Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Networked Systems and Services

Enhancing the Security of the Internet of Things: Design and Implementation of Security Mechanisms for Embedded Platforms

Szilárd Dömötör, Márton Juhász, Gábor Székely, István Telek

Supervisor:
Dr. Levente Buttyán

October 28, 2018

Contents

Abstract

The Internet of Things, or IoT in short, refers to connecting millions of embedded devices to the Internet. Most of these devices have resource constraints, which often results in sub par security. However, cyber attacks against these systems may often lead to substantial physical damage or monetary loss in these applications. Therefore, one of the biggest challenges today, which hinders the adoption of IoT technologies in certain areas, is the lack of security guarantees.

In typical application environments, IoT devices are not directly connected to the Internet, but they are using gateway devices. Gateways usually have more resources and they may be physically better protected, so they can perform security functions to protect themselves and the IoT devices that they serve. Therefore, placing IoT devices behind gateways and protecting the gateways from cyber attacks seems to be the best approach to increase the security of IoT systems.

In this paper, we propose a set of security mechanisms, which together form a security architecture for IoT gateways. Our solution is heavily based on a Trusted Execution Environment (TEE), which provides an isolated execution space that we can better trust for behaving correctly and protecting sensitive secrets during computations than does the main firmware/OS itself. Such a TEE could be provided by a security co-processor to which sensitive operations can be delegated, but this approach is often too expensive in practice. That is why we choose a more cost efficient solution: a software based TEE, with some hardware support such as ARM's TrustZone technology. In particular, we use OP-TEE, an open source TEE implementation.

In our paper, we provide solutions for securing the boot and update processes and apply various OS hardening techniques. Leveraging OP-TEE, we improve trustworthiness by using run-time integrity monitoring and remote attestation of the device state. Finally we create a way to use secrets securely stored in OP-TEE for remote access.

Kivonat

Az Internet of Things, röviden IoT, lényegében több millió beágyazott eszköz internethez való csatlakoztatását foglalja magába. A legtöbb ilyen eszköz korlátozott erőforrásokkal rendelkezik, ami gyakran átlagon aluli biztonságot eredményez. Azonban az ilyen rendszerek elleni kibertámadások gyakran jelentős fizikai, vagy pénzügyi károkat okozhatnak. Ezért napjaink egyik legnagyobb kihívása, ami hátráltatja az IoT technológia elterjedését bizonyos területeken, a biztonsági garanciák hiánya.

A tipikus alkalmazási környezetükben, az IoT eszközök nem közvetlenül kapcsolódnak az internethez, hanem egy átjárón keresztül. Ezen átjárók általában több erőforrással rendelkeznek és fizikailag is jobban védhetőek, így elláthatnak biztonsági funkciókat saját maguk és az általuk kiszolgált IoT eszközök számára. Ebből kifolyólag, ezen rendszerek biztonságának növeléséhez a legjobb hozzáállásnak az tűnik, ha az IoT eszközöket átjárók mögé helyezzük, és magát az átjárót védjük meg kibertámadások ellen.

Ebben a dolgozatban olyan biztonsági mechanizmusokat javaslunk, melyek együttesen egy biztonsági architektúrát alkotnak IoT átjárók számára. Megoldásunk erősen támaszkodik egy Trusted Execution Environment (TEE) alkalmazására az átjárón, mely elkülönített végrehajtási környezetet biztosít, amiben jobban meg lehet bízni, hogy helyesen működik és megvédi a titkainkat, mint a fő firmware vagy operációs rendszer. Egy ilyen környezetet biztosíthat egy dedikált biztonsági processzor aminek delegálhatunk érzékeny feladatokat, viszont ez a megközelítés a gyakorlatban sok esetben túl költséges. Ezért mi egy költséghatékonyabb, szoftver alapú TEE-t választottunk, mely az ARM TrustZone hardveres technológiára támaszkodik. A megoldásaink az OP-TEE nyílt forráskódú TEE implenetációt használják.

Dolgozatunkban, megoldást nyújtunk a rendszerindítási és -frissítési folyamat biztonságossá tételére, illetve alkalmazunk különböző hardening technikákat. Az OP-TEE használatával növeljük a bizalmat az eszköz állapotában futásidejű integritás ellenőrzéssel és ennek távoli igazolhatóságával. Továbbá módot adunk rá, hogy az OP-TEE-ban biztonságosan tárolt titkokat távoli elérésre használjuk.

Chapter 1

Introduction

Moore's law [30] – a prediction that the number of transistors on a single silicon die doubles around every 2 years – has been holding for more than 50 years. Combining this steady increase in computing power and shrinking size with decreasing price resulted in small computers that can be embedded in many everyday items, such as fridges, weather measurement stations, home thermostats, cars, coffee makers, yoga mattresses, lightbulbs, doorbells, *etc.* Moreover, as internet access became cheaper and widely accessible, these embedded computers are now connected to the Internet, to enable remote operation and monitoring, as well as to allow embedded devices to perform networked based transactions like shopping or uploading data in the cloud. The resulting ecosystem has come to be called the *Internet of Things* or IoT for short, and the devices that belong to this ecosystem are called *IoT devices* or "smart" devices.

However, besides the advantages that Internet connection can bring, a new attack surface for adversaries is created, which can expose devices that were before unreachable for attackers. The impact of attacks on smart medical devices, connected vehicles or smart factories is high, potentially leading to significant monetary loss, or even danger to human life. On the other hand, as a result of the history of IoT, low price remains one of the main objectives, and when costs need to be saved in a product, security is usually one of the first aspects that suffers. It is also very hard for the average consumer to evaluate the security of a device, so there is no monetary incentive for investing a lot of money into producing a very secure product until a huge cyber incident happens. This has lead to IoT devices on the market having terrible security: default passwords in realeased devices [25], and many incidents, like the gigantic DDoS attack on major web based services by the Mirai botnet that is built from compromised IoT devices [2].

In this work, we address the current insecurity of the Internet of Things, and we propose a set of security mechanisms that together form a security arcitecture applicable to embedded IoT devices. We believe that making embedded computing platforms secure results in increased overall security of the entire IoT ecosystem. More specifically, in this work, we address the following problems:

- **Secure boot:** The first step of securing an embedded device is securing its boot process. It must be ensured that after a reset, the device boots into a known and secure state. This can be achieved by digitally signing the software components loaded during the boot process, and enforcing each stage to verify the digital signature on the next stage before it is loaded and executed. If the verification fails, the boot process should be halted, ensuring that compromised devices are excluded from the system.

- **Secure firmware update:** Devices may have software vulnerabilities that could be exploited to compromise them. The immediate response to a run-time compromise could be rebooting the device, as secure boot would bring it back to a known good state. However, in the long run, the vulnerability that made the compromise possible should also be eliminated, otherwise the device can be compromised again and again by exploiting the same vulnerability. Eliminating vulnerabilities is solved usually by patching or updating the entire fixed firmware/operating system. Both require a secured software update process, which must also be fail-safe, meaning that if an update goes wrong and results in the device not being able to function properly, it should automatically be reverted to the latest stable version, at least temporarily. At the same time, however, version rollback must be prevented, to stop an attacker from loading a previous, vulnerable version of the software/firmware/OS.

- **Security hardened firmware/operating system:** Beyond secure boot and software/firmware/OS update, in order to improve the security and reduce the attack surface even more, the firmware/OS running on the device should be hardened. Hardening is about disabling or restricting unused/unnecessary services and enabling or upgrading available protection mechanisms that are optional and not used by default. These are disabled by default, because they can slow down the system and/or can cause incompatibilities. Hardening reduces the attack surface and makes the system more resistant to typical known attacks.

- **Integrity monitoring and remote attestation:** Despite hardening, the device may still have vulnerabilities that could be exploited at run-time. Secure boot would bring the device back to a known good state temporarily, and secure software updates could be used to eliminate the vulnerabilities, but a prerequisite for both is to detect that the device has been compromised in the first place. Thus, periodical integrity verification of the system is useful and can identify possible run-time compromises, malicious components, failures, misconfigurations, break-in attempts, or any other anomaly that may be an indication of compromise. In general, the integrity verification of the running firmware/OS should be performed by a system component located in a so called Trusted Execution Environment, rather than in the firmware/OS itself, since the latter may have already been compromised, making the integrity verification result unreliable. Having the integrity verification mechanism executed with a trusted environment ensures that it cannot be defeated even if the firmware/OS of the devices is compromised. Fortunately, modern embedded

processors, such as ARM, support the establishment of such a Trusted Execution Environment. In addition, while integrity monitoring is useful to detect changes in the device's state, remote attestation can be used to assess the integrity of the device from a remote location. This technique is especially convenient for monitoring the integrity of small connected devices on a large scale, because it can be automated and it does not require physical proximity to the device.

- **Secure communications:** Embedded IoT devices should be able to communicate securely with their owner or operator. This is needed not only for data transmission but also for providing secure remote access to the device for configuration and management purposes. Secure communications can be implemented with cryptographic mechanisms. However, long-term secret passwords or private keys needed for authenticating the device to the owner/operator when setting up a secure communication session should not appear in memory accessible to the potentially compromised firmware/OS. This means that such secret keys must be stored in secure storage and the cryptographic operations that use them should be implemented by trusted applications executed in the same Trusted Execution Envrinment where the integrity monitoring and remote attestation components are running.

The above described problems represent a number of challenges when one tries to solve them on resource constrained IoT devices. For instance, implementing secure boot requires some hardware root of trust that provides a trusted implementation of an initial boot loader and secure storage of a public key with which the initial boot loader can verify the digital signature of the next stage software compnent. A hardware root of trust is typically some tamper resistant module that can withstand physical attacks and that can serve as a root of trust for other security functions as well. Such physically protected elements may not be available on cheap IoT devices or even if they are available, they provide limited functionality and protection. So we must make minimal assumptions about the availability and functionality of any hardware root of trust that we use in our design.

Let's take secure firmware update as the next example for challenges. The update process must be fail-safe, which requires a capability to detect faulty updates, including the detection of a system hang and some sort of checking whether the updated firmware is working correctly or not, and a mechanism to roll back the device to a previous working firmware version when the update does not work. At the same time, we must prevent version rollback attacks, where attackers force a device to roll back to a previous, insecure firmware version.

Security hardening also comes with challenges. In practice, hardening includes the modification of various build and run-time settings. However, increasing the level of security and reducing the attack surface typically results in a noticeable performance drop, and maybe even loss of some functionality.

Integrity monitoing and remote attestation are also hard problems. Integrity monitoring itself is far from being trivial, even if it is executed in a Trusted Execution Environment.

First, the application, running in the trusted environment, must have complete access to the OS or firmware memory. This can be challenging, since the two execution environments probably use different memory management tables, so memory addresses from the OS must be translated, to become usable in the trusted environment. The integrity monitoring application may extract the list of the currently running processes to use with a whitelist, or it can calculate a hash value for processes to detect possibly malicious modification of their binaries. For these procedures, we must correctly interpret and use the same data structures the OS uses, which are highly dependent on the OS kernel version. In addtion, implementing remote attestation techniques can be challenging, because many security criteria have to be met, and a secure protocol has to be implemented.

Finally, protecting cryptographic keys and operation has its own difficulties too. There are multiple scenarios where delegating cryptographic operations to a Trusted Execution Environment is advantageous, such as encrypted communication, disk encryption and remote access to the device. Therefore, it is useful to use a uniform API to make this functionality available to multiple purposes. There exist standard cryptographic APIs for this, such as the PKCS#11 API, which is a widely used standard that many client applications already support. However, PKCS#11 is a rather large specification, so one may only want to implement a subset of it, and this subset must be carefully chosen.

The rest of the paper is organized in the following manner: In Section 2, we explore the literature of the state of the art in Trusted Execution Environments and the various problems outlined in the above sections. In Section 3, we introduce the design of our proposed security components. In Section 4, we expand on the details of our implementation of these components. In Section 5, we evaluate our results, and finally, in Section 6, we give a summary of our work and some possible future improvements.

Chapter 2

Related work

Below, we introduce technologies and research findings that are relevant to our work.

2.1 Secure Boot

To have a basic understanding of securing the boot process in an embedded environment, a timesys blog post [6] summarizes its key concepts, and lists various useful tools to implement it.

Looking up the available methods, we found, that the Chromium OS project developed a great one [20]. As a matter of fact they also designed and implemented or helped some of the tools, commonly used in a secure boot process. Their solutions are well documented, and reading their related design documents [37, 39, 38, 40] was a good start in the making of our solution.

2.2 Secure Firmware Update

There are different ways of making a secure firmware update process in an embedded environment, and a timesys blog post [7] helps to clarify its main aspects.

Just like the secure boot process, understanding the secure firmware update process in the Chromium OS project [36] was also a good start in the making of our solution.

2.3 Integrity monitoring

Approaching kernel integrity monitoring using tools that are not isolated from the operating system they are protecting, should not be considered a secure technique. The compromise of the kernel could lead to the compromise of the monitoring tools. In the recent years, mainly two types of approaches appeared—hardware based and hypervisor based solutions.

An example of the hypervisor based approach is HIMA (Hypervisor-Based Integrity Measurement Agent) [3]. HIMA advocates two key requirements for integrity measurement—strong isolation between the monitored system and the monitoring tool, and Time Of Check To Time Of Use (TOCTTOU) consistency between the measured and executed versions of the target (so the attacker can not modify a program after measurement but before execution). It performs integrity monitoring of Virtual Machines (VM) running on top of a hypervisor, and ensures the previous capabilities. The first capability is satisfied by memory protection—stopping attackers from modifying the monitoring tool or its results. For the second, the monitoring tool responds to events happening in the memory of the VMs—ensuring that measurements are up-to-date.

The problem with hypervisor based approaches is that the hypervisors are complex, and exposed to software vulnerabilities which can be exploited to break the isolation.[1] Also, virtualization processor extensions are not available on all platforms.

Hardware based solutions (e.g., [45, 5]) mostly use trusted execution environments such as ARM TrustZone[2], AMD SVM [13], or Intel TXT [23]. Alternatively, they could be based on a separate hardware component (e.g., security co-processor or other SoC), as seen in [33, 29]. One usual shortcoming of hardware based solutions is that they lack event driven monitoring capabilities, therefore the previously mentioned problem of TOCTTOU affects them. Also, they usually lack control over Normal World kernel functions, e.g., the NW kernel having full control over memory management can cause rootkits to remain undetected from integrity monitoring.

The two following examples of the hardware based approach solves the previous shortcomings: TZ-RKP (TrustZone-based Real-time Kernel Protection) and SPROBES [4, 18]. They were developed in parallel, with the aim to enforce kernel code integrity using ARM TrustZone technology. In TZ-RKP, the integrity monitoring software is running in Secure World, and novel methods are used to deprive the Normal World kernel from using privileged system functions. Thus, assuring that attackers can not bypass the monitoring tools, modify or inject kernel binaries, or modify the system memory layout. It can intercept critical operations and analyze their security impact. It is used in phones and tablets. SPROBES present an instrumentation mechanism backed by TrustZone which can transparently break on any Normal World instruction and give control to Secure World event handlers. It also identifies enforceable invariant that can restrict rootkits from removing SPROBE calls.

2.4 Remote attestation

Remote attestation is a challenging problem on low-end embedded devices, because existing techniques often require secure hardware components such as TPMs [44] or secure

[1] `http://www.cvedetails.com/vendor/252/Vmware.html` `https://www.cvedetails.com/vendor/6276/XEN.html` (links last visited on 2018–10–23)

[2] `https://developer.arm.com/technologies/trustzone` (last visited on 2018–10–23)

coprocessors [43]. Several principles are crucial for attestation architectures, and while an ideal attestation architecture would satisfy all of them, many constraints have to be met in a real world application, which is why it is hard to create a universal implementation [9]. Several low-end ARM devices offer the TrustZone Trusted Computing technology, but this technology doesn't include an existing hardware-based attestation mechanism [32] which is why it is not directly suitable to provide a remote attestation architecture. Purely software-based attestation techniques often fail to provide the necessary security level, so the development of a mixed hardware-software solution is advised [17].

2.5 Keystore APIs and TEE based keystores

The idea to implement a keystore in the TEE is not new. Android has the Android Keystore [10], which is accessible through the KeyChain API and the Android Keystore provider. If the manufacturer of the phone provides a driver that enables it it to use a TEE, it will do so, otherwise it defaults to a software implementation. Other examples of APIs that allow operations on stored keys and offer protection for them can be found in cryptographic tokens such as smart cards and hardware security modules (HSM). Namely, [8] mentions two such: the PKCS#11 [31] and the IBM CCA (Common Cryptographic Architecture). PKCS#11 is a widely used interface for all kinds of cryptographic tokens and IBM CCA is primarily used by banks, with IBM cryptoprocessors providing the interface. [8] constructs a provably secure cryptographic interface and creates an emulation of this security policy by restricting the PKCS#11 interface.

