on the TEE implementation. In the case of OP-TEE, the TEE recognizes the data corruption and generates an alert.

## 5.4 Attack surface and security measures

During the design process, we discovered an architectural weakness in the rootkit detection software. What we had to consider is that while our TA is waiting for I/O operations (i.e. reading file contents) to complete, control may be given back to the REE. When this happens, pre-scheduled jobs may be executed by a job scheduler (e.g. cron). Hence, in theory, a rootkit can hide a persistent component in a new or existing program file, and schedule the execution of said program before removing itself from memory.

We would detect no integrity violations or inconsistencies in the memory, because the rootkit has made sure to remove all its traces. Then, the malicious program could be executed by the job scheduler during the file hashing operation performed by our TA. When executing, the rootkit could move itself from the program file immediately to

[2] https://lwn.net/Articles/682276/, Last visited: 03.12.2020.

DocuSign Envelope ID: E4E568A1-8978-433E-A15E-38A89FE5B794

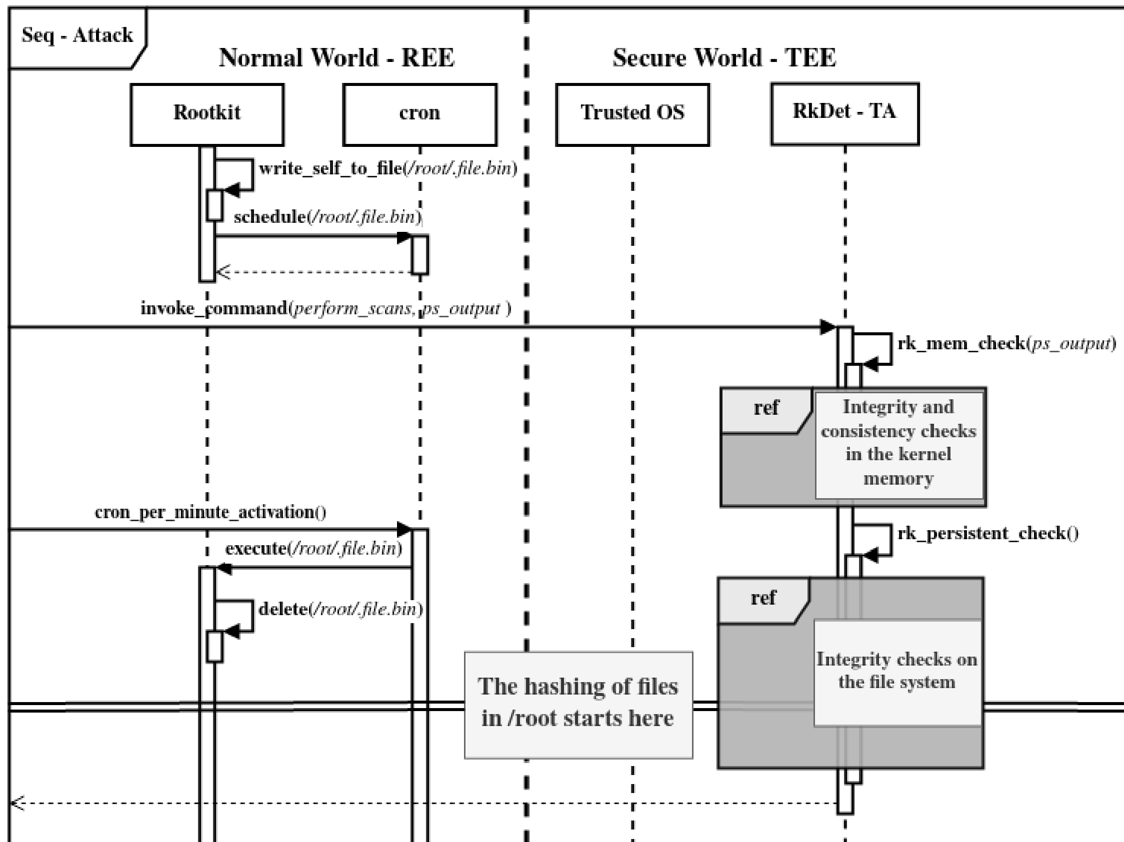


**Figure 5.6:** Cron executing the attacker's code during the timing window between the integrity checks.

memory and delete the program file before the hashing process reaches the directory containing it. Alternatively, the rootkit could move itself to a different file that has already been hashed at that point, and execute at some time in the future. In either situation, the computed hash values would match the baseline, the rootkit would be undetected, and could continue to operate. Figure 5.6 illustrates how a rootkit could avoid our checks by using this technique.

As I/O operations are usually slow, our TA is mostly waiting during the file hashing, which means that control is mainly at the normal world, and hence, chances of the above described scenario happening are not negligible. This problem is even more prevalent on multi-core systems, where it is possible for one core to execute code in the TEE, while another is executing code in the REE. To cope with this issue, our CA disables the execution of programs before invoking our TA and re-enables it only after the detection process completes, and control is returned to the CA. To

DocuSign Envelope ID: E4E568A1-8978-433E-A15E-38A89FE5B794

disable the execution of programs, we use the fanotify API, which we provide more information about in Section 6.5.