Chapter 3

Design

3.1 Used Components

3.1.1 ARM TrustZone

"The ARM TrustZone technology provides system-wide hardware isolation for trusted software."[1] It aims to enable the construction of a trusted execution environment, where the integrity and confidentiality of assets can be protected from attackers. The separation of the two environments—Secure World (SW)[2] for the security subsystem, and Normal World (NW)[3] for every other system component—is achieved by partitioning the SoC's resources. This is enforced via hardware logic on the system bus, disabling the Normal World from accessing the assets residing in–, and resources dedicated to Secure World [1]. A high level overview example of the hardware isolation between the two worlds can be seen on figure 3.1.

3.1.2 GPD TEE Specification

Devices offer a Rich Execution Environment (REE) which brings flexibility and capability, but leaves the device vulnerable to a number of security threats. The Trusted Execution Environment (TEE) is designed to reside alongside the REE and provide a safe area of the device to protect its assets and execute trusted code.

The highest level of the TEE is an environment which meets the following criteria:

- All code executing inside the TEE must have been authenticated.

[1] `https://developer.arm.com/technologies/trustzone` (last visited on 2018–10–23)
[2] Also called Trusted Execution Environment
[3] Also called Rich Execution Environment
[4] `https://www.timesys.com/security/trusted-software-development-op-tee/` (last visited on 2018–10–23)

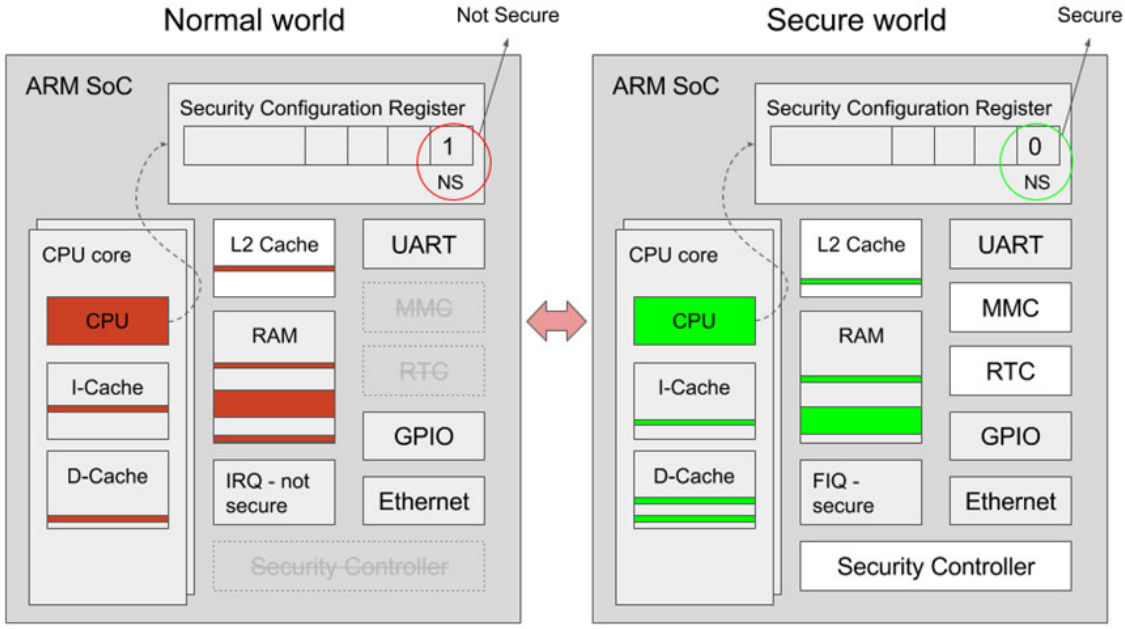

Figure 3.1. *Hardware overview example of ARM TrustZone Security*[4]

- The integrity of the TEE assets and the confidentiality of the contents of TEE data has to be assured through isolation, cryptography and other security mechanisms. This includes keys which are used by the cryptographic primitives.
- TEE has to resist known remote and local software attacks, and a set of external hardware attacks.
- Code and other assets have to be protected from unauthorized tracing and control through debug and test features.

The GPD TEE Internal Core API contains a Cryptographic Operations API [22], that provides an interface to perform cryptographic operations with the help of the TEE. Depending on the actual TEE and its configuration, the algorithms may be implemented in software or backed by hardware as well. However, as long as the given TEE follows the standard we can be sure that the API calls do what they are specified to do and the data and keys can't be accessed from the Normal World. These properties would make this interface ideal for our use case, however this API is only exposed to Secure World, and also not as widely used in client applications as PKCS#11 for instance.

3.1.3 OP-TEE

There are a numerous TEE implementations, but it was still hard to find one that fits our needs. One of the reasons is that many of the TEEs are closed source, and developing applications for them is either impossible, or requires signing NDAs. It is also reasonable to limit our scope to TEEs that are compatible with ARM chips, because IoT devices are mostly based on such SoCs. As a consequence we will rely on TrustZone technology.

[41] places TEEs based on TrustZone technology into two groups: industrial and academic TEEs. We wanted our solutions to be usable in real world devices, so we chose an industrial TEE. Some of the industrial TEEs mentioned in [41]:

- QSEE
- Trustonic
- Securi-TEE
- SierraTEE
- OP-TEE
- Nvidia TLK

Out of these, closed source options were instantly discarded, since they don't allow modification of the system, that may be needed. From the remaining TEEs we chose OP-TEE, as it seemed to be the easiest to access - it is available on GitHub - and had the most active development. It supports a wide range of devices.

Another option would be using a separation kernel that is mathematically proven to correctly separate different running processes. An example of this is the seL4 microkernel [26]. However the drawback of this aproach is that this limits adoption, since in our experience there aren't as many programs written for these microkernels and there are fewer people who would have the knowledge to create applications in such an environment.

"OP-TEE is an open source project which contains a full implementation to make up a complete Trusted Execution Environment. The project has roots in a proprietary solution, initially created by ST-Ericsson and then owned and maintained by STMicroelectronics. In 2014, Linaro started working with STMicroelectronics to transform the proprietary TEE solution into an open source TEE solution instead. In September 2015, the ownership was transferred to Linaro. Today it is one of the key security projects in Linaro, with several of Linaro's members supporting and using it."[5]

OP-TEE consists of two main components, the OP-TEE Kernel[6] and the OP-TEE Client API[7]. There are several platforms which are supported by default. OP-TEE implements the interfaces defined by the GPD TEE Specification.

3.1.4 Linux

In our design, the Linux operating system plays a main role. Numerous IoT devices are running some variant of the Linux kernel (e.g., SOHO wireless routers). In OP-TEE, the

[5] `https://www.op-tee.org/about/` (last visited: 2018-10-22)
[6] `https://github.com/OP-TEE/optee_os` (last visited: 2018-10-22)
[7] `https://github.com/OP-TEE/optee_client` (last visited: 2018-10-22)

only supported Normal World operating system is Linux, because the required NW software and kernel modules are only implemented for it. Therefore, the proposed secured gateway also uses Linux.

Our design also includes a Linux kernel module which is required for the normal operation of the integrity monitoring algorithms. This topic is discussed in section 4.5.8. A kernel module is a compiled object file which can be used to extend the functionality of the kernel, solving the general extensibility and modularity problems of monolithic kernels. They can be loaded and unloaded upon demand, at run time [42].

3.2 Secure Boot

The principle is rather simple here, as the process relies on the well known method of establishing a chain of trust. To be able to build up a chain of trust, an element, that is considered as the root of trust, is needed. Because the boot loader must be verified first, the root of trust must be a hardware component, dedicated to do that, as after powering on the device, the first software to run is the boot loader. In a chain of trust the stages are digitally signed, so the hardware root of trust must store the public signature verification key of the boot loader in a physically write-protected memory (for example, a one-time programmable memory, written during manufacturing), to only allow such a boot loader to run, that is signed with the matching private key.

With the boot loader verified, the process must carry on to the next stage, which contains the systems software components. The components are loaded and executed by the boot loader, so such a boot loader must be chosen, that is capable of verifying all of those components with their public signature verification key. The public key must be stored in the boot loader itself, so only such components are allowed to be loaded and executed, that are signed with the private key of the same key pair.

The operating system is one of the above mentioned system components, so it is already verified by the time it is running. But if it uses any other component, not verified by the boot loader (for example, a file system), it must verify that by itself. In such a case, the operating system must have the ability to verify those components, like the boot loader verifies the other system components. And with any further extra component, the secure boot process can be extended in the same manner.

If the verification of any given stage fails, the boot process must be halted, as it is the essence of the secure boot process. Even if the operating system is already running, but the verification of a subsequent stage, which is still part of the secure boot process, fails, the process must be halted. Whether there is a fallback system, or the device is intentionally inoperable in case of the failure, is use case dependent. Generally the easiest and maybe most secure approach is to render the device intentionally inoperable in case of a failure, however fallback mechanisms are discussed later in the secure firmware update process.

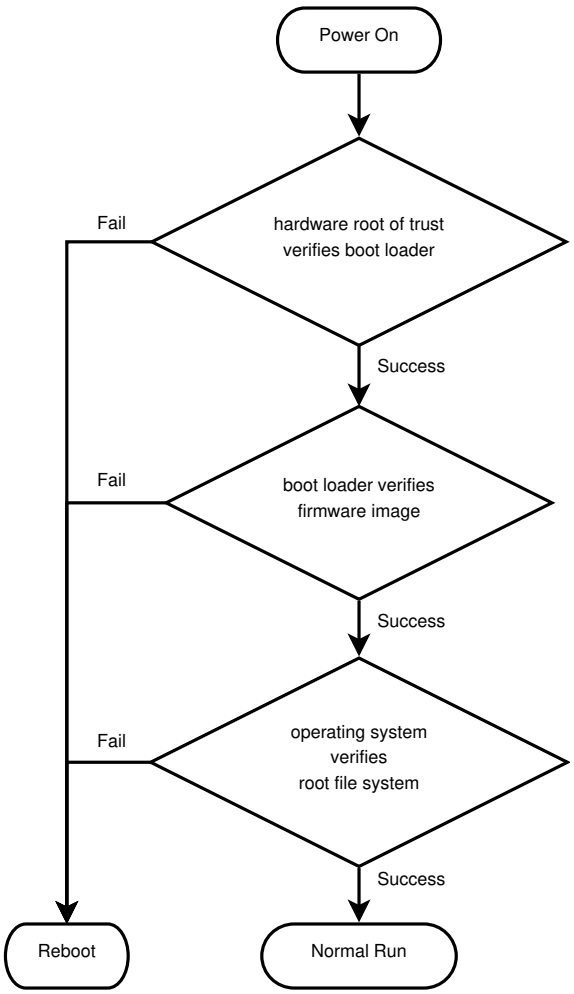

Figure 3.2. *Flowchart about the secure boot process*

With a chain of trust like this, the boot process is not only secure, but also modular, meaning that by possessing the right keys, the stages can be changed or updated without the need to modify the hardware. Having such a modular and secure boot process also provides a great base for a secure firmware update process.

3.3 Secure Firmware Update

A secure firmware update, that is also fail-safe, and prevents version rollback, is a complex process. So as with any complex problem, the best way to deal with, is to tear it down to simple parts. One of the parts is restarting the device with the updated firmware, therefore the previously described secure boot process provides a great base in the solution of that part. Obviously it must be completed with fault detection and fallback mechanisms, to be fail-safe, and with proper versioning, to prevent rollback attacks. As well as with a part, that is responsible for managing the updates.

There are two key components in a fail-safe firmware update process, the fault detection mechanisms, which are part of the live system, and the fallback mechanism, which ensures

that the device is operable, even after a faulty update. More precisely fault detection mechanisms must include the use of a watchdog, to detect a system hang, and some sort of a system self-test, to check whether the updated firmware is working correctly, or not. The system self-test must check every essential component needed for the correct operation of the device. And the fallback mechanism must provide at least one way to automatically downgrade the system to a previous working firmware, in case of a faulty update, be it a system hang or an error in the self-test or a verification failure during the secure boot process.

The fallback mechanism however, has to prevent version rollback attacks with the help of secure versioning. To achieve this, first of all, the version of the firmware must be bound cryptographically to the firmware, meaning that in an update, the version number is digitally signed together with the firmware. Furthermore, logs must be kept of the installed versions, and they must be checked during the secure boot process, so only a conforming firmware can be booted. According to the logs and to the version of the firmware, strict rules must be followed when booting a given firmware. If the version of the given firmware is higher than the highest previously logged version, meaning that it is an update, it can be booted. Else, if the version of the given firmware is equal to the previously logged version, meaning that it is the current up to date firmware, and its self-test was successful, it can be booted. And, if the version of the given firmware is lower than the previously logged version, meaning that it is a fallback, and the self-test of the updated firmware failed or there was a system hang with the updated firmware, it can be booted. In compliance with the above mentioned, the logs must include the version numbers and the self-test results of the booted firmwares.

The logs must also be checked by the update-manager, in order to download conforming firmwares only. In general, this means higher and not failed versions only, to prevent wasting resources on not conforming firmwares. Although there can be one exceptional case, when the update failed because of a verification failure, possibly caused by an accidentally damaged download instead of a faulty firmware. In this particular situation the given update could be tried again until a limited number of tries, which constraint is there to prevent wasting resources if the failure has an other source instead. Obviously the logs must have support for the described exception.

As might be expected by now, logs are parts of the core of the secure firmware update process, and must be kept secure. Because if any of them is damaged or missing, the device must not boot with any normal firmware automatically.

3.4 Security hardened firmware/operating system

In general a firmware/operating system is optimized for performance, perhaps for compatibility. In contrast, hardening focuses on optimizing the firmware/operating system for security.

In order to achieve a security hardened firmware/operating system numerous default settings have to be changed. Although it is most likely, that not every hardening option can be used for a given firmware/operating system, since some of them might conflict with other parts of the given firmware/operating system. Clearly, those, which cause required features to fail or to malfunction, cannot be selected, alternatively the feature-set has to be altered, though in most cases it is not an available option. Generally the hardening settings demand more resources compared to the default ones, so this can also be a constraint, especially in an embedded environment. All in all, when hardening a firmware/operating system the goal is to reach an acceptable trade-off between features, security and performance.

3.5 Remote Attestation and Integrity Monitoring

In this section we describe the design of our remote attestation and integrity monitoring implementation. Here we aim to show the architecture and high level operation of the created system along with the features of each component.

In general, the integrity verification of the running Linux system should be performed by an other system component than the OS itself, since it may be already compromised, making the result unreliable. This other system component must be running on the same or higher privilege level than the Linux kernel to ensure that Linux user mode applications do not interfere with the monitoring processes. The privilege level must be high enough to enable the integrity monitoring application to access the memory area of the Linux kernel and user mode applications. In our implementation these requirements are satisfied by a Trusted Application running in Secure World and by extending the OP-TEE kernel with functionalities that provide access to the entire OS memory for our TA.

3.5.1 Integrity Monitoring Techniques

The integrity verification code may perform a variety of anomaly detection functions on the Normal World, including the Linux kernel and user mode applications. For example the code might try to identify potentially malicious system components, running processes, suspicious network connections and activity. Shellcode and shared library injection could also be detected by looking for suspicious protection bits belonging to the memory regions of a process, as described in [27].

In our implementation the integrity monitoring consists of two techniques. First, the list of currently running processes can be extracted from the Normal World. This list can contain various meta information about the processes, for example their name, process identifier (PID), process start time stamp or the user identifier (UID) of the user who started the given process. This list then can be sent to the remote client requesting the monitoring task and the client can compare the list to a whitelist. Depending on the Normal World software ecosystem if the creation of a nearly complete whitelist is possible this method

could potentially detect malicious running processes. For example on a strict and closed system with limited number of known and trusted running processes, the listed processes which are not on the whitelist could be indicators of system compromise — although false positives have to be taken into account and handled correctly.

The second technique is to calculate hash values for the processes in memory. In our solution the hash is calculated for the code segment of the processes, where the compiled machine code resides. This enables the detection of malicious code injections into trusted or whitelisted or otherwise know processes. The replacement, modification or other form of tampering with program code in binaries stored on the system is also detectable. The remote client receiving the calculated hash value could have a database of known processes and their respective precalculated code segment hashes. Using this database the previous detection methods should be applicable, since any kind of modification in the code segment of the binaries causes the calculated hash values to differ from the precalculated hash values.

In the previous techniques a remote client is assumed, but the methods could also work with a local client, for example another Trusted Application.

3.5.2 Secure World Pseudo Trusted Application

In OP-TEE a normal Trusted Application can only access its own memory region. This separation and access control is the same concept as virtual memory in any operating system for example in Linux. For integrity monitoring tasks we need access to the Normal World memory from the Secure World. The GDP TEE specification and OP-TEE does not specify an API we could use to read from or perform any operation on NW memory content. Our solution is a Pseudo Trusted Application (PTA) which is in essence an extension for OP-TEE kernel—much like kernel modules in Linux—exposing the functions mentioned before. PTAs are running on the same privilege level as OP-TEE kernel, which is necessary to access the complete physical memory.