DocuSign Envelope ID: E4E568A1-8978-433E-A15E-38A89FE5B794

# Chapter 6

# Implementation

## 6.1 Development for OP-TEE

In the past, TEE development required proprietary hardware and software, which is why it was done within the bounds of the company that owned the license for the technology. With the release of OP-TEE, TEE development became a lot easier for independent individuals. Unfortunately, while the software was open source, supported hardware was still hard to acquire. Soon after the release of OP-TEE, Linaro added support for running OP-TEE in QEMU, and some time later QEMU received official support for the ARM TrustZone technology. As of today, OP-TEE supports a variety of available devices, but development is still often done in QEMU, because the environment is more convenient to set up.

QEMU, short for "quick emulator", is a widely used open source machine emulator. It is capable of emulating a variety of client architectures across a number of host architectures, with near native performance through the use of dynamic binary translation.

For our implementation, we use QEMU to run OP-TEE version 3.6 in an emulated, ARMv8-A based system. OP-TEE currently only supports the C language for Trusted Application (TA) development, and only a small part of the standard library is available in the secure world.

DocuSign Envelope ID: E4E568A1-8978-433E-A15E-38A89FE5B794

## 6.2 The hashing process

The hashing process is performed entirely in the Trusted Execution Environment (TEE). We take our predefined list of directory and/or file paths to be monitored, and create a hashing context for each entry. In the case of directories, we recursively read the contents of all files from all subdirectories, and load them into the hashing context. Alternatively, we provide a non-recursive option for hashing directories, where we load only the top level files into the hashing context, and do not traverse the subdirectories. We also provide the option to exclude specific files or subdirectories from the hashing of a directory. Each entry in the list of targets consists of an absolute path, a flag to indicate whether to traverse the subdirectories recursively or not, and a list of files to exclude from the hashing. We produce a single SHA-256 digest for every individual entry in the target list, as shown in Figure 6.1. We use the SHA-256 hash function from the libtomcrypt library.

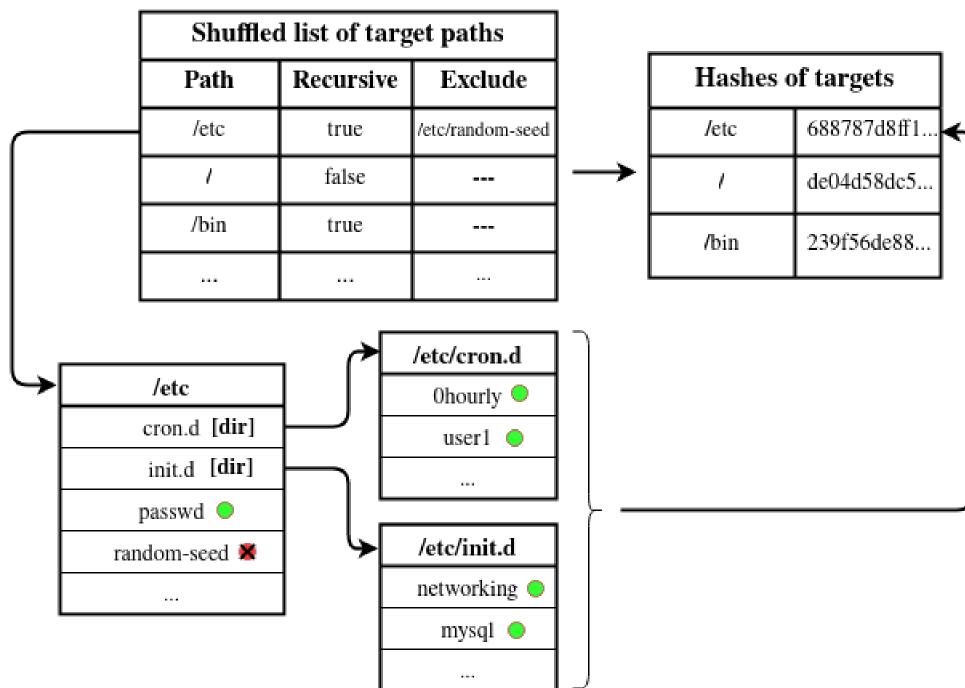


**Figure 6.1:** The recursive hashing of directories results in a single hash for each target entry.

The list of paths to be hashed is shuffled before each new scan to provide a degree of unpredictability to the hashing order; this serves as a secondary hardening technique against timing window attacks, supplementing the use of the fanotify API. This

DocuSign Envelope ID: E4E568A1-8978-433E-A15E-38A89FE5B794

does not affect the digests themselves, since the order in which we read files inside directories remains unchanged.

The baseline hashes are loaded from the trusted storage and compared to the computed hashes. Targets with mismatching hashes are noted and should be examined further by the operators. Since we produce a single hash for each directory, we cannot pin-point exactly which file caused the mismatch in the directory tree. Pinpointing mismatching locations would require keeping a 32-byte hash on the Secure Storage for every entity present on the file system. This approach, however, requires much storage space, which is a drawback in the case of IoT devices, as they are often equipped with only a small flash memory for storage. As a result, a trade-off must be made between the amount of storage available and the precision with which mismatching locations can be identified. It is recommended to keep a full reference hash set based on the initial state of the storage in order to find the exact cause of the mismatch.

## 6.3 Accessing REE file systems from the TEE

As mentioned previously, OP-TEE does not provide access to the REE file system by default. However, since the trusted storage implementation of OP-TEE stores the encrypted Secure Storage data on the REE file system, there must exist a way to access arbitrary files and directories on the same file system.

We discovered that the tee-supplicant daemon can be instructed via RPC calls to perform REE file operations. However, the set of RPC functions available for performing file operations were designed specifically for the trusted storage, not for general purpose access. As a result, to be able to pass arbitrary filenames as parameters, we had to modify the `open` and `opendir` functions. We re-implemented said functions in a Pseudo Trusted Application (PTA), with the parts specific to accessing Secure Storage objects removed. We also had to prefix the path strings with `"../.."`, because the root of the Secure Storage on the REE file system is the `/data/tee` directory.

DocuSign Envelope ID: E4E568A1-8978-433E-A15E-38A89FE5B794

We use this PTA as an interface for our TA, through which we can read and hash data from the REE file system. The PTA interface provides two functions for the TA.

The **hash_file** function expects a filename as a string and a buffer to store the computed hash in. It opens the requested file (if exists), reads its content by 4096 bytes and passes these blocks to a hash function. When the end of the file is reached, it finalizes the hash and copies it into the output buffer.

The **hash_dir** function takes four parameters: a directory name, an output buffer, an integer to indicate if we want to hash recursively (0 for false, 1 for true) and a pointer to a null-terminated array of strings. It creates a hash context, opens the specified directory (if exists) and reads its content. For every entry, we check the blacklist first (to avoid hashing files with volatile content, e.g. `/etc/random-seed`). If the entry is not on the blacklist, we try to determine if it is a regular file or a directory. Since we have no `stat`-like primitive available, we do this by invoking `opendir`. If `opendir` fails, we have a file, otherwise, a directory. For files, we do the same as above: read the file by blocks and feed every block to the hash function. If the entry is a directory and we hash recursively, then the function calls itself recursively on the entry, otherwise, the entry is skipped. Subdirectory traversal is done up until the length of the file path reaches 4096 bytes (same value as `MAX_PATH`, as defined in `linux/limits.h`), and any paths longer than this are assumed to be signs of tampering. Finally, the hash is copied to the output buffer.

In order to ensure the proper functioning of the above described PTA, we had to apply two patches to the tee-supplicant daemon. First, we had to give it root privileges, otherwise it cannot read certain files. Second, tee-supplicant's `readdir` handles certain directories improperly: if a directory only stores hidden files, it is considered to be empty. From the aspect of our persistence checks, this behavior can be fatal, so we had to patch the appropriate function to only skip `"."`, `".."` and `".nfs*"` (these files are created by an NFS server, when an open file is deleted).

DocuSign Envelope ID: E4E568A1-8978-433E-A15E-38A89FE5B794

## 6.4 Updating the baseline

Regular software updates are an integral part of security, and it is important to realize, that users will install updates a lot more frequently, if the update process is convenient and seamless. With every system update, the hashes of the files on the system can and will change. Consequently, our baseline hashes have to be recalculated whenever an update is installed.

The baseline must only be calculated at a moment when the integrity of the system can be verified. To verify system integrity during the boot process, we use a Secure Boot protocol. Secure Boot is based on a chain-of-trust, where each piece of the boot process must be digitally signed before it can start up. The root-of-trust is generally provided by a hardware-based cryptography subsystem, for example, CryptoCell[1] on ARM TrustZone platforms, or simply stored in a non-volatile memory chip. Once one piece of code has been validated, it can then validate the next section and so on until the system is fully booted and control handed over to the operating system.

Because of Secure Boot, integrity can be verified up until the OS has started. This allows us to calculate the baseline during the `init` process. We created a new CA, that is started at a stage of `init`, where the tee-supplicant is active, but networking has not been set up yet, so remote interference is not possible. The new CA will call the update function in our TA. The update function performs the hashing of the predefined directories and files, saves the result on the Secure Storage. Before returning, the TA makes sure that the update function cannot be called again until the device is restarted. We achieved this by introducing a global variable for indicating whether the update function has already been called, and compiling the TA with the `keep_alive` tag, causing that the TA instance context will be preserved, even when there are no sessions connected to the instance.

[1] https://www.arm.com/products/silicon-ip-security/crypto-cell-300, Last visited: 03.12.2020.

DocuSign Envelope ID: E4E568A1-8978-433E-A15E-38A89FE5B794

## 6.5 Securing the detection software

In order to defend against the timing window attack described in Section 5.4, we disable the execution of programs during our rootkit detection process. Our first solution for this problem was to hook certain system calls, similarly to rootkits, but we found a more elegant solution in the form of the fanotify API[2].

The Linux kernel provides APIs for monitoring file system operations (e.g. fanotify, inotify, which both use the same underlying kernel mechanism, fsnotify). Fanotify was introduced in Linux 2.6.36, and was intended to supersede inotify, and solve its deficiencies relating to scalability. It received major updates in Linux 4.2 and Linux 5.1, and became an efficient way to intercept and be notified about file system events on a large file system.