Physical memory addresses are used in the PTA to remain platform and implementation independent as much as possible. Therefore the caller first must translate NW (kernel and process) virtual addresses to physical ones. Using this API makes access to NW memory possible to normal TAs, possibly eliminating the need to rewrite those applications as PTAs. Writing large business applications which might contain many features as PTAs might not be recommended because of the following reasons. First, since they run with kernel privileges, any exploitable vulnerability in them might have a greater impact compared to normal TA vulnerabilities. Second, their development and maintenance might be more difficult, since they are compiled as part of the OP-TEE kernel, so making changes in them requires the replacement of the running kernel (either in emulated environments or on the physical development hardware).

There are two features implemented in our solution. First, the caller can request reading from a memory range by specifying physical addresses. The other is calculation of hashes for

memory ranges. Here the caller can request a singe hash for every given range or multiple separate hash for those ranges. The requested operation is performed and the result is passed back to the caller.

3.5.3 Secure World Trusted Application

The Secure World Trusted Application (TA) provides remote attestation and integrity monitoring services to Normal World clients. It processes the requests from the Server Application in NW and passes back the results. First, the TA verifies the authenticity and integrity of the requests using cryptographic signatures and only continues processing if the verification is successful. Next, the requested monitoring task is executed on the NW system. There are two types of monitoring tasks in our implementation, as mentioned in Section 3.5.1. The monitoring result is then cryptographically signed and the TA generates and sends the response using the result and signature.

The Trusted Application can not directly access the memory area of Linux or Normal World applications, however this level of access is required by the monitoring algorithms. The Pseudo Trusted Application is used for accessing and calculating hashes of the Normal World memory. Every time the TA reads or accesses the NW memory by any means it sends the PTA the address range or ranges for the required operation. The PTA executes the received operation and sends back the result to the TA.

The PTA operates on physical memory addresses, so the translation of NW virtual addresses must take place in the TA. There can be many memory management methods on different architectures, platforms, and CPU models, therefore the translation functions are highly platform dependent. Aside from that, the two monitoring algorithms shown here, mostly depend on the used kernel version, because kernel data structures can frequently change during kernel development. We created `dwarfparse`[8] for this very reason, however, it can only help with the structure definitions, but not with the algorithms using those structures.

The TA also includes a basic asymmetric cryptographic key pair provisioning solution. This provisioning solves the problem of installing the key pairs on the device storage securely, since hard coding them in the source code or storing them without any protection should be considered a bad practice. The keys are used in the remote attestation protocol for the verification and creation of signatures, i.e., the public key of the remote client and the private-public key pair of the device itself. The secure storage of the keys are of high importance, since they assure the integrity and authenticity of the incoming attestation requests and the outgoing responses (which contain the actual state of the device). The Normal World must not be able to modify or tamper with these keys without the Secure World at least noticing. We use the GPD Trusted Storage specification, which is implemented in OP-TEE. This implementation uses the NW storage to store objects encrypted

[8] See section 4.5.9

and integrity protected, therefore missing or modified keys are not used.[9]

3.5.4 Remote Attestation

Remote attestation is a method by which a device can authenticate its state to a remote host (client). This enables the client to determine the level of trust in the integrity of the device and its running configuration.

Attestation is performed in the TEE which ensures that the state of the device can be always reliably determined. The attestation protocol uses public key cryptography to verify the authenticity and the integrity of the messages. The public and private keys can be stored in the TEE Secure Storage, where it is protected from the REE.

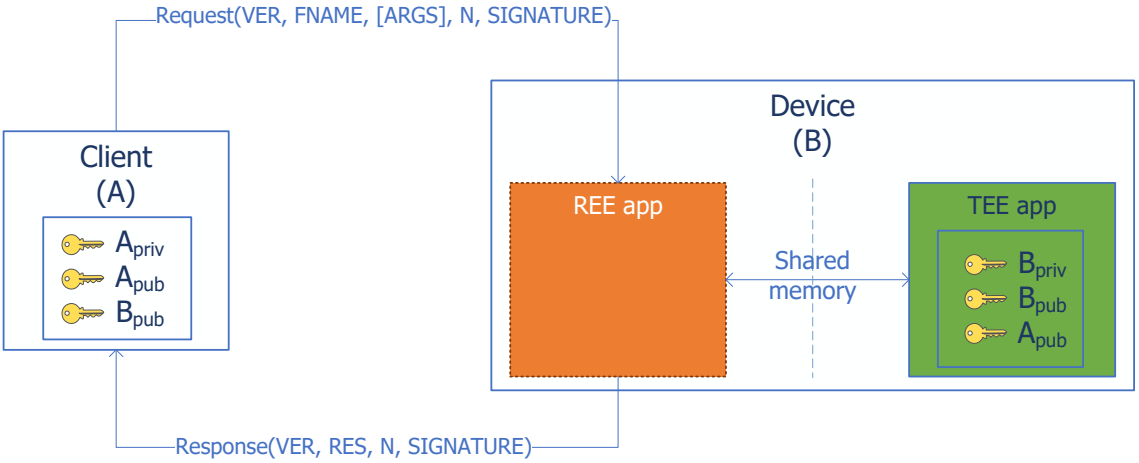

Figure 3.3. *Remote attestation architecture*

Protocol overview

The following steps are performed during an attestation session:

1. The client sends a digitally signed attestation request to the remote device.
2. The device verifies the request and performs the requested attestation method.
3. The device creates a digitally signed response and sends it to the client.
4. The client verifies the response.

Attestation techniques

Various attestation methods can be performed, which can include listing running processes, measuring the integrity of the executables loaded into the memory, and checking hardware and software configuration.

[9] `https://github.com/OP-TEE/optee_os/blob/master/documentation/secure_storage.md` (last visited 2018–10–21)

We implemented two methods, firstly getting the list of the running processes by inspecting the Linux kernel data structures from the TEE, secondly checking the integrity of a running process, by calculating a checksum of the process' code segment which is loaded into the memory. These two methods allow us to create a simple attestation technique by using a white-list to detect unwanted or modified programs on the device.

3.6 Secure Communication

3.6.1 PKCS#11

PKCS#11 was originally one of the Public Key Cryptography Standards, that are a set of standards created by RSA Laboratories, but in 2013, further development has been turned over to the OASIS PKCS 11 Technical Committee[10]. PKCS#11, also referred to as Cryptoki, is a general purpose API for accessing devices capable of storing cryptographic keys and executing cryptographic operations [31]. Some examples of such devices are smartcard security tokens, hardware authentication tokens (e.g., Yubikey) and cryptographic hardware security modules (HSMs). PKCS#11 is widely used in the industry, and as a result, a lot of applications support it, including, but not limited to OpenSSL[11], OpenSSH[12], Mozilla Firefox and Mozilla Thunderbird through Mozilla NSS[13], OpenDNSSEC[14]. There are also several tools that provide a user interface to manipulate tokens (e.g.: generate, import, export keys, encrypt or decrypt data), for example the pkcs11-tool that is part of the OpenSC project[15]. It is important to note, that a device may only support a subset of the mechanisms defined in the standard and it would be still compliant, so some tokens may not work with all of these applications, depending on what API calls they support.

3.6.2 Relevant security guarantees

In the PKCS#11 terminology a token is the device that stores the cryptographic keys, certificates, data, etc. The data that is stored on the token is organized into objects (e.g., a private key object) and can be accessed through handles, that can be likened to the pointers we use in programming. A slot is what we can access a token through (e.g., a smartcard reader). Certain objects can only be used if the client logs in to the token, by providing a PIN. Regardless of whether a user is logged in, if an object has the sensitive flag set, it shall not be revealed off the token, so for example it can't be viewed with

[10] `https://www.oasis-open.org/news/pr/oasis-enhances-popular-public-key-cryptography-standard-pkcs-11-fo` (last visited 2018. 10. 21.)

[11] `https://developers.yubico.com/YubiHSM2/Usage_Guides/OpenSSL_with_pkcs11_engine.html` (last visited 2018. 10. 21.)

[12] `https://github.com/OpenSC/OpenSC/wiki/OpenSSH-and-smart-cards-PKCS%2311` (last visited 2018. 10. 21.)

[13] `https://developer.mozilla.org/en-US/docs/Mozilla/Projects/NSS/PKCS11/Module_Installation` (last visited 2018. 10. 21.)

[14] `https://www.opendnssec.org/softhsm/` (last visited 2018. 10. 21.)

[15] `https://github.com/OpenSC/OpenSC/wiki` (last visited 2018. 10. 22.)

the `C_GetAttributeValue` mechanism, but it may be exportable if wrapped with another key. If an object has the unexportable flag set it shall not be exportable at all, and the unexportable flag shall not be modifyable either. These guarantees have to hold, even if the token connected to a slot in a malicious machine. [31]

3.6.3 Example usage

Usually the client applications access the PKCS#11 API through shared objects, or dynamically linked libraries based on the operating system. For example, if someone wants to use her smartcard to authenticate to a server with OpenSSH, she would have to find the shared object that is able to communicate with her smartcard. The manufacturer should provide this, or the OpenSC project could be a good place to start looking. Once she has the shared object it can be specified with the `-I` command line flag. Internally OpenSSH includes the `pkcs11.h` header, and calls the functions declared in it. When we specify the shared object, it dynamically links the actual hardware specific implementation for the exact card we want to use.

3.6.4 Soft tokens

It is not necessary for the token to be implemeted as a hardware component, it can also be implemented completely in software. Of course this way the security guarantees can't be enforced, but it is useful for developing and testing applications. These were also helpful for us as reference implementations while writing code for this project. First, SoftHSM[16], which is developed as part of the OpenDNSSEC project. OpenDNSSEC uses the PKCS#11 API to handle and store its cryptographic keys. The purpose of SoftHSM is to allow OpenDNSSEC to be used if the user can't afford, or simply doesn't want to use a hardware token. Another example of a soft token is part of the openCryptoki project[17], and its supposed to be used for testing.

3.6.5 SKS

Secure Key Storage (SKS) is a proposal[18] by Etienne Carrière for a token that is implemented as an OP-TEE TA, and accessible through the PKCS#11 API. It is currently under development, when we first started working with it, it only supported AES mechanisms, at the time of writing there is also partial support for RSA and EC mechanisms. The project consists of three parts: the TA implementation of the token, the shared object that provides access to the token from normal world and the regression tests that are integrated

[16] `https://www.opendnssec.org/softhsm/` (last visited 2018. 10. 21.)

[17] `https://github.com/opencryptoki/opencryptoki` (last visited 2018. 10. 21.)

[18] `http://connect.linaro.org.s3.amazonaws.com/hkg18/presentations/hkg18-402.pdf` (last visited 2018. 10. 22.)

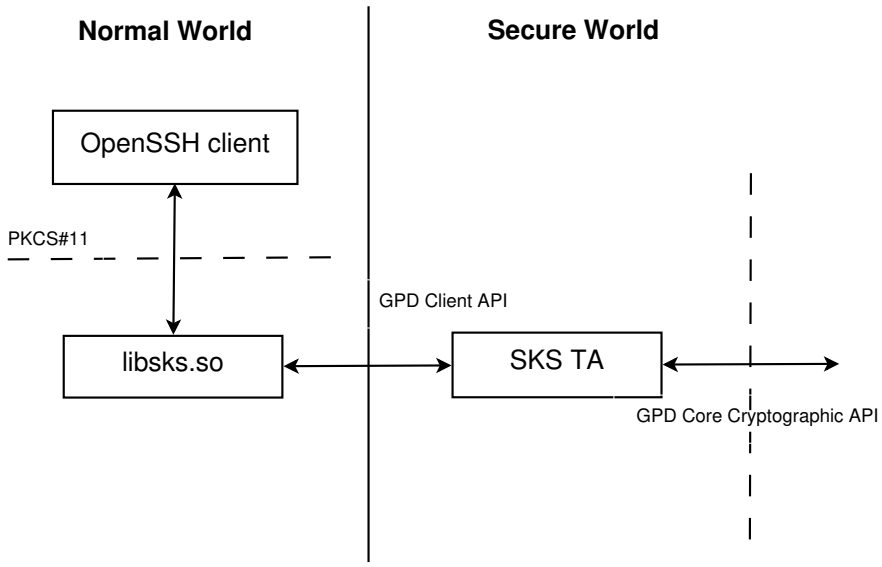

Figure 3.4. *SKS arhcitecture*

with the OP-TEE xtest test suite. The project can be found on GitHub under the user etienne-lms in the `sks` branches of the forks of OP-TEE repositories.

When there is a PKCS#11 call, the SKS shared object converts the constants from the PKCS#11 values to the ones used by SKS internally and serializes the data, so it can be passed through the OP-TEE TA call interface. In the TA, the data is deserialized, and the requested operations are executed. Most of the time the PKCS#11 mechanisms can be easily translated into GPD TEE Core API calls, so the TA heavily relies on these calls.

The regression tests are based on the tests for the OP-TEE Internal Core API cryptographic methods. These are implemented through a test TA that basically just forwards the calls to the OP-TEE kernel and returns the results. For the SKS tests these calls are not called through the test TA, but SKS. There are also functionalities in SKS — for example `C_Login` — that cannot be tested this way, these have their own tests too.

3.6.6 Architecture

Building on the above described components, we can build a system, where cryptographic keys are strongly protected. The TrustZone technology provides two separated execution environments that can only communicate through well defined interfaces. The lower priviliged environment, called the Rich Execution Environment (REE) or Normal World can house a conventional operating system (i.e. Linux) and the higher priviliged one, called Trusted Execution Environment (TEE) or Secure World, holds a trusted operating system, in our case OP-TEE. Inside OP-TEE, we have the SKS, utilizing the separation between the two worlds and store cryptographic keys in OP-TEE's secure storage and provide the PKCS#11 interface to Normal World. Using the PKCS#11 API we can use the keys stored in SKS by any client application that supports the interface, while that application can never directly access the key itself. Even in a scenario, where a Linux application with root

priviliges is compromised and it can recover the PIN for the SKS token (e.g.. by replacing the shared object with a malicous implementation, that reveals the PIN), the attacker can only use the keys stored on it, but not recover and steal them.

In this paper, we choose OpenSSH as the client application. This allows us to copy files with `scp`, connect to a server and run scripts, or even provide a remote shell with ssh reverse tunneling (the reverse tunneling is necessary, because the `-I` option is only available in client mode). OpenSSH is also a nice choice, because it is easy to set up compared to a VPN and can be easily tested by calling `ssh root@localhost`.

Chapter 4

Implementation

4.1 Environment

4.1.1 Raspberry Pi 3 Model B

The above discussed design plans were implemented on the Raspberry Pi 3 Model B. This platform was chosen, due to its widespread use and to the large software development community it has. It is also important, that it has the Broadcom BCM2837 System on a Chip (ARMv8) in it, which supports the ARM TrustZone technology. Unfortunately the hardware is poorly documented, and it misses a lot of security functions. As a result some of the implementations are only proof of concept.

4.1.2 QEMU

QEMU[1] (Quick EMUlator) is a machine emulator, it can emulate a target system and run unmodified operating system and programs designed for it, regardless of the host system's CPU architecture and operating system. QEMU can be run on all the major operating systems (i.e., Linux, Windows, Mac OS). In our case, using QEMU came with multiple advantages. The first and obvious one is that this eliminates the need to have access to hardware for development. Other than that, the development process is also faster and more convenient this way: the whole project can be built by issuing one make command, no need to flash SD Cards, or be near a physical device at all. Opening the two different consoles (to Normal and Secure World) is as easy as creating two `screen` sessions.

4.1.3 U-Boot

Das U-Boot or The Universal Boot Loader [14] is an open source boot loader for many different architectures, and it is mostly used in embedded devices. U-Boot operates on a

[1] `https://www.qemu.org/` (last visited 2018. 10. 22.)

quite low-level and provides a low-level interface through its commands, yet this makes it suitable for plenty of devices. Its settings are stored in environment variables, and it has a shell command interpreter. These enable scripting, therefore any custom boot program can be created conveniently.

U-Boot verified boot

> U-Boot 2013.07 introduces a feature allowing for the verification of a kernel and other images. This can be used to implement a form of secure boot which we will call "verified boot" [...]. U-Boot's new verified boot feature provides a mechanism for verifying images while still allowing them to be field-upgraded. It fits in seamlessly with the existing image loading infrastructure in U-Boot. [19]

4.1.4 Build System

repo

Repo[2] is a repository management tool originally built for managing the Google AOSP project. It allows collecting the necessary code from multiple git repositories by specifying them in a manifest file. Multiple manifest files can be provided for different configurations: for example in OP-TEE there are different manifests for the different target architectures. It doesn't replace git, rather its goal is to make working with source code organized into numerous git repositories easier.

For OP-TEE, the manifests are stored on github, in the `OP-TEE/manifest.git` repository. We used the `default.xml` to create a basis for developing SKS. Following the instruction in the `OP-TEE/build.git` repository, we could easily produce a working Linux plus OP-TEE system running in QEMU. Of course, this system didn't have the SKS and some other tools that we needed yet.