Its intended use cases are virus scanning and hierarchical storage management. Compared to the inotify API, it includes the ability to monitor all of the objects in a mounted filesystem, the ability to make access permission decisions, and the possibility to read or modify files before access by other applications.

To disable the execution of programs, we first create and initialize an fanotify notification group in our CA before invoking our detection function in the TA. This gives us a file descriptor referring to the group. The fanotify notification group is a kernel-internal object that holds a list of files, directories, file systems, and mount points for which events shall be created. We place fanotify marks on all file systems that possess execute permissions.

Our marks are created with the `FAN_OPEN_EXEC_PERM` mask, which results in an event being created whenever a execution type operation occurs on a marked file system. As the name of the mask suggests, these events are not simple notifications, but requests for permission. These requests will be placed in a queue, inside the notification group, along with an open file descriptor for the object being accessed, and a PID or TID for the process or thread that caused the event. The programs that were trying to invoke the execute will be forced to wait until the requests are

[2]`https://www.arm.com/products/silicon-ip-security/crypto-cell-300`, Last visited: 03.12.2020.

DocuSign Envelope ID: E4E568A1-8978-433E-A15E-38A89FE5B794

granted or the marks are released. Our CA will not grant any permissions, but will release the marks after the detection function finishes and the TA returns control back to the CA.

Since fanotify is another kernel functionality and we assume that the kernel is compromised, our TA needs to check if the fanotify marks placed by our CA are intact. Therefore, the CA passes the file descriptor of the notification group, the pid of the CA, and the count of the marks placed to the TA. The TA then locates the fsnotify group among the open files using the received PID and the file descriptor. The TA counts the marks of the proper type, and if it is not equal to the count received from the CA, then the fsnotify group is not intact, and we detected an integrity violation.

DocuSign Envelope ID: E4E568A1-8978-433E-A15E-38A89FE5B794

# Chapter 7

# Evaluation

## 7.1 Verification and validation

In order to illustrate how our solution can detect the persistent components of rootkits, we created a proof of concept user space rootkit, that attempts to avoid detection by executing the timing window attack described in Section 5.4.

Our example rootkit is a statically linked C program, which stores its own executable binary's bytes on the heap, and is able to place and delete the binary on the file system at will. To simulate the perfectly timed scenario, instead of using cron, we used a small bash script to execute the binary on time.

The flow of the attack is shown on Figure 7.1 When first executed, our rootkit loads its own binary into the heap, and then deletes the binary from the file system. After that, it waits for the rootkit detection process to start. Before the bash script launches the scanning process, the rootkit writes the bytes of the binary from the heap into a file, and terminates. At this point, the integrity checks in the memory execute, and do not find signs of the rootkit process, since it terminated before our client gathered the IDs of the running processes.

After the checks in the memory have finished, the bash script, simulating cron, gives execute privileges to the rootkit binary and attempts to execute it. If the fanotify marks have been placed down by our client, the execution will cause an event, and will force the bash script to wait for permission. The rootkit binary will eventually

DocuSign Envelope ID: E4E568A1-8978-433E-A15E-38A89FE5B794

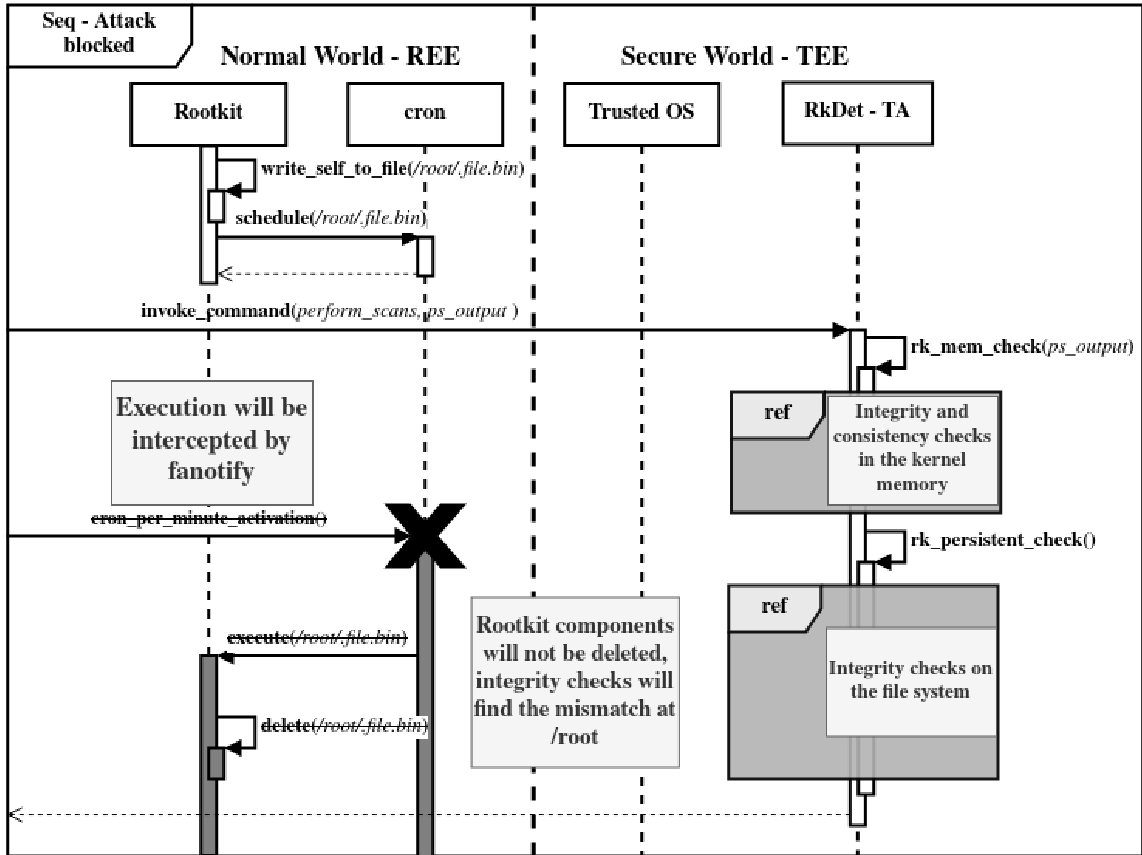


**Figure 7.1:** Cron's activation is blocked by fanotify, the timing window attack will fail.

be read and hashed by the TA, causing a mismatch in the directory hash. If the marks have not been in place, the script would have launched the rootkit, which would have immediately loaded and deleted its binary from the file system, before it could be read and hashed by the TA.

We also tested the attack with cron, instead of the bash script that we used to simulate it. In that case, we scheduled a job to launch our rootkit on time. When the cron daemon returned from its sleep state, it attempted to execute `/bin/sh` to open a shell session where it could run the scheduled job. This execution was blocked by fanotify, so cron could not execute any jobs, including the one that was intended to launch the rootkit.

We tested the persistent component detection process for different inputs, and made sure it works correctly for cases where files are added, removed, or modified. We made sure the process does not fail if it encounters irregular files, like faulty symbolic links. Additionally, we performed static code analysis on the project using

DocuSign Envelope ID: E4E568A1-8978-433E-A15E-38A89FE5B794

flawfinder[1], and investigated each warning, to make sure we have not used any dangerous functions incorrectly, and have not made mistakes that could lead to a buffer overflow.

## 7.2 Performance measurements

Our implementation runs in a virtualized environment with 1057 MBs of RAM and 2 Cortex-A57 cores. The persistent component detection is the most time-consuming part of the whole rootkit detection process, since it requires RPC calls and world switches to read the bytes of the files. Unfortunately, since the I/O operations take up most of the runtime, we cannot significantly reduce it by optimizing other parts of the software.

The directories we hash during each scan are listed on Table 7.1, along with their overall size and number of contained file objects. The average runtime of the process, with the default entries is `49` seconds, with a standard deviation of `1.33` seconds, and a range (max - min) of `3.8` seconds (based on 20 individual measurements).

| Path | Recursive | Num. of objects | Size (Kbytes) |
|---|---|---|---|
| /etc | Y | 37 | 160 |
| /bin | Y | 74 | 49644 |
| /sbin | Y | 54 | 36720 |
| /usr | Y | 192 | 110808 |
| /lib | Y | 62 | 11196 |
| /mnt | Y | 15 | 26136 |
| / | N | 3 | 688 |
| Total | - | 437 | 235352 |

**Table 7.1:** Our default list of monitored directories.

To measure the execution time depending on the number of files and amount of data read, we created files with random content from `/dev/urandom`, and ran the hashing process on the directory containing these files. As shown on Figure 7.2, runtime scales evenly with the amount of data read and hashed, and no errors or additional slowdowns are caused when hashing large files, since we load data into our hashing context in fix sized chunks.

[1] `https://dwheeler.com/flawfinder/`, Last visited: 06.12.2020.

DocuSign Envelope ID: E4E568A1-8978-433E-A15E-38A89FE5B794

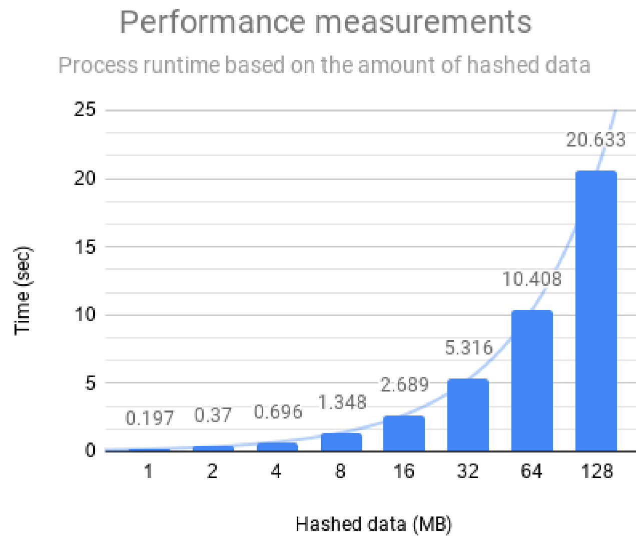


**Figure 7.2:** The runtime of the process scales evenly with the amount of data read and hashed.

Figure 7.3 shows what happens when the same amount of data is divided into multiple files. Having to open and close files causes an additional I/O overhead.