Buildroot

Buildroot[3] is a tool that aims to make cross compilation and building custom Linux systems for embedded devices easier. It is basically a collection of makefile scripts and can be configured through various interfaces, e.g., menuconfig, just like the Linux kernel. As an output, buildroot produces a whole root filesystem, ready to be flashed onto the device and used. There is no package manager in the system created, so updating a single program is only possible by updating the whole system. There are many packages available in buildroot that can be included in the build, by simply selecting them in menuconfig, however it is

[2] `https://github.com/aosp-mirror/tools_repo` (last visited 2018. 10. 22.)
[3] `https://buildroot.org/` (last visited 2018. 10. 22.)

also possible to create external packages, if something is needed that is not already part of buildroot. This is done by creating two new files: first, the `Config.in` file describes how the package looks in menuconfig and what other packages it depends on and the `package_name.mk` file describes how to acquire the source and build it. There some other files that can be included (e.g.: `patchname.patch`, `package_name.hash`), but these two are essential.

Recently, OP-TEE has started to integrate its build process into buildroot, so adding the SKS and other needed tools had to be done through buildroot. We added the client and test part of sks, by simply replacing the source of optee_client and optee_test with the sks branch of the GitHub user etienne_lms. The SKS TA was added as an external buildroot package. OpenSSH was already part of buildroot, but the OpenSC pkcs11-tool had to be added as and external package as well.

Raspberry Pi

In our environment for developing every solution other than SKS, Buildroot is used as a build system for the Raspberry Pi 3. We integrated the building of Linux kernel, U-Boot boot loader, various software packages and the creation of SD Card images. The device and component configuration files (e.g., configuration for the kernel and U-Boot) can be easily deployed with Buildroot. We also used the packages provided by the OP-TEE community—packages for the OP-TEE kernel, tests and client running in Linux. We created packages for our Trusted Application, remote attestation server application and our auxiliary software. Some necessary patches were applied for existing Buildroot packages, to be able to use their latest releases.

4.2 Secure Boot

The first stage in a secure boot process is the hardware root of trust, but as already mentioned, the Raspberry Pi platform misses a lot of security functions, including the hardware root of trust. Therefore, in this proof of concept implementation no hardware root of trust is used, and U-Boot is considered trusted, thus becoming the root of trust.

U-Boot supports booting a Flattened Image Tree (FIT). The U-Boot documentation says the following about FIT images:

> It is a flattened device tree (FDT) in a particular format, with images contained within. FIT supports hashing of images so that these hashes can be checked on loading. This protects against corruption of the image. However it does not prevent the substitution of one image for another. The signature feature allows the hash to be signed with a private key such that it can be verified using a public key later. Provided that the private key is kept secret and the public

> key is stored in a non-volatile place, any image can be verified in this way. The public key can be stored in U-Boot's CONFIG_OF_CONTROL device tree in a standard place. Then when a FIT is loaded it can be verified using that public key. Multiple keys and multiple signatures are supported. [16, 15]

For the hashing of images SHA-1, and for the signing 2048-bit RSA key pairs are supported.

In order to verify the next stage in the chain of trust, the signing 2048-bit RSA key pair has to be generated. OpenSSL can be used to create this key pair. U-Boot verifies the next stage with FIT image verification and is hard-coded to only load FIT images that are signed with the correct key.

In this implementation the FIT image builds up from the Linux kernel with an initial RAM file system, the flattened device tree and OP-TEE. All of the nodes are hashed with SHA-1, and the whole configuration is signed with the generated RSA private key. This method ensures, that the whole FIT image is signed, and none of its parts can be modified without breaking the signature.

The RSA public key is included in U-Boot's control device tree, so U-Boot can verify the image during boot. Then U-Boot is configured and built with that control device tree in a way, that it only boots FIT images that are signed with the included RSA public key.

Provided that OP-TEE and Linux are booted the next step is the verification of the root file system, which is done by dm-verity. The following is a quote from dm-verity documentation.

> Device-mapper is infrastructure in the Linux kernel that provides a generic way to create virtual layers of block devices.
>
> Device-mapper verity target provides read-only transparent integrity checking of block devices using kernel crypto API.
>
> [...]
>
> The dm-verity was designed and developed by Chrome OS authors for verified boot implementation. [12]

The hash of the root file system can be created by veritysetup (a tool for dm-verity). MD5, SHA-1, SHA-256 and SHA-512 are supported, and the default is SHA-256. Then the hash is placed into the initial RAM file system, that is linked into the Linux kernel, as a result it is protected by the signature on the FIT image.

With the help of veritysetup the operating system hashes the root file system and compares the result with the hash in the initial RAM file system, and only switches from the initial RAM file system to the root file system and continues to run, if the two hashes are identical. This functionality is implemented in the init scripts.

Since the root file system is read-only, write-enabled file systems have to be mounted in order to store persistent application/user data. The verification of such mounts is the responsibility of the respecting applications.

4.3 Secure Firmware Update

Depending on the use cases, there are different approaches to the implementation of a secure firmware update. One aspect might be whether the complete firmware on the device is updated — meaning that the update is atomic — or the individual components of the firmware can be updated separately. One other aspect might be whether there can be more than one version of the firmware on the device at the same time, so it can easily be reverted to the previous working version in case of a faulty update, or strictly only one version at any given time, so it is more difficult to revert but easier to manage. One different aspect might be whether the main operating system updates the firmware, or there is a dedicated simplified operating system to manage the updates, which case is more complex but also more secure. One additional aspect might be whether there is a fallback operating system and/or an interactive boot loader, in case of a faulty update with no way to revert it, or not, and in this given scenario the device is intentionally inoperable. The above mentioned aspects are independent from each other, and so they can be freely combined to achieve the desired use cases.

A significant decision is, whether to update the complete firmware on the device, so the update itself is atomic, or to provide the possibility to update the individual components of the firmware separately. From here on, the complete and atomic firmware update process is discussed, because it suits an embedded device better. Such an update must be well tested, therefore it will be more stable and secure. In most cases there is no need for individual component update, which is more complex, further reinforcing our decision.

In case of an update there can be two different versions of the firmware on the device at the same time — the newer, updated, but not yet tested one, and the older, stable one. In this way, the device can easily be reverted in case of a faulty update. Once the update is tested however, it becomes the stable version, and no other version is allowed to be on the device until another update. This is a good trade-off between easy reversibility and easy manageability with rollback-protection.

In this proof of concept implementation the main operating system updates the firmware with the use of different partitions. Another method could be using a dedicated simplified operating system, which manages the updates.

In this given model there is no additional fallback operating system and no interactive boot loader. Though it is constructed in a way, that any of them can easily be added.

The so far described firmware format is enhanced with versioning. The previously implemented FIT image is extended with a new version node, that adds the version-number of the firmware to the image, so that it is also signed together with the firmware. The version-number of the firmware is in binary format and two bytes long for the major and minor version-numbers. The version-number is also placed both into the initial RAM file system and into the root file system, therefore only the matching root file system can be loaded with the firmware, as the operating system checks the version-number as well.

The format of the boot-log file is defined as follows. An entry starts with three fixed bytes, all in binary format, because of the low-level operation of U-Boot. The first two bytes are there for the major and minor version-numbers of the firmware, and the third byte is the result of its self-test. When the third byte is 0x00 it indicates that everything is OK with that version, anything else indicates an error. 0x01 in particular means that the verification of the integrity of root file system failed, so the integrity of the given version is broken. After the first three fixed bytes the entry can freely be expanded. Newer entries are prepended to the boot-log, because in U-Boot it is much easier to read from the beginning of a file than from the end of it.

With scripts, stored in the environment variables of U-Boot, a modular boot program is implemented, which works as described below. It tries to load the updated firmware, then checks the boot-log and the watchdog as described in Section 3.3, then tries to boot it according to the rules of verified boot. If there is no updated firmware or any of those steps fails, it continues with the stable firmware. It tries to load the stable firmware, then checks whether the verification of the updated firmware failed, or not. If it did, it adds an indicating parameter to the boot-arguments then continues, else it just continues. Then checks the boot-log as described in Section 3.3, then tries to boot it according to the rules of verified boot. Again if there is no stable firmware or there was a failure, it prints a message, saying that the device is bricked, and resets the device.

As everything is implemented modularly in the above detailed boot program, it is relatively easy to extend it with a fallback firmware at the end of the chain, or with an interactive boot loader, as the latter can be enabled with the help of an environment variable.

The rest of the secure firmware update process is implemented in the operating system, by extending the init scripts already implemented in the secure boot process. First of all, if the verification of the root file system fails, the boot-log is written accordingly (error-code 0x01) before the reboot, so that U-Boot can know that the integrity of the given firmware is broken. If the verification of the root file system was successful and the operating system switched to it and performed other necessary initializations, the kernel command line — that can be specified with the boot-arguments — is checked for the indicating parameter of an integrity failed update firmware. On the condition that it is there, the update firmware is deleted and logged in a way, that the update-manager knows about the integrity failure of that specific firmware. After this, or if the indicating parameter is not there, the self-test is executed. As an example the xtest test suite of OP-TEE is used as self-test. In the event that the self-test fails, the boot-log is written accordingly (with the failures matching error-code) and the device is rebooted. Else the boot-log is written with code 0x00, meaning that everything is OK with the current firmware. If there is, any different firmware is deleted and its version-number logged, so that the update-manager never tries to download it in the future. Then the update-manager is started, and the device starts its normal operation.

The update-manager periodically checks for updates, and only downloads such firmwares, which conform to the logs. In case of a conforming update the update-manager downloads it, and schedules a reboot.