While our scanning process is being executed in the TEE, it uses only one of the cores. As a result, the REE can execute on the other core. Processes running in the REE are not halted while we perform our checks, but significantly less resources are available to them until the checks are completed. Until our TA reaches the file system checks, the core it uses can only execute REE code when an interrupt occurs that has to be handled in the REE. During the file system checks, however, our implementation needs to wait for a lot of I/O operations, and while waiting, control is given back to the REE, where the Linux kernel's scheduler can execute other tasks on the first core as well.

## 7.3 Limitations and possible improvements

The first limitation we have to consider relates to the the way we read and hash data. The sequence that file and directory names are returned by calls to `readdir(2)` is unpredictable. The order of the names returned is entirely dependent on the implementation of the directory, which can differ based on the type of the file system,

DocuSign Envelope ID: E4E568A1-8978-433E-A15E-38A89FE5B794

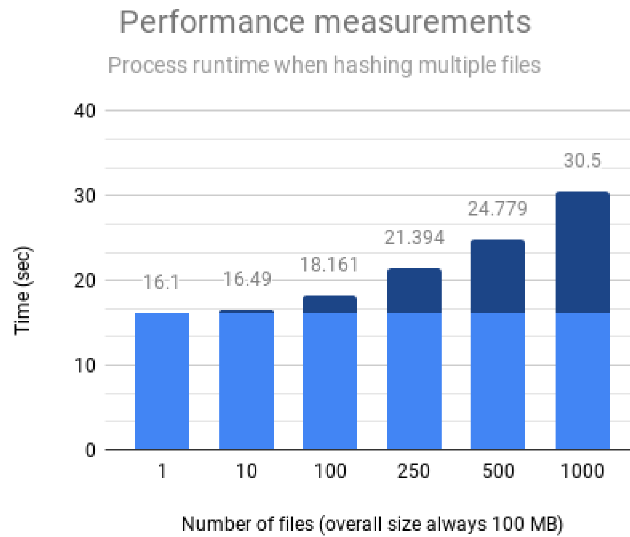


**Figure 7.3:** Hashing a larger number of smaller files results in additional I/O operation cost.

and file order can change as the file system fills gaps in the directory list after files are removed. Inconsistent file order can result in data being loaded into the hashing context in an incorrect order, causing a false positive mismatch detection. A possible solution would be to make a small change to the PTA, to always gather all the file names when opening a directory, and only read them in an alphabetical order.

The second limitation comes from OP-TEE and the way it handles interrupts. Interrupts are divided into two groups, foreign and native interrupts. The former one needs to be handled by the normal world, and the latter one by the secure world. If an interrupt rises and the CPU which should handle it is not in the appropriate world, the machine switches to the correct world, and the interrupt handler is executed. However, OP-TEE does not have its own scheduler, but it uses the Linux kernel's scheduler. When a CPU is executing code in the secure world and a foreign interrupt occurs, the execution of the TA does not continue immediately after the handler exits. It only resumes execution when the scheduler gives the CPU to the thread associated with the TA.

In addition, on a system with multiple cores, it is possible for one core to execute in the Normal World and another in the Secure World. This behavior can make our inconsistency checks unreliable, and can cause false positive detections. This

DocuSign Envelope ID: E4E568A1-8978-433E-A15E-38A89FE5B794

issue might be resolved if we can disable other cores during our checks, and disable interrupts as well to ensure the uninterrupted execution of our checks. Disabling a core is possible from the REE, however, it has a negative impact on the performance of the system. Disabling interrupts is a bit more complicated, although, PTAs can do it while they are running. It might be possible to disable and re-enable interrupts for other secure world threads as well, or the check itself can be implemented in a PTA in order to use this feature.

Finally, a way to bypass all of our checks would be for a malware to uninstall itself before the checks are performed. When the checks are completed, the malware could be reinstalled by exploiting the same vulnerability it originally exploited to infect the system. Note, however, that this can work against any rootkit detection approach, because there remains nothing malicious to detect in the system.

DocuSign Envelope ID: E4E568A1-8978-433E-A15E-38A89FE5B794

# Chapter 8

# Conclusion

In this thesis, we addressed the problem of detecting persistent rootkit components on embedded IoT devices.

Rootkits are malicious software that run with elevated privileges, and employ a wide variety of techniques to remain hidden. Attackers install rootkits on compromised systems to maintain the privileged access they obtained by exploiting a vulnerability in the system. Rootkits are able to survive system reboots through the use of persistent components, which they write on the persistent storage (i.e. the hard disk or solid state drive). Persistent components may be hidden using different hooking techniques, but these techniques leave detectable traces in the kernel memory. A sophisticated rootkit may attempt to avoid rootkit detection software by restoring the integrity of the memory, hiding itself in a persistent component, and restarting after the detection process finishes. Such a rootkit could only be detected by monitoring the file system for persistent components.

Our work was part of a larger project, that aimed to create a full-fledged rootkit detection software, which leverages the Trusted Execution Environment technology. Such Trusted Execution Environments are supported on many embedded platforms used in IoT applications, and their protection measures ensure that malicious code cannot interfere with our detection mechanisms even when running with root privileges. Our solution for the detection of persistent rootkit components is based on File Integrity Monitoring (FIM), which is a technique for monitoring and detecting

DocuSign Envelope ID: E4E568A1-8978-433E-A15E-38A89FE5B794

changes in files by comparing the hashes of the monitored files to a secure baseline. We described in detail how we read and hash the file system data of the untrusted execution environment, how we solved the secure storage of our baseline hashes, and how we implemented a secure, automatic baseline update mechanism. We addressed the potential vulnerabilities in the designed architecture, and built countermeasures to make the application safe and reliable. Finally, we evaluated our design and implementation by testing the prototype with a rootkit that we developed for this purpose, and listed some limitations along with some possible solutions.

In summary, we are able to detect the presence of rootkit components in the persistent storage of the IoT device in a reliable and safe way, with an acceptable impact on system performance. Despite its limitations, we believe that our work can contribute to protecting the small embedded devices used in IoT applications from sophisticated and powerful software based attacks.

DocuSign Envelope ID: E4E568A1-8978-433E-A15E-38A89FE5B794

# Acknowledgements

The presented work was carried out within the SETITProject (2018-1.2.1-NKP-2018-00004), which has been implemented with the support provided from the National Research, Development and Innovation Fund of Hungary, financed under the 2018-1.2.1-NKP funding scheme.

I would like to thank my colleague and research partner, Roland Nagy, for his guidance and insights into this field, and for his great ideas that always helped make the project move forward.

DocuSign Envelope ID: E4E568A1-8978-433E-A15E-38A89FE5B794

# List of Figures

1.1 Number of connected IoT devices worldwide. . . . . . . . . . . . . . . . 2

2.1 Components of the Linux Virtual File System (VFS). . . . . . . . . . . 12

3.1 The two types of hypervisors. . . . . . . . . . . . . . . . . . . . . . . . . . 18

3.2 The arrangement of security states and exception levels in ARMv8-A architecture processors. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 21

3.3 The structure of OP-TEE. . . . . . . . . . . . . . . . . . . . . . . . . . . . 22

5.1 The functionality of the client application. . . . . . . . . . . . . . . . . 28

5.2 The functionality of the trusted application. . . . . . . . . . . . . . . . 29

5.3 The extraction of shell commands from crontab files, and resolution of non-absolute paths. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31

5.4 A simple shell command and its corresponding AST. . . . . . . . . . . 32

5.5 The format of the hashes stored on the Secure Storage. . . . . . . . . . 34

5.6 Cron executing the attacker's code during the timing window between the integrity checks. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 36

6.1 The recursive hashing of directories results in a single hash for each target entry. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 39

7.1 Cron's activation is blocked by fanotify, the timing window attack will fail. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 46

DocuSign Envelope ID: E4E568A1-8978-433E-A15E-38A89FE5B794

7.2 The runtime of the process scales evenly with the amount of data read and hashed. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 48

7.3 Hashing a larger number of smaller files results in additional I/O operation cost. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 49

DocuSign Envelope ID: E4E568A1-8978-433E-A15E-38A89FE5B794

# Bibliography

[1] T. Abou-Assaleh, N. Cercone, V. Keselj, and R. Sweidan. N-gram-based detection of new malicious code. In *Proceedings of the 28th Annual International Computer Software and Applications Conference, 2004. COMPSAC 2004.*, volume 2, pages 41–42 vol.2, 2004. DOI: `10.1109/CMPSAC.2004.1342667`.

[2] M. Antonakakis, T. April, M. Bailey, M. Bernhard, E. Bursztein, J. Cochran, Z. Durumeric, J. A. Halderman, L. Invernizzi, M. Kallitsis, D. Kumar, C. Lever, Z. Ma, J. Mason, D. Menscher, C. Seaman, N. Sullivan, K. Thomas, and Y. Zhou. Understanding the Mirai botnet. In *USENIX Security Symposium*, August 2017. URL `https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/antonakakis`.

[3] Arati Baliga, Vinod Ganapathy, and Liviu Iftode. Detecting kernel-level rootkits using data structure invariants. *IEEE Trans. Dependable Sec. Comput.*, 8: 670–684, 09 2011. DOI: `10.1109/TDSC.2010.38`.

[4] Boldizsár Bencsáth, G. Pék, L. Buttyán, and M. Félegyházi. The cousins of stuxnet: Duqu, flame, and gauss. *Future Internet*, 4:971–1003, 2012.

[5] Robert Buhren, Julian Vetter, and Jan Nordholz. The threat of virtualization: Hypervisor-based rootkits on the arm architecture. volume 9977, 11 2016. ISBN 978-3-319-50010-2. DOI: `10.1007/978-3-319-50011-9_29`.

[6] S. Devik. Linux on-the-fly kernel patching without LKM. *Phrack Magazine*, 2001. http://phrack.org/issues/58/7.html, Last visited: 21.09.2020.

DocuSign Envelope ID: E4E568A1-8978-433E-A15E-38A89FE5B794

[7] Sh. Embleton, Sh. Sparks, and C. Zou. SMM rootkit: a new breed of OS independent malware. *Security and Communication Networks*, 6(12):1590–1605, 2013.

[8] Tal Garfinkel and Mendel Rosenblum. A virtual machine introspection based architecture for intrusion detection. *NDSS*, 3, 05 2003.