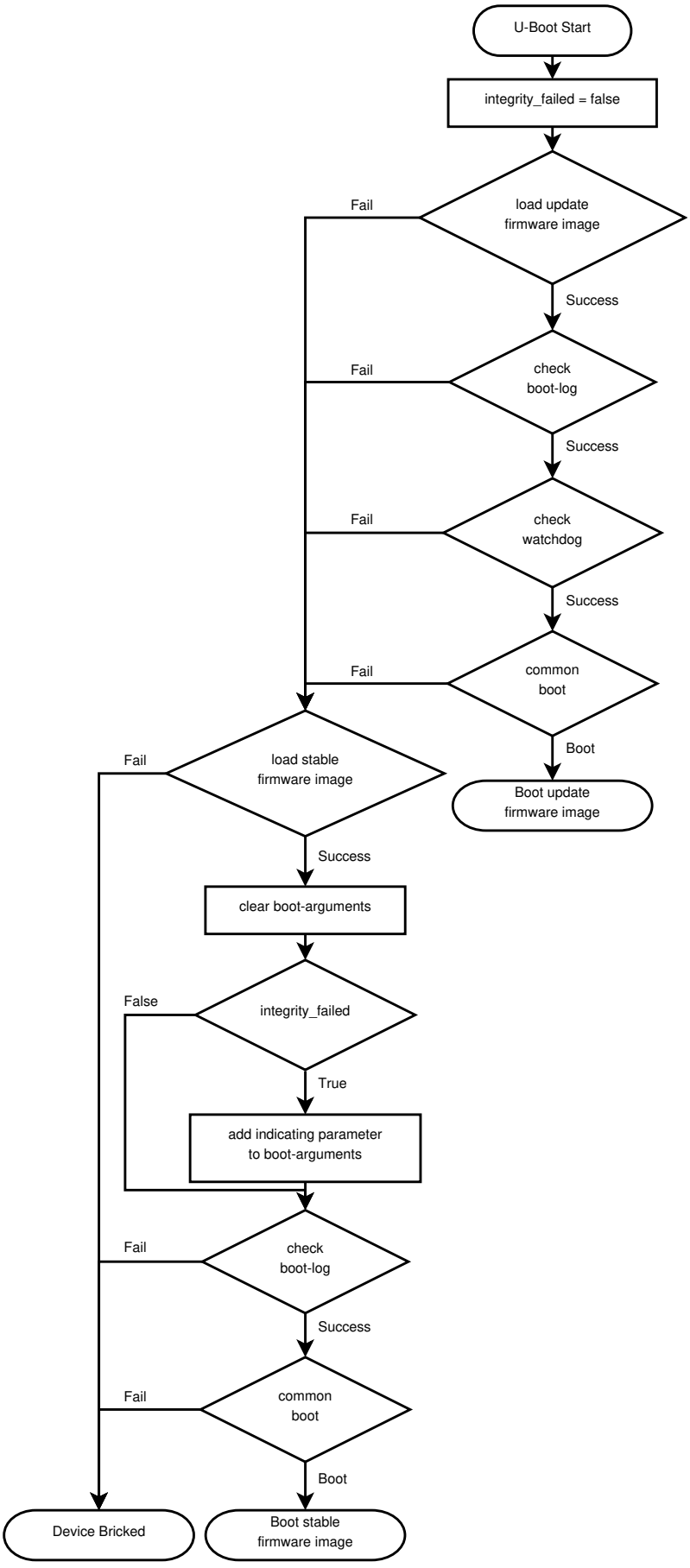

Figure 4.1. *Flowchart about the main control flow in U-Boot*

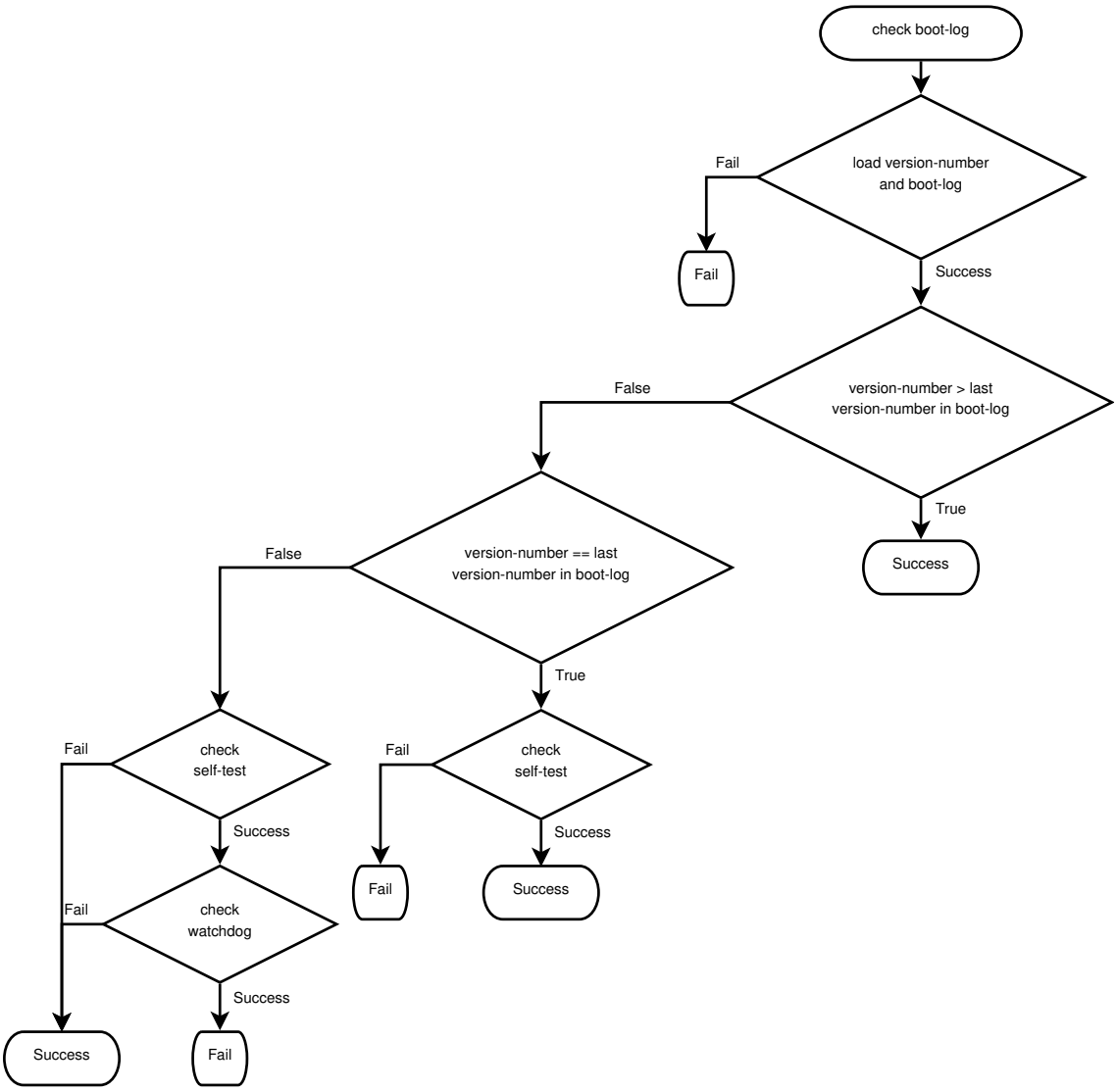

Figure 4.2. *Flowchart about the check boot-log part*

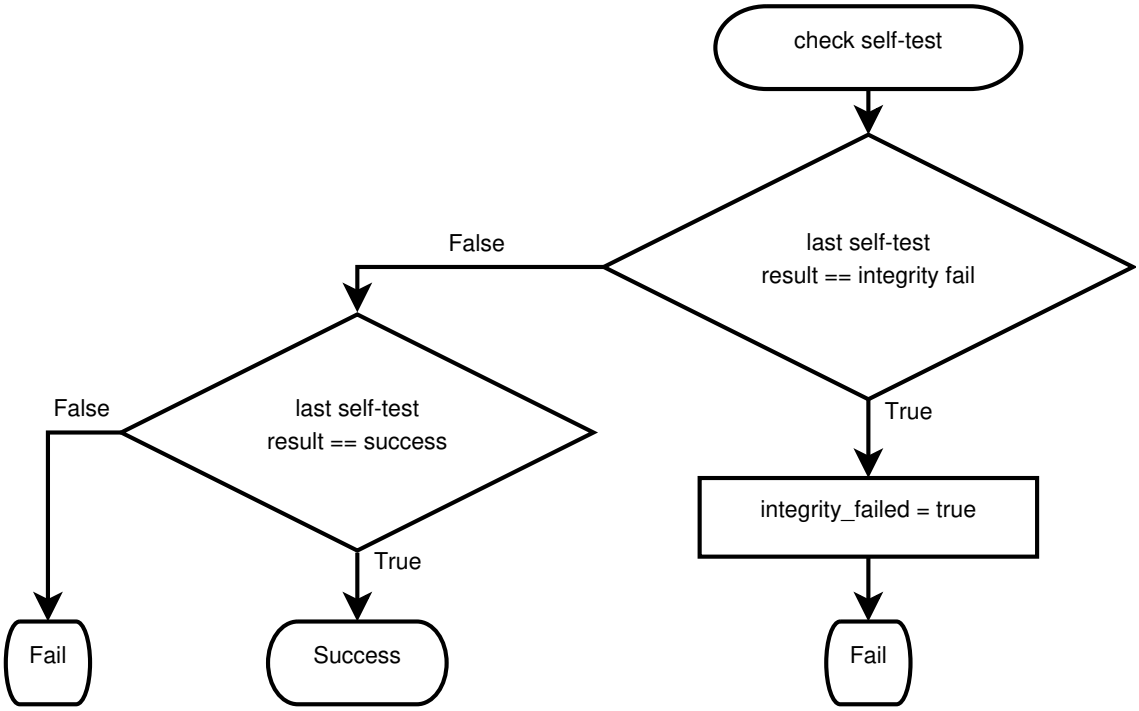

Figure 4.3. *Flowchart about the check self-test part*

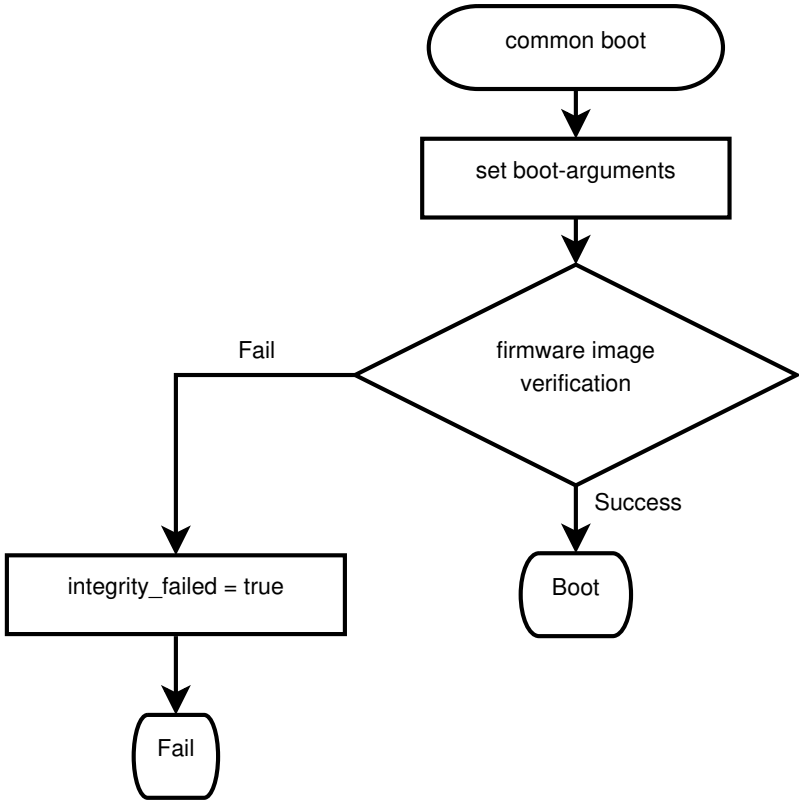

Figure 4.4. *Flowchart about the common boot part*

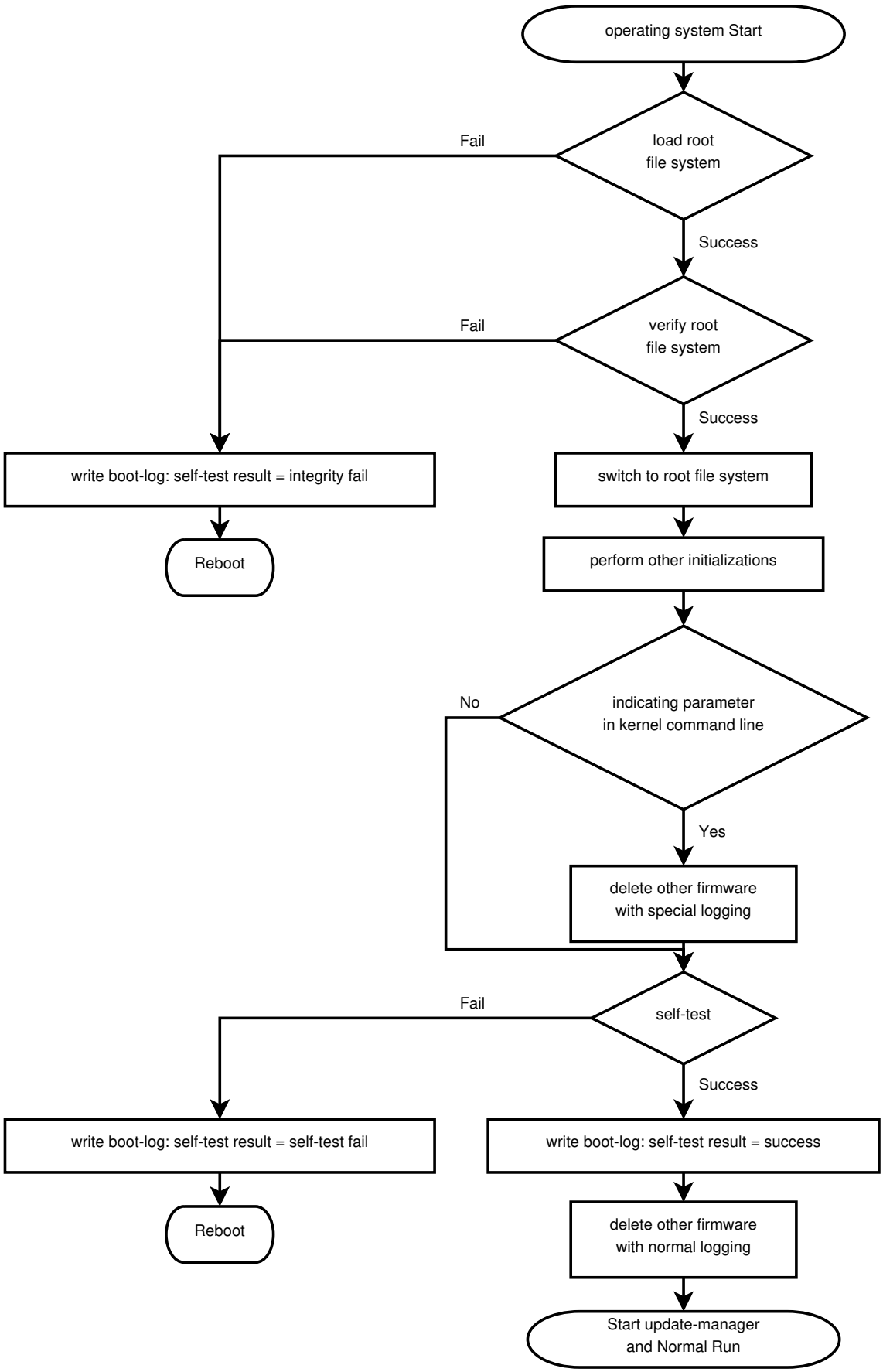

Figure 4.5. *Flowchart about the main control flow in the operating system*

4.4 Security hardened firmware/operating system

As a matter of fact numerous hardening features already exist in Linux based systems, along with a project, that aims to gather, improve and build them into the official Linux kernel. That project is called the Kernel Self Protection Project, and it has the following Mission Statement.

> This project starts with the premise that kernel bugs have a very long lifetime, and that the kernel must be designed in ways to protect against these flaws. We must think of security beyond fixing bugs. As a community, we already find and fix individual bugs [...]. Those efforts are important and on-going, but if we want to protect our billion Android phones, our cars, the International Space Station, and everything else running Linux, we must get proactive defensive technologies built into the upstream Linux kernel. We need the kernel to fail safely, instead of just running safely.
>
> These kinds of protections have existed for years [...]. For various social, cultural, and technical reasons, they have not made their way into the upstream kernel, and this project seeks to change that. Our focus is on kernel self-protection, rather than kernel-supported userspace protections. The goal is to eliminate classes of bugs and eliminate methods of exploitation. [34]

The Kernel Self Protection Project has a recommended settings page [35] about various kernel build CONFIGs and run-time settings, such as kernel command line options and sysctls. The contents of that page is the base of this hardening implementation.

The kernel build CONFIGs can be added to Buildroots configuration, in order to build a hardened kernel. The kernel command line options can be added to the boot-arguments in U-Boot, in order to boot with hardening options enabled. And the sysctls can be set by the init scripts in run time.

In the first approach the incompatible ones were excluded, in particular those, that are meant for different architectures, those, that are not supported by the implemented system, for example by U-Boot, and those, that would break dependencies. Then it was followed by a series of tests to exclude ones, that caused errors in the live system. At the end of the process a security hardened operating system was achieved, while still supporting the desired features.

4.5 Remote Attestation and Integrity Monitoring

In this section we present the implementation details for our approach to Remote Attestation and Integrity Monitoring on the Raspberry Pi 3 Model B embedded device. The system components are shown similar to Section 3.5, only on a lower abstraction level, and with more details.

4.5.1 Secure World Trusted Application

In the following sections, we describe the various problems and challenges that have arisen during the development of the Trusted Application—and also the solutions we realized. We present these mostly focusing on their impact on integrity monitoring and remote attestation.

The remote attestation and integrity monitoring functionalities are implemented in one TA, since we did not find a solution in OP-TEE, mature enough, to satisfy our requirements of inter process communication between Trusted Applications using shared memory.[4]

Requirements

The auxiliary system components we created (shown in Section 3.5), are required for the proper functioning of the TA. As described in Section 4.1.4, we created a Buildroot environment with many tasks automated—including building the following required applications—however the satisfaction of other requirements do require manual configuration (e.g., configuring memory mappings in OP-TEE kernel).

First, the Pseudo Trusted Application is required to access the NW Linux memory from the TA. The PTA must be built into OP-TEE kernel, and the target memory regions (that we want to access) must be mapped in the Memory Management Unit of OP-TEE. The details of the configuration is described within section 4.5.2.

Second, after accessing the memory, we have to interpret the contents of it. To achieve that, we need the exact definitions of the data structures Linux stores there, i.e., the C header files matching the running kernel version. These header files contain many type and structure definitions, for example the `task_struct` structure. The `task_struct` represents a running process in the NW memory and contains many process related information, e.g., a pointer to the Virtual Memory Area (VMA) linked list of the process. These data structures are discussed later in Section 4.5.1. These header files can be obtained by running the `dwarfparse` auxiliary script on a dummy kernel module compiled for the used Linux kernel. In our implementation a single C header is generated from the dwarf debug information of the compiled module, containing every type definition and structure needed to access the NW memory properly. We also created a Buildroot package for running the script, generating the header file, and sharing it with the TA package in Buildroot, so it can be used when compiling the TA. See section 4.5.9 about the script details.

Third, the demand paging functionality of modern Linux kernels must be dealt with. Demand paging means that the binary of a running process is not completely loaded into RAM when the kernel starts the process. Then, during execution of the application, when the program flow reaches an area of the code segment of the binary that is not loaded, a page fault happens. The kernel then loads the page corresponding to the faulty address

[4] `https://github.com/OP-TEE/optee_os/issues/1068` (last visited on 2018–10–21)

into memory. This behavior has significant performance benefits, but it is undesirable when hashing the code segment of a larger binary. The main problem is that causing NW page faults from the SW is not easily achievable, since the two worlds use different memory management data structures and methods. Also, the SW is not able to request services from the NW, since this functionality is not implemented in OP-TEE.[5] In our solution a NW Linux kernel module (`mod-pslist`) is used to cause page faults for the code segment of each running process, essentially force loading the binaries into memory, so later the TA can access those pages without problems. The NW Server Application calls this module each time a hashing request is received from the remote client.[6] We also created a Buildroot package for the kernel module, and the module should be loaded at system boot time. This is implemented in the Buildroot root overlay with System V style initialization scripts.

NW-SW Interface

The interface that the TA provides, is quite flexible and does not restrict the usable protocols between the NW Server Application and the remote client. For example, one could use simple sockets or gRPC with TLS or other Machine to machine protocols for the Server Application – remote client communication. In our implementation we choose gRPC. For more details see section 4.5.2. The only coupling or restriction between the TA and the remote client might be that the signing and verification cryptographic algorithms used must match on both communicating sides, however this originates from the requirement that the integrity and authenticity of the requests and responses between the two parties must be assured.

We use shared memory between the NW and SW for the data transfer, since this is the technique OP-TEE implements and uses for NW-SW communication.[7] The NW Server Application provides the input and output buffers for the operation, by registering or allocating shared memory.[8] The interface supports three different commands. Each command has a command identifier (command id), that is a simple integer number, defined in a shared header file accessible for both the TA in Secure World and the Server Application in Normal World. The Server Application sends this id to the TA, specifying the requested operation, as described in the GPD Specification [21]. Along with the id, an operation argument format is also specified in the GPD standard. There are four usable arguments, which can be (but are not limited to) simple integers or fixed size buffers whose structure we can define [21, 22].

We specified the arguments the following way. Two arguments correspond to the request

[5] `https://github.com/OP-TEE/optee_os/issues/2333` (last visited on 2018–10–19)

[6] An alternative solution could be to calculate and send hashes for every page separately. This way if one of the pages is not loaded or swapped out, the code segment still could be partially checked.

[7] `https://github.com/OP-TEE/optee_os/blob/master/documentation/optee_design.md#7-shared-memory` (last visited 2018–10–22)

[8] Registering shared memory (SHM) means, that an already existing memory buffer (located either on the heap, stack, or defined as global) is used as SHM buffer, whilst when allocating shared memory, a new memory area is allocated in the registered SHM region of the RAM

and two to the response. The first GPD argument of the request contains the metadata for the requested integrity monitoring operation. This includes the signature of the request, the number of elements in the second GPD argument buffer, the version of the communication protocol in use, and a nonce[9] value. For the exact definition of the metadata structure see Listing 4.1.

The second GPD argument of the request is the parameters for the integrity monitoring operation, simply serialized as an array of zero terminated strings. For example, when calculating process hashes the process can be specified here (the number of strings in this buffer is included in the first GPD argument as mentioned before). We choose this serialization method because it is simple but could be powerful enough for many kind of different parameter formats, and is similar to how command line arguments are passed to the main function in C and C++ (`buf_num` being `argc`, the buffer itself being `argv`).

The third GPD argument (first for the response) consists of the same structure definition like the first GPD argument, since the format of the requests and responses are quite similar (both are signed, have the same signature length, and the nonce and version must be present in both). It also contains the signature of the response, and the number of strings serialized in the fourth GDP argument.

The fourth GPD argument (second for the response) is the result buffer of the integrity monitoring algorithms. In our implementation this could be either the list of running processes, or one or more hash values for the memory of the requested process. The number of serialized strings can be found in the response metadata as mentioned before.

A more formal description of the arguments above can be found in Listing 4.2. The `MEMREF_INPUT` and `MEMREF_OUTPUT` declarations matches the GPD specification [22]. `MEMREF_INPUT` is a memory reference buffer and is considered input only, from the point of view of the TA. `MEMREF_OUTPUT` is a memory reference buffer and is considered output only, from the point of view of the TA.

```
#define RA_SIGNATURE_SIZE 256
struct ra_cmd_meta {
        uint8_t sign[RA_SIGNATURE_SIZE];
        uint32_t version;
        uint64_t nonce;
        size_t buf_num;
};
```

Listing 4.1. *Metadata structure for operation*

```
GPD TEE param#0: MEMREF_INPUT
  .buffer: [ struct ra_cmd_meta ] metadata for the request
  .size: [ size_t ] should be sizeof(struct ra_cmd_meta)
GPD TEE param#1: MEMREF_INPUT
```

[9]The nonce is a random 64-bit integer value generated by the remote client and sent with the request. The TA includes it in the response to prove that the response corresponds to the request.

```
  .buffer: [ uint8_t[] ] request arguments: zero terminated strings serialized
  .size: [ size_t ] size of the argument buffer
GPD TEE param#2: MEMREF_OUTPUT
  .buffer: [ struct ra_cmd_meta ] metadata for the response
  .size: [ size_t ] should be sizeof(struct ra_cmd_meta)
GPD TEE param#3: MEMREF_OUTPUT
  .buffer: [ uint8_t[] ] operation results: zero terminated strings serialized
  .size: [ size_t ] size of the result buffer - number of bytes written by
      the TA operation
```

Listing 4.2. *Formalized shared argument definitions*

The interface exposes three commands: first, to get the list of the running processes of the NW Linux the corresponding command id is `RA_CMD_GET_PSLIST`. Input arguments are not required, the input buffer should be NULL. For each running process a zero terminated string is serialized into the output buffer. One such string contains the full file system path where the given process is located. This is preferable to the process name, for the reasoning behind this see section 4.5.1. The running kernel threads are not returned, since those do not have a file system path value. Also, their code segment is part of the kernel code segment, therefore would not make a good candidate to be input arguments for the process hashing command. The command itself also has a return value, which is a flag indicating the state of the operation. There are return values that are common to each integrity monitoring command, these can be found in Table 4.1.

`TEE_ERROR_BAD_PARAMETERS`	The provided parameters does not match the specification.
`TEE_ERROR_SIGNATURE_INVALID`	The provided signature in the request metadata is invalid.
`TEE_ERROR_ITEM_NOT_FOUND`	Path not found in the VMAs of a process while iterating over the process list of the NW Linux.
`TEE_ERROR_OUT_OF_MEMORY`	Heap memory allocation failed in OP-TEE.
`TEE_SUCCESS`	The requested operation successfully completed.

Table 4.1. *Return values common to integrity monitoring commands*

Second, to get the hashes of the code segments of the processes running from the provided path, the command id is `RA_CMD_GET_PSHASH`. Since an application can be started multiple times, the command returns a hash of the code segment of each process running from the binary specified in the parameters. If the returned hashes are not the same, the binary or the code segment of a process was modified between different executions of the application, which could be an indication of system compromise. Other than the return codes specified in Table 4.1, this command could return the error code: `TEE_ERROR_GENERIC`, meaning the currently hashed task does not have a memory descriptor structure (i.e., is a kernel thread) or its code segment is not completely mapped in RAM. Kernel threads, by definition, does not have `mm_struct` structures:

> Kernel threads do not have a process address space and therefore do not have an associated memory descriptor. Thus, the mm field of a kernel thread's process

descriptor is NULL. This is the definition of a kernel thread—processes that have no user context. [28]

To be able to hash the code segment of larger binaries a NW kernel module is used for force loading them into memory. For details see section 4.5.8.

The last command is used for key provisioning (as mentioned in Section 3.5.3), the command id is `RA_CMD_KEY_PROVISIONING`. This command saves an RSA key pair (of the device) and a public key (of the remote client) into Secure Storage. The saved keys are later used for signing and verifying the incoming requests and outgoing responses. The implementation uses a C source file which contains a key pair (consisting of modulus, private and public exponent) and a public key (only modulus and public exponent). This source file must be only compiled and linked into the resulting TA when key provisioning/management is required, therefore the resulting TA binary only contains the keys when it is compiled in provisioning mode, not during normal operation. For details see section 4.5.1.

Using the PTA

Every time access is needed to the Linux data structures located in NW memory e.g., when dereferencing a pointer to NW memory location, that data is loaded using the PTA. This must be done with explicit function calls, because user TAs can not access arbitrary memory locations. The same memory separation and protection principles are applied to user TAs in SW as Linux processes in NW. In the TA, in-memory buffers must be used to hold the copies of NW Linux data structures to avoid repeatedly calling the API functions of the PTA. Since a few of these structures can be rather large (>1 kB) (e.g., `task_struct` structure), and since the stack of a TA is quite limited (usually few kilobytes) they can't be stored on the stack. Although the supervised stack size of the Trusted Applications can be configured at build time[10], the available Secure Memory[11] is often also limited on embedded devices, it is more feasible to use the heap to store the buffers. Alternatively, they could be defined as global variables, however, we found that this approach might lead to unexpected behavior when multiple incoming requests are received, and the resulting problems might be hard to debug, therefore we suggest using dynamic memory allocation on the heap. Also, since the PTA works with physical addresses, every pointer containing NW virtual addresses must be translated first in the Trusted Application, which is discussed in the following section.

[10] `https://github.com/OP-TEE/optee_os/blob/master/documentation/optee_design.md#malloc-pool` (last visited on 2018–10–22)

[11] `https://github.com/OP-TEE/optee_os/blob/master/documentation/optee_design.md#secure-memory`
`https://github.com/OP-TEE/optee_os/blob/master/documentation/porting_guidelines.md#8-memory-firewalls--tzasc` (links last visited on 2018–10–22)

Using process paths in monitoring

In the current integrity monitoring implementation, paths are chosen to identify a process, since paths are globally unique in the Linux file system, and can not be easily altered by the process itself in run time, unlike the name of the process. A process can change its run time name quite easily. For example, a C program can modify the `argv[0]` input parameter in its main function which contains the process name. The modified name appears in NW Linux system monitoring programs like `top`. The process can do this, because the `argv[0]` parameter is located in its own address space, which it has write access to. On the other hand, the file system path to the process binary is stored in kernel memory which is protected from user land processes. Therefore, making the path feasible to use to (partially) identify the running processes. The path combined with the process PID and process command line parameters can be used with more fine-grained whitelisting and identification procedures if required. In the current implementation only the path is used. Multiple processes can have the same path, e.g., a process makes a fork of itself and the same binary is used for the children process, or starting a program two or more times in succession, in which case the paths are the same, but the PIDs is guaranteed to be different. However, PIDs can be unreliable to exactly identify processes in the long run, since the default maximum PID number is 32768, and if this limit is reached, Linux starts to reuse previous PIDs.[12] The `proc` section of the Linux Programmer's Manual contains the following:

> `/proc/sys/kernel/pid_max` (since Linux 2.5.34)
>
> This file specifies the value at which PIDs wrap around (i.e., the value in this file is one greater than the maximum PID). PIDs greater than this value are not allocated; thus, the value in this file also acts as a system-wide limit on the total number of processes and threads. The default value for this file, 32768, results in the same range of PIDs as on earlier kernels. On 32-bit platforms, 32768 is the maximum value for `pid_max`. On 64-bit systems, `pid_max` can be set to any value up to 2^{22} (`PID_MAX_LIMIT`, approximately 4 million).

Linux data structures, macros, and definitions

In this section we briefly explain the most important Linux kernel data structures that we used. A more lengthy description can be found in [27] (p. 603 – 307) and in [28] (p. 85 – 113).

Linux processes are represented by `task_struct` structures in kernel memory.[13] An instance of this structure is around 3 KB in our environment[14], and it contains a lot of

[12] `http://man7.org/linux/man-pages/man5/proc.5.html` (last visited on 2018–10–19)

[13] `https://elixir.bootlin.com/linux/v4.6.3/source/include/linux/sched.h#L1394` (last visited 2018–10–20)

[14] Measured in our environment — 64-bit kernel, version 4.6.3

information about the particular running process. These structures are stored in the kernel memory cache (`kmem_cache`) — part of the kernel memory. The key points of the structure are the following:

- `pid_t pid;` Contains the PID of the process
- `struct list_head children;` Holds a circular, double linked list with the children of the given process.
- `struct list_head sibling;` Holds a linkage to the parent's children.
- `char comm[16];` Contains the name of the process. If a kernel process' name ends with a / character followed by a number then the number indicates which processor core the thread is running on. (Note: this is not used in our integrity monitoring algorithms, because of the reasons discussed in Section 4.5.1)

Each `task_struct` structure contains a link to its first child, which is linked to the sibling field of the child. The last `task_struct` sibling of the structure in this chain is linked back to the parent's children field, which is shown in the following example:

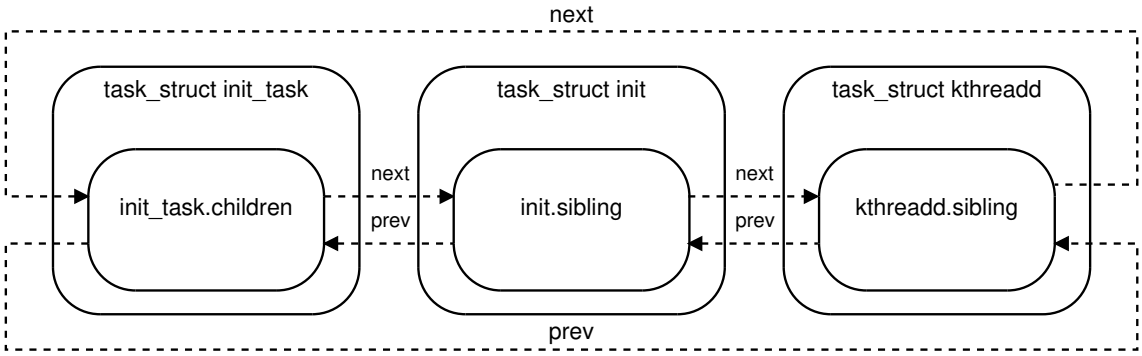

Figure 4.6. *Linux `task_struct` linkage example*

Figure 4.6 shows a circular, doubly linked list with three data nodes. The traversal of the list starts at the `init_task` node on the left, which is the head of the linked list data structure. The other nodes are reachable using the `next` and `prev` sibling pointers, and the circularity of the list is assured by the first and last nodes having pointers to each other (on the example `init_task` and `kthreadd`).

The previous list structure is a great example of general linked lists in the kernel. These definitions can be found in `include/linux/list.h` kernel source file. The `list_head` structure can be used to store lists. There are many macros and functions defined in the kernel to operate on lists. For example, we heavily relied on the `list_entry` macro (defined in `include/linux/list.h`), which is in fact an alias of the `container_of` macro (defined in `include/linux/kernel.h`). The comment for this macro states the following: "cast a

member of a structure out to the containing structure".[15] We used this to iterate over the list elements.

In Linux, the address space of the processes are described with the `mm_struct` structure. This structure is called the memory descriptor, and contains every information about the process address space. It is defined in `linux/mm_types.h`. Among many things, it contains the start and end addresses of the various sections of the process virtual memory space, corresponding to the sections in the ELF binary backing the process. We used the `start_code` and `end_code` members of the structure, in particular, because those contain the start and end virtual address of the process code section. The `vm_area_struct *mmap` pointer is also interesting, because it points to the VMA list of the process, which is a list of all the memory areas in this address space.

Virtual Memory Areas, represented as `vm_area_struct` structures, describe a "single memory area over a contiguous interval in a given address space", according to [28]. They can be backed by various objects, e.g., physical memory pages or files, like executable binaries and loadable libraries. The VMAs are important to us, because they can be used to get the file system path to the process binary, and also the path of the shared libraries the process uses. We use the `vm_start` and `vm_end` members—which describe the lowest and highest virtual addresses corresponding to the virtual memory area—and the `*vm_file` pointer, which identifies the file mapped to the area. Also, there are VMA flags (`vm_flags`)—read, write, and execute specifically—which can be used by integrity monitoring algorithms to detect suspicious protection bits which may indicate shellcode or shard library injection, as can be seen in [27], however our implementation does not use them.

The `mm_struct` also provides access to the page tables used by the kernel. Virtual memory addresses are used by applications, but they must be converted to their physical counterparts, which are used by the processor. This process and the translation of kernel virtual addresses are described in the next section.

In the Linux virtual file system (VFS), `dentry` means the following:

> VFS treats directories as a type of file. In the path `/bin/vi`, both `bin` and `vi` are files—`bin` being the special directory file and `vi` being a regular file. [...]
>
> A dentry is a specific component in a path. Using the previous example, `/` , `bin` , and `vi` are all dentry objects. The first two are directories and the last is a regular file. This is an important point: Dentry objects are all components in a path, including files. Resolving a path and walking its components is a nontrivial exercise, time-consuming and heavy on string operations, which are expensive to execute and cumbersome to code. The dentry object makes the whole process easier. [28]

[15] A brief documentation can be found here too: `https://www.kernel.org/doc/htmldocs/kernel-api/API-list-entry.html` (last visited on 2018–10–20)

Address translation

NW virtual addresses can not be directly accessed from the TAs since NW and SW use different memory management page tables. These addresses must be translated to physical ones. In our environment, kernel address space layout randomization (KASLR) is not used in the Linux kernel running on the Raspberry Pi 3 with 64–bit Linux kernel, because our verified boot process (along with OP-TEE) does not support this feature, making the translation of kernel virtual addresses significantly easier.

The translation for Linux kernel virtual addresses is quite simple but also architecture and platform dependent. On 32–bit systems, kernel memory is mapped linearly in the lower memory areas (lowmem), but non-contiguously for higher addresses (highmem).[16] The need for this separation arises from the fact that on 32–bit devices, only 2^{32} bits of memory is addressable. While Physical Address Extension aims to solve this problem, it might not be supported on every embedded device. Our implementation should work on 32–bit kernels, since it uses kernel memory structures which are located in the linearly mapped region [28]. On 64-bit systems the complete kernel memory is linearly mapped, there is no lowmem/highmem separation, because the available address space is significantly larger, making the before mentioned separation unnecessary. This translation is implemented as a macro called `__virt_to_phys` in the kernel source code, as seen in Listing 4.3.[17]

```
1 #define __virt_to_phys(x) ({                                          \
2     phys_addr_t __x = (phys_addr_t)(x);                               \
3     __x & BIT(VA_BITS - 1) ? (__x & ~PAGE_OFFSET) + PHYS_OFFSET :     \
4     (__x - kimage_voffset); })
```

Listing 4.3. *Macro for translating kernel virtual addresses to physical addresses*

This macro first checks the type of the kernel virtual address and it calculates the offsets accordingly. We implemented the same macro in the Secure World TA to handle kernel address translation. The challenge in the implementation was that we had to re-implement the complete chain of other Linux kernel macros and definitions that this macro uses. The `PAGE_OFFSET` value is dependent on the load address of the kernel while booting. This value can change if the load address changes, so recalculation might needed if the address defined in the FIT ITS file for U-Boot changes. The Linux kernel solves this problem on some platforms by patching itself at run time, calculating the offset on every boot when built with `CONFIG_ARM_PATCH_PHYS_VIRT=y`.[18] Recalculation of the offset can be done e.g., with using a known virtual – physical address pair, or with a kernel module reading the

[16] `https://elinux.org/images/b/b0/Introduction_to_Memory_Management_in_Linux.pdf`
`https://users.nccs.gov/~fwang2/linux/lk_addressing.txt`
`http://thinkiii.blogspot.hu/2014/02/arm64-linux-kernel-virtual-address-space.html`
(each link last visited on 2018–10–19)

[17] `https://elixir.bootlin.com/linux/v4.6.3/source/arch/arm64/include/asm/memory.h#L89` (last visited on 2018–10–19)

[18] `https://stackoverflow.com/questions/16909655` (last visited on 2018–10–19)

offset directly from Linux. During development we tested and used both approach. As mentioned before, KASLR is not used in our environment, so the previous simple macro is enough for the address calculations.

The translation for per process virtual addresses is more complicated than for kernel virtual addresses, since paging is used in the memory management of processes [11]. The page table structures used in Linux memory management must be walked manually from the Secure World TA. This is highly architecture and platform dependent, since page tables can contain different number of levels, different masks and bit field widths must be used to calculate the offsets for each level. Modern Linux kernels generally use a 4 level page table (in software), but the platform dependent implementations differ.[19] For example when the architecture uses 2 level hardware page tables, the 2 non-existent levels are "fixed up" when Linux translates addresses. This essentially means that the functions—corresponding to the tables of those levels—are simply returning the values of the previous levels, they are transparently left out.[20] Solutions similar to this can be found in the kernel at multiple places. They aim to enable the writing of more generic algorithms and code. Another example is the generic linked list representation in the kernel, where macros are used to calculate the addresses of the neighbor node pointers, which by default does not point to the next node of the list, but to another generic node pointer type located and defined inside the structures of the nodes. A visual example and description of this can be seen in Figure 4.6. More information about kernel programming can be found in [11].