[9] J. Heasman. Implementing and detecting an ACPI BIOS rootkit. *Black Hat Europe*, 2006.

[10] T. Hudson and L. Rudolph. Thunderstrike: EFI firmware bootkits for Apple MacBooks. In *Proceedings of the 8th ACM International Systems and Storage Conference*, pages 1–10, 2015.

[11] T. Hudson, X. Kovah, and C. Kallenberg. Thunderstrike 2: Sith Strike. *Black Hat USA Briefings*, 2015.

[12] Ralf Hund, Thorsten Holz, and Felix Freiling. Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms. pages 383–398, 01 2009.

[13] Sye Loong Keoh, Sandeep Kumar, and Hannes Tschofenig. Securing the internet of things: A standardization perspective. *Internet of Things Journal, IEEE*, 1: 265–275, 06 2014. DOI: `10.1109/JIOT.2014.2323395`.

[14] Samuel King, Peter Chen, Yi-Min Wang, Chad Verbowski, Helen Wang, and Jacob Lorch. Subvirt: Implementing malware with virtual machines. volume 2006, pages 314–327, 01 2006. DOI: `10.1109/SP.2006.38`.

[15] D. Kushner. The real story of stuxnet. *IEEE Spectrum*, 50(3):48–53, 2013. DOI: `10.1109/MSPEC.2013.6471059`.

[16] Iain Kyte, P. Zavarsky, D. Lindskog, and R. Ruhl. Detection of hardware virtualization based rootkits by performance benchmarking. 2010.

[17] M. Leibowitz. Horse Pill: A new kind of Linux rootkit. In *Black Hat USA*, 2016.

DocuSign Envelope ID: E4E568A1-8978-433E-A15E-38A89FE5B794

[18] D. Narsingyani and O. Kale. Optimizing false positive in anomaly based intrusion detection using genetic algorithm. In *2015 IEEE 3rd International Conference on MOOCs, Innovation and Technology in Education (MITE)*, pages 72–77, 2015. DOI: 10.1109/MITE.2015.7375291.

[19] B. Ngabonziza, D. Martin, A. Bailey, H. Cho, and S. Martin. Trustzone explained: Architectural features and use cases. In *2016 IEEE 2nd International Conference on Collaboration and Internet Computing (CIC)*, pages 445–451, 2016. DOI: 10.1109/CIC.2016.065.

[20] Nick L. Petroni, Timothy Fraser, Jesus Molina, and William A. Arbaugh. Copilot - a coprocessor-based kernel runtime integrity monitor. In *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*, SSYM'04, page 13, USA, 2004. USENIX Association.

[21] Sandro Pinto and Nuno Santos. Demystifying arm trustzone: A comprehensive survey. *ACM Computing Surveys*, 51:1–36, 01 2019. DOI: 10.1145/3291047.

[22] D. Krishna Sandeep Reddy, Subrat Kumar Dash, and Arun K. Pujari. New malicious code detection using variable length n-grams. In Aditya Bagchi and Vijayalakshmi Atluri, editors, *Information Systems Security*, pages 276–288, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[23] Ethan M. Rudd, Andras Rozsa, Manuel Günther, and Terrance E. Boult. A survey of stealth malware: Attacks, mitigation measures, and steps toward autonomous open world solutions, 2016.

[24] M. Sabt, M. Achemlal, and A. Bouabdallah. Trusted execution environment: What it is, and what it is not. In *2015 IEEE Trustcom/BigDataSE/ISPA*, volume 1, pages 57–64, Aug 2015. DOI: 10.1109/Trustcom.2015.357.

[25] Morteza Safaei Pour, Elias Bou-Harb, Kavita Varma, Nataliia Neshenko, Dimitris A. Pados, and Kim-Kwang Raymond Choo. Comprehending the iot cyber threat landscape: A data dimensionality reduction technique to infer and characterize internet-scale iot probing campaigns. *Digital Investigation*, 28:S40 – S49, 2019. ISSN 1742-2876.

DocuSign Envelope ID: E4E568A1-8978-433E-A15E-38A89FE5B794

DOI: https://doi.org/10.1016/j.diin.2019.01.014. URL http://www.sciencedirect.com/science/article/pii/S1742287619300246.

[26] sasha / beetle. A strict anomaly detection model for ids. *Phrack Magazine*, 2000. http://phrack.org/issues/56/11.html, Last visited: 27.11.2020.

[27] Muazzam Siddiqui, Morgan Wang, and Joohan Lee. A survey of data mining techniques for malware detection using file features. pages 509–510, 01 2008. DOI: 10.1145/1593105.1593239.

[28] A. Todd, J. Benson, G. Peterson, T. Franz, M. Stevens, and R. Raines. Analysis of tools for detecting rootkits and hidden processes. In Philip Craiger and Sujeet Shenoi, editors, *Advances in Digital Forensics III*, pages 89–105, New York, NY, 2007. Springer New York. ISBN 978-0-387-73742-3.

[29] Zhi Wang, Xuxian Jiang, Weidong Cui, and Peng Ning. Countering kernel rootkits with lightweight hook protection. pages 545–554, 01 2009. DOI: 10.1145/1653662.1653728.