We used the kernel memory management source code to implement the page walk in the TA. Using the `mm_struct` structure belonging to a process we can translate a virtual address from the address space of the process to a physical memory location. To obtain the `mm_struct`, we traverse the linked list of the running processes (starting from the `init_task` symbol, representing the list head), until the `task_struct` corresponding to our process is reached. The `task_struct` contains a pointer to the `mm_struct`, which is needed because each process has their own virtual memory, and this structure describes that area, as mentioned in Section 4.5.1. The first level is called Page Global Directory (PGD). It is defined in the `mm_struct` as a pointer, and contains the physical memory address of the second level, page upper directory (PUD), which then contains a pointer to the third level, page middle directory (PMD), and which contains a pointer to the fourth level, page table entry (PTE). Using the lowest bits of the address as offset into the page itself, the final physical address can be calculated. This segmentation of the address can be seen in Figure 4.7 [21]

[19] The kernel uses four level page tables since 2005 (version 2.6.10): `https://lwn.net/Articles/117749/` Five level page tables were introduced in 2017 (version 4.11-rc2), although currently there is no hardware supporting five levels: `https://lwn.net/Articles/717293/`

[20] `https://manybutfinite.com/post/how-the-kernel-manages-your-memory/` (last visited on 2018–10–19)

[21] The masks, offsets, and indexes are defined in the Linux kernel source code (mostly in `arch/arm/include/asm/page.h` and `arch/arm/include/asm/pgtable.h`).

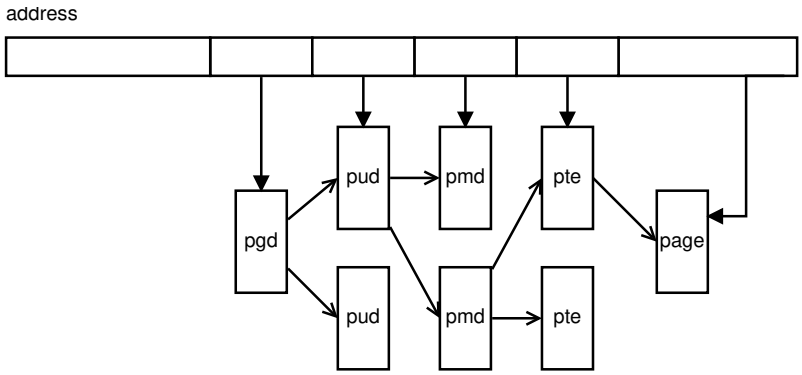

Figure 4.7. *Virtual Address*

Traversing the process linked list

Traversing the list of running processes is important for our integrity monitoring procedures. This is a circular doubly linked list of `task_struct` structures, representing a tree data structure, as mentioned in Section 4.5.1. The virtual address of the head of the list can be found in the `system.map` file, generated when building the Linux kernel, or can be obtained dynamically from `/proc/kallsyms` on the running system. The name of the symbol is `init_task`. In our implementation, it is stored as a C define in a header file, however, since it can change with every build of the kernel, it should be updated manually (or automatically during building). The algorithm is a simple recursive DFS tree traversal. The next and previous pointers can not be accessed directly (pointees are in NW memory), so the PTA must be called on each access. A callback function is called on every `task_struct` item in the tree (making the algorithm generic). The callback function has two arguments: the current level in the tree and a void pointer representing the context or payload for the function. In most cases the payload should contain a pointer to the current task and a pointer to a structure representing the current attestation request and response. The `list_entry` function is a macro used in the Linux kernel to help the generic traversal of data structures, as shown in section 4.5.1.

Getting path to the binary of a process

Each process has a list of Virtual Memory Area (VMA) structures representing the distinct regions of its address space, as described in section 4.5.1. The algorithm in our implementation, simply walks the VMA list of a process (`mm.mmap`) until it reaches the first VMA corresponding to the start of the code segment. We can be sure that the file belonging to the code section area, actually contains the code being executed. The algorithm then extracts the path to the file backing the VMA—the process binary file in the file system. A helper function is implemented to get the path in the correct canonical order.[22] Since the directory entry (`dentry`) belonging to the file is the last part of the canonical path,

[22]We defined a constant to limit the path length, since a completely reliable limit for the full path of a file does not exist in Linux: `http://insanecoding.blogspot.com/2007/11/pathmax-simply-isnt.html` (last visited on 2018–10–22)

recursion can be used to get the full path in correct order (the filename and directory entry are parts of a linked list with pointers to the parent directory entry).

Getting the process list

The list of running processes can be obtained by traversing the process list, and collecting the paths from the process VMAs. This algorithm simply uses the list traversal and path accessing algorithms, introduced in section 4.5.1. In the current implementation, only the path is collected, however with simple modifications more information can be gathered for each process, e.g., PID, UID, GID, time of start, process state (running, terminated, waiting, etc.), and other stats usually shown by NW Linux system monitoring tools like `top` or `htop`. The callback of our process traversal algorithm gets the path of the current task and writes it into the shared memory output buffer. Since the number of running tasks and the path length for each task is unknown beforehand, in the current implementation the buffer is written until the size specified in the GPD parameter size is reached, and if the traversal was not completed, the procedure stops and returns an error code (`TEE_ERROR_SHORT_BUFFER`). The NW Server Application then can decide if it allocates a larger buffer and repeats the request or returns an error to the remote client. The payload to the traversal algorithm consists of pointers to a `task_struct` and to a path buffer, helper pointers to use with serializing the paths into the buffer, and a pointer to the structure representing the attestation request and response. The `task_struct` and path buffer is allocated on the heap and reused for each process, since even the heap could be quite limited in OP-TEE.[23] Also, OP-TEE uses `bget` generic heap allocator (written in 1972), which can fragment the heap when allocating and freeing large chunks of memory.[24]

Calculating hash for the code segment of a process

The hash of the code segment of processes can be used to verify that the process running from the given path has not been replaced, modified or tampered with. On the remote client side, a list of known hashes can be maintained and used to check against the hashes of the running processes. This way the integrity of critical applications can be monitored, and the remote client can be sure that the code running on the system can be trusted. The maximum path length of a binary specified in the request, is limited, to avoid potential overflows in the monitoring algorithms, and the request can contain only one path with this length. The list of hashes is obtained by traversing the process list, selecting the tasks whose binary paths match the path set in the request, calculating hashes for their code sections, and storing the results in the response buffer. If the response result buffer is not large enough to hold the hash of the process currently iterated, `TEE_ERROR_SHORT_BUFFER` error

[23] `https://github.com/OP-TEE/optee_os/issues/947#issuecomment-235893257` (last visited on 2018–10–20)

[24] `https://github.com/OP-TEE/optee_os/issues/2395#issuecomment-397400065` (last visited on 2018–10–20)

code is returned to the caller (NW Server Application). For the given process, its SHA256 hash is calculated. Memory is allocated on the heap for the `mm_struct` of the process, and for an array containing the addresses of the pages where the code segment is located. The number of pages to hash is calculated using the following equation, where $page\_count$ is the number of pages, end_code and $start\_code$ are virtual addresses corresponding to the start and end of the code segment, and $PAGE\_SIZE$ is the size of physical pages used by the kernel (usually 4 KB when huge pages are not used):

$$page\_count = \frac{end\_code - start\_code}{PAGE\_SIZE} + 1$$

The current page length also needs to be calculated, because the data might not fill the entire page (`PAGE_SIZE` bytes), and only the actually used range needs to be hashed. This is done by checking if the end address of the code segment is inside the current page, as shown in listing 4.4, and also while iterating over the pages (shown in listing 4.5)

```
uint32_t start_address = mm->start_code;
uint32_t end_address = mm->end_code;
<snip>
if ((start_address & ~(PAGE_SIZE - 1)) + PAGE_SIZE > end_address)
    page_len = end_address - start_address;
else
    page_len = PAGE_SIZE - (start_address & (PAGE_SIZE - 1));
```

Listing 4.4. *Calculating page length*

```
uint32_t start_address = mm->start_code;
uint32_t end_address = mm->end_code;
<snip>
for (size_t i = 0; i < page_count; ++i) {
    <snip>
    start_address += page_len;
    if (start_address + PAGE_SIZE > end_address)
        page_len = (end_address - start_address);
    else
        page_len = PAGE_SIZE;
}
```

Listing 4.5. *Iterating over the pages*

The calculated digest is converted to a hexadecimal, zero terminated string representation and returned in the hash buffer, which was provided in the payload. [25]

[25] The complete code section of the given process must be present (loaded) in memory for this feature to work. For more details about the problem and solution see section 4.5.8.

Key provisioning

A simple key provisioning mode is available in the TA, as mentioned in section 3.5.3. The RSA key pair of the device and the public key of the remote client is located a C source file. The values are currently hard coded and the file only gets compiled if the TA is built with provisioning mode enabled (recommended only during development). To enable or disable the mode the following line can be included in the TA build makefile:

```
srcs-y += provision.c
```

This assures that the keys are not present in the binaries during normal operation in any form. The name of the stored RSA key pair and public key objects stored in Secure Storage are defined the following:

```
1 #define RA_STORAGE_KEYPAIR_NAME "ra_keypair"
2 #define RA_STORAGE_ATTESTER_PUBKEY_NAME "ra_attester_pubkey"
```

OP-TEE uses the strings above to find an object in Secure Storage, since objects are protected by and stored in a hash tree (implemented as a binary tree).[26] The algorithm first tries to open the object with the given name, if it does not exist, a transient object is created in memory and initialized with the given modulus, public and/or private exponent (depending on which object is being saved). The transient object is then saved as a persistent object, and is written to the Normal World file system.

Request verification and response signing

Every incoming request is verified as the start of the TA command invocation step. The request is not processed if the verification fails. For the verification, an SHA256 hash of the request is calculated using the version, monitoring operation name, arguments, and nonce. Then the digest and the signature are passed to the signature checking algorithm, which uses the keys in Secure Storage to verify the given signature. Signing is done the same way. The SHA256 hash of the response is calculated: version, results, nonce. Then the digest is signed with the private key of the device from Secure Storage.

4.5.2 Secure World Pseudo Trusted Application

The Secure World Pseudo Trusted Application provides access and services to the Normal World Linux memory for Trusted Applications in OP-TEE. It is implemented as a Pseudo TA to have sufficient privileges to access otherwise inaccessible physical memory locations. The exposed interface is TA $\leftrightarrow$ TA only, meaning that the PTA is not callable from the Normal World, to avoid breaching the security measures between NW applications, set up by the Linux kernel (i.e., unrelated applications can not access the memory areas of each other).

[26] `https://github.com/OP-TEE/optee_os/blob/master/documentation/secure_storage.md` (last visited 2018–10–21)

Physical memory addresses are used in the API to remain platform and implementation independent as much as possible, otherwise the address translation functionalities would have to be implemented in the PTA. This would make it necessary to re-implement them in different environments, breaking the portability of the PTA, also it would result in more complex code running with kernel privileges, which should be avoided. Therefore, the caller first must translate NW (kernel and process) virtual addresses to physical ones. This translation process often depends on the used platform, architecture, Linux Kernel version, etc., as described in section 4.5.1. Using this API makes access to NW memory possible to user TAs, eliminating the possible need to rewrite those applications as PTAs (which are usually harder to maintain as consequence of the reasons detailed in section 3.5.2).

The PTA exposes the following features: reading from physical memory address ranges; hashing of physical memory address ranges with SHA256 algorithm—calculating a single hash for one or more address range and calculating separate hashes for every given address range.

In OP-TEE there are mechanisms in place that prevent the TAs from accessing the memory area of other TAs, OP-TEE kernel, and other areas including NW memory. Those are the reasons, only pseudo TAs can access NW memory. The following manual preparation and configuration is necessary for the proper functioning of the PTA.

Memory mappings

OP-TEE uses separate virtual memory translation tables from the NW Linux kernel. To access the physical memory, the corresponding address ranges must be configured in OP-TEE. This configuration procedure is called memory mapping. The mappings depend on the particular use cases, for example one can map the entire physical memory to have access to everything, or if only the kernel memory area is needed one should only map that range. It might be a good practice to have only the needed ranges mapped, because OP-TEE has limited number of translation tables, and that limit might be reached when mapping large areas. We reached this limit when we tried to map the complete kernel memory, like in listing 4.6, we set the `MAX_XLAT_TABLES` constant to `100` (from `5`) in `core_mmu_lpae.h`, because with lower values, OP-TEE kernel panics in the memory mapping functions. We found that `core_mmu_entry_to_finer_grained` function tries to make the tables "finer grained" (likely trying to create more small L1 tables), but can not, since it reaches the defined maximum translation table limit. Thus, setting the limit higher solves the problem.

The default OP-TEE mappings are defined in the `core/arch/arm/plat-*/` platform specific folders. For example, add the lines from listing 4.6 to `core/arch/arm/plat-rpi3/main.c` in order to register a new physical memory mapping:

```
1 #define MAP_KERNEL_START        0x00001000
2 #define MAP_KERNEL_SIZE         (128 * 1024 * 1024)
```

```
3 #define MAP_KERNEL_RAM_START    0x0a000000
4 #define MAP_KERNEL_RAM_SIZE     (848 * 1024 * 1024)
5
6 register_phys_mem_ul(MEM_AREA_RAM_NSEC, MAP_KERNEL_START, MAP_KERNEL_SIZE↩
      );
7 register_phys_mem_ul(MEM_AREA_RAM_NSEC, MAP_KERNEL_RAM_START, ↩
      MAP_KERNEL_RAM_SIZE);
```

Listing 4.6. *Registering a memory region in OP-TEE*

The registered memory ranges must be `PAGE_SIZE` (4096 byte) aligned (same as `CORE_MMU_DEVICE_SIZE` constant in OP-TEE).[27] The `ROUNDUP` macro can be used to ensure this. In the example listing 4.6, the mapping consists of an 128MB area of the entire kernel memory (where the `init_task` symbol and the other `task_structs` are located), and an 848MB area used by the user land programs. This mapping covers most memory locations that Linux uses, except the memory mapped IO devices at high memory addresses (e.g., watchdog, UART, etc.), which are mapped in OP-TEE by default.[28]

[27] `https://github.com/OP-TEE/optee_os/issues/1343#issuecomment-282841451` (last visited on 2018–10–21)

[28] `https://github.com/OP-TEE/optee_os/blob/ee595e950f5be1ace3e831261c22a0e99f959046/core/arch/arm/plat-rpi3/main.c#L41` (last visited on 2018–10–21)

```
                   0x4000_0000          DRAM0_SIZE
                  +--------------------------------+
                  |    Device Base                 | UART: 0x3f215040
                  |                                | WDT : 0x3f100000
                  |0x3f00_0000                     |
                  +--------------------------------+
                  |    NW RAM                      |
                  |                                | ~860 MB usable RAM probably
                  |0x0a00_0000 or 0x0a40_0000      |
                ^ +--------------------------------+
                | |    ???                         |
     Secure RAM | |                                | 12 MB or 8 MB
         32 MB  | |0x0980_0000                     | unused secure RAM?
      or 28 MB  | +--------------------------------+
                | |    TA RAM                      |
                | |                          16 MB |
                | |0x0880_0000    CFG_TA_RAM_START |
              ^ | +--------------------------------+
              | | |    OP-TEE Core RAM             |
   OP-TEE RAM | | |                                | BL32
        4 MB  | | |0x0842_0000   CFG_TEE_LOAD_ADDR |
              | | +--------------------------------+
              | | |    ARM TF                      |
              | | |                        128 KB  | BL31
              | | |0x0840_0000         TZDRAM_BASE | == CFG_TEE_RAM_START
              v v +--------------------------------+
                  |    NS SHM                      |
                  |                          4 MB  | Non-secure shared memory
                  |0x0800_0000     CFG_SHMEM_START |
                ^ +--------------------------------+
                | |    Linux DTB                   |
Linux kernel RAM| |                                |
       127.5 MB | |0x0170_0000                     |
                | +--------------------------------+
                | |    Linux kernel                |
                | | +----------------------------+ |
                | | |    BL30 MCU FW             | | BL30: early tmp buffer for
                | | |                      1 MB  | | MCU firmware, parsed by BL32
                | | |0x0100_0000                 | |
                | | +----------------------------+ |
                | |0x0008_0000                     |
                v +--------------------------------+
                  |    U-Boot                      | Stubs + U-Boot,
                  |                                | U-Boot self-relocates
                  |0x0000_0000          DRAM0_BASE | to high memory
                  +--------------------------------+
```

Listing 4.7. *Raspberry Pi 3 Model B Memory Map*

Listing 4.7 shows the different memory regions of the Raspberry Pi 3 Model B in our environment. The different addresses are gathered from various source code, configuration, and documentation files of the Linux kernel, OP-TEE, and ARM Trusted Firmware. Every constant name on the map is from OP-TEE `platform_config.h`. The most important sections are described below:

NS SHM Non-secure shared memory, used for communication between the NW and SW applications. It is non-secure by definition, because the NW allocates this memory

area, and has complete access to it. In OP-TEE the default configuration is 4MB.

ARM TF The ARM Trusted Firmware binary is located at this area, it is responsible for the context switching of the processor between Normal and Secure operation modes. This memory area is accessible by Secure World only.

OP-TEE Core RAM The kernel memory area of OP-TEE kernel. Secure, accessible by Secure World only.

TA RAM Memory area for the Trusted Applications. Secure, accessible by Secure World only.

NW RAM Memory area for the Linux user applications. Non-secure, accessible by both worlds.

Device Base Memory mapped IO devices are located here (e.g., watchdog, UART, etc.). Non-secure, accessible by both worlds.

Linux kernel The kernel memory area belonging to Linux kernel. Non-secure, accessible by both worlds.

Some inconsistencies can be found in the different files, these are indicated with the keyword "`or`" on listing 4.7. ARM TF platform specific configuration constants and OP-TEE configuration constants slightly differ: ARM TF defines the secure RAM 28 MB (`DRAM_SEC_SIZE`) in size, while OP-TEE defines it as 32 MB (`TZDRAM_SIZE`).[29] Also, there is 8 or 12 MB unused secure RAM according to the definitions in the configuration files.

Building into OP-TEE Core

Pseudo TAs are built into the OP-TEE kernel. To build a PTA, the source code of the Trusted Application must be in the OP-TEE kernel directory tree, and a target for it must be added to the OP-TEE build system. To achieve this, PTAs can be placed in the `core/arch/arm/plat-*/` platform specific folders or in `core/arch/arm/pta/` alongside the default OP-TEE PTAs. For example, we created a symbolic link to one of the previous locations for the PTA source folder. To add a build target, we added the following line to `sub.mk` in the chosen folder:

```
srcs-y += nw_memory_api/nw_memory_api.c
```

[29] `https://github.com/linaro-swg/arm-trusted-firmware/blob/1da4e15529a32fa244f5e3effc9a90549beb1a26/plat/rpi3/rpi3_def.h#L55` and `https://github.com/OP-TEE/optee_os/blob/c21bf051b0191b5ee81eb138994b7f3a3d579a1a/core/arch/arm/plat-rpi3/platform_config.h#L74` respectively for the two configuration constants (each link last visited on 2018–10–21)

Security considerations, trade-offs

The feature set of the PTA is intended to be minimal in order to keep the possible attack surface of the API minimal, since the PTA runs at the same privilege level as the OP-TEE kernel. As mentioned before, only TA $\leftrightarrow$ TA interface is provided, thus restricting the possible users of the API to other Trusted Applications, to prevent the NW from using it directly to read arbitrary (possibly SW) memory.

Also, it is possible to use the API to read from the private memory of other TAs. This is an inherent property of it being implemented as a Pseudo TA (OP-TEE kernel can access the complete secure memory, therefore PTAs compiled into to it, can too). Proper access control and address range checking should be introduced to minimize the impact of this.

One could whitelist the users of the API, with a predefined set of TA UUIDs hard coded into the PTA (or possibly loaded from Secure Storage). This way only TAs with whitelisted UUIDs can use the services exposed by the PTA. This could be useful if not every TA can be trusted on the system (i.e., third party ones).

As another possible solution, since TAs can have different flags that can be set at build time and can be checked at run time, these flags could be used for less restrictive access control to the API. Adding a new user of the API could be done when building the given TA and setting a flag, whilst with whitelisting, if the whitelist is built into the PTA it is necessary to fully rebuild the OP-TEE kernel image or if the whitelist is stored in Secure Storage, the Secure Storage objects must be manipulated.

To limit the accessible physical address range to safe ones (e.g., only Linux kernel memory), checks could be run at every memory access to determine the conformity of the given range and only complete requests with conform ranges. These techniques could introduce a considerable overhead, since the API is called quite frequently in our implementation (e.g., when dereferencing a NW pointer). Note: In the current implementation, neither whitelisting, nor the conformity checks mentioned above are used, as the project is intended as a proof of concept solution.

Interrupts

Since frequent calls are expected for this API, the execution time and performance of each function is quite important. During the execution of every TA function, foreign and native interruptions can occur.[30] Native interrupts (i.e., secure interrupts from the Secure World) are handled by OP-TEE. Foreign interrupts (i.e., not secure interrupts from the NW Linux) are not handled by OP-TEE, just simply passed to the Normal World for handling, therefore they cause context switches. Context switching is not desired during the memory accessing or hash calculating operations, because it can cause unnecessary overhead. To

[30] `https://github.com/OP-TEE/optee_os/blob/master/documentation/interrupt_handling.md` (last visited 2018–10–21)

reduce the possible latency, every interrupt is disabled during the execution of the API functions. Another reason for this, is that otherwise, the memory of a measured program may change while it is being hashed, resulting in unreliable or straight up manipulated results.

Interface, commands

As mentioned before, the PTA provides two functionality types, namely reading from the physical memory and calculating hashes for physical memory regions. Before usage, the physical addresses must be converted to OP-TEE virtual addresses. This can be done by the `phys_to_virt` function provided by OP-TEE memory management code. In our implementation, only the start and end of the address range is checked for existing memory mappings, and the range should be contiguously mapped, otherwise `TEE_ERROR_BAD_STATE` is returned. More extensive checking of the memory mappings could introduce undesired overhead.

The memory reading command is `NW_MEMORY_API_READ_MEM`. The implementation consists of a simple `memcpy` function call, which copies bytes from the physical memory address specified in the first GPD argument (`value.a`), into the buffer given in the second GPD parameter (`memref`), and the number of copied bytes are exactly the size of the given buffer (this range is $[value.a, value.a + memref.size]$). The function can be found in listing 4.8.

```
1  static TEE_Result read_mem(uint32_t param_types,
2  TEE_Param params[TEE_NUM_PARAMS])
3  {
4      void *nw_mem_addr;
5      <snip>
6      nw_mem_addr = P2V((void *)params[0].value.a);
7      <snip>
8      memcpy((void *)params[1].memref.buffer, nw_mem_addr,
9             params[1].memref.size);
10
11     return TEE_SUCCESS;
12 }
```

Listing 4.8. *Reading from NW memory with the PTA*

The hash calculating commands use the underlying `crypto_hash_*` interface, which is backed by the LibTomCrypt cryptography library.[31] The first memory hash calculating command is `NW_MEMORY_API_HASH_MEM`. It calculates a single SHA256 hash of multiple NW memory ranges defined in the input buffer (first GPD argument) and copies the digest into the output buffer (second GPD argument). The second hash calculating command is `NW_MEMORY_API_HASH_MEM_MULTI`. It calculates separate SHA256 hashes for each NW

[31] `https://github.com/OP-TEE/optee_os/blob/master/documentation/crypto.md` (last visited on 2018–10–22)

memory range specified in the first GPD parameter, and serializes the hashes (as zero terminated strings) into the output buffer (second GPD argument).

4.5.3 Attestation session

An attestation session is a request-response message exchange session, where the client sends a request message to the device which attests its state in a response message. The simplest solution is to use a client-server architecture, where the device acts as the server, and clients can connect to request attestation from the device. The protocol is based on an open-source Remote Procedure Call library called **gRPC**[32] which is maintained by Google, and it is available for numerous programming languages. We chose this library because it allows us to create a quick and convenient implementation of the protocol, and it also provides an SSL/TLS communication layer[33] to prevent man-in-the-middle attacks. The gRPC library uses Protocol Buffers[34] which is a language-independent extensible data serialization mechanism. The message and service definitions are listed as Protocol Buffers v3 definitions. We use this format for communication between the client and the server application. We convert these messages to our custom format on both the server and the client side, because it allows us to verify the message signature more easily.

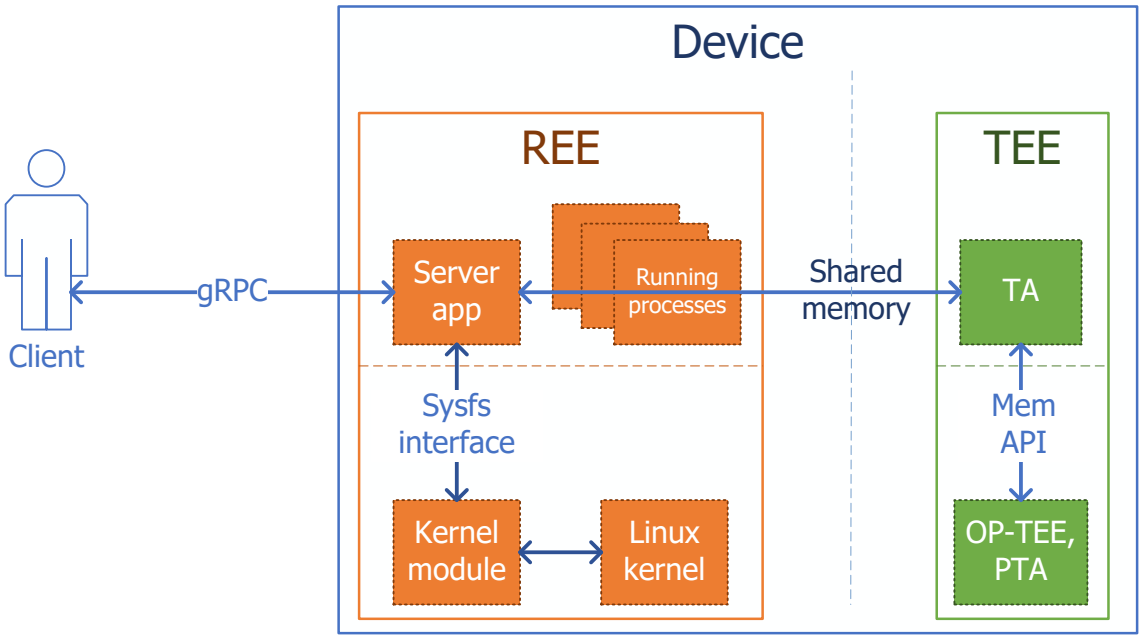

Figure 4.8. *Remote attestation architecture*

4.5.4 Protocol messages

The request and response messages are digitally signed, therefore we need to assemble a byte array from the request and response fields to calculate a cryptographic hash as

[32] `https://grpc.io` (last visited: 2018-10-22)
[33] `https://grpc.io/docs/guides/auth.html#supported-auth-mechanisms` (last visited: 2018-10-22)
[34] `https://developers.google.com/protocol-buffers/` (last visited: 2018-10-22)

the input of the signing primitive. We created a simple and extensible structure for this purpose for both the request and the response messages. The definitions of these structures are listed below.

Request message

The request message consists of the following five fields:

- `ver`: version of the request (32-bit unsigned integer)
  The protocol version is a required field which is a number. If a message format of the protocol is changed, this number should indicate the changes.
- `fname`: the name of the attestation method (character array)
  This field specifies the attestation method which will be executed by the Trusted Application. This is required because the Trusted Application should ensure the correct method was called from the Normal World application.
- `args`: optional arguments (character array)
  If there are any arguments which should be passed to the attestation method this field should be used. If the method doesn't require any arguments, the field should be an empty string.
- `n`: randomly generated nonce (64-bit unsigned integer)
  The nonce is generated to ensure the response is a valid reply for the request. This is a randomly generated number which is large enough to be able to avoid possible collisions.
- `signature`: digitally signed request message digest (character array)
  This is the digital signature of the message. It is a byte array encoded in a string, where each character represents a hexadecimal number.

Figure 4.9. *Request message fields*

```
syntax = "proto3";
package attestation;

message ProcessListRequest {
        uint32 version = 1;
        uint64 challenge = 2;
        string signature = 3;
}
message ProcessHashRequest {
        uint32 version = 1;
        string process_path = 2;
        uint64 challenge = 3;
```

```
        string signature = 4;
}
```

Listing 4.9. *gRPC request message prototypes*

Response message

The response message consists of the following four fields:

- `ver`: version of the response (32-bit unsigned integer)
  The protocol version is a required field which is a number. If a message format of the protocol is changed, this number should indicate the changes.
- `res`: attestation results (character array)
  This is the result of the attestation method represented as a null terminated string.
- `n`: nonce generated by the client (64-bit unsigned integer)
  The nonce is generated to ensure the response is a valid reply for the request. This should have the same value as the value in the request message.
- `signature`: digitally signed response message digest (character array)
  This is the digital signature of the message. It is a byte array encoded in a string, where each character represents a hexadecimal number.

Figure 4.10. *Response message fields*

```
syntax = "proto3";
package attestation;

message ProcessListReply {
        uint32 version = 1;
        repeated string process_path = 2;
        uint64 challenge = 3;
        string signature = 4;
}
message ProcessHashReply {
        uint32 version = 1;
        repeated string process_hash = 2;
        uint64 challenge = 3;
        string signature = 4;
}
```

Listing 4.10. *gRPC response message prototypes*

4.5.5 Message signature

The messages are digitally signed with the private keys. We calculate the SHA256 digest of the message which is then signed with the private key with a probabilistic signature scheme called RSASSA-PSS[35]. The message can be authenticated with the public key on the other side of the communication channel. This signature scheme is implemented in the OP-TEE API so it is easy to use for creating signatures in the Secure World. On the client side we use the pycrypto library which has built-in support for RSASSA-PSS. The signature creation and verification methods are the following:

Sign(**digest**,**privkey**): PKCS#1-PSS signature function

- **digest**: message digest calculated with a cryptographic hash function
- **privkey**: RSA private key

Verify(**signature**,**digest**,**pubkey**): PKCS#1-PSS verification function

- **signature**: PSS signature created by using an RSA private key
- **digest**: message digest calculated with a cryptographic hash function
- **pubkey**: RSA public key

4.5.6 Attestation client

The client is implemented in Python using the **grpcio**[36] and **pycrypto**[37] libraries. The client can be configured with the following command line parameters:

- `-host <host>`: hostname or IP address of the server (default: localhost)
- `-port <port>`: the server port (default: 4433)
- `<command>`: the command to run (can be `pslist` for process list query, or `pshash` for process hash query)
- `[param [param ...]]`: optional parameters (for process hash query, the process paths)

Two commands are available, the first is called `pslist`, which fetches a list of the binaries which are loaded on the server operating system. This command has no parameters and it receives a list containing the full paths of the loaded binaries.

[35] `https://tools.ietf.org/html/rfc3447#page-29` (last visited: 2018-10-22)
[36] `https://pypi.org/project/grpcio/` (last visited: 2018-10-22)
[37] `https://pypi.org/project/pycrypto/` (last visited: 2018-10-22)

The second command is `pshash` which requires a list of full file paths. The server searches the process list for matching binaries and calculates the checksum of the code segment for each target it founds. The response contains a list of these checksums. Multiple paths can be specified in which case the client sends a single request to the server for each path. The client can compare the process' checksums to a list of previously calculated hashes. This can be done by adding a new entry to the `process_db.csv` file with the calculated SHA256 hash and the process path. If multiple checksums exist for a given path, the script tries to match it to every instance. The script shows a warning if no matches exist, or if the process path is not found in the database.

```
hash,path
472c[..]e5bc,/usr/sbin/sshd
373f[..]5154,/bin/busybox
9a21[..]7bdb,/usr/bin/attest_server
1d87[..]d4bf,/usr/sbin/tee-supplicant
```

Listing 4.11. *process_ db.csv with shortened hashes*

4.5.7 Attestation server

The server app is implemented by using the gRPC C++ library to communicate with the client, and the OP-TEE library for communicating with the Secure World side. gRPC supports communication over SSL/TLS which ensures a secure communication channel is used. By default, both the server and the client validate each other's certificate using a common root certificate. The server functionality is implemented in a class called `AttestationServer`. This class is a child of the `RemoteAttestation::Service` class which is auto-generated using the **protoc**[38] C++ code generator and the `.proto` file.

```
syntax = "proto3";
package attestation;

service RemoteAttestation {
        rpc ProcessList(ProcessListRequest) returns (ProcessListReply) {}
        rpc ProcessHash(ProcessHashRequest) returns (ProcessHashReply) {}
}
```

Listing 4.12. *gRPC service definitions*

The source code is written in C++ and CMake is used to build the output binary. The only project dependecies are the **gRPC**, **protobuf** and the **OP-TEE client libraries**. CMake provides an easy way to build the sources, and it allows us to easily integrate the server application into Buildroot as a package. The compiled binary is called `attest_server`. The protobuf sources are automatically compiled with `protoc`, the sources are inside the `protos` folder.

[38] `https://github.com/protocolbuffers/protobuf` (last visited: 2018-10-22)

The server uses a secure TLS channel for communication, for which it needs a private key and a server certificate. The server certificate is signed by a root CA whose certificate can be shared between the client and the server. We automated the certificate and key generation by using the OpenSSL binary and a bash script.

```
#!/bin/sh
openssl genrsa -out root-key.pem 2048
openssl req -x509 -new -nodes -key root-key.pem -sha256 -days 365 \
        -out root-crt.pem -config root.cnf
openssl genrsa -out server-key.pem 2048
openssl req -new -out server.csr -key server-key.pem -config server.cnf -sha256
openssl x509 -sha256 -req -in server.csr -CA root-crt.pem -CAkey root-key.pem \
        -CAcreateserial -extensions san -extfile server.cnf -days 365 \
        -out server-crt.pem
openssl genrsa -out client-key.pem 2048
openssl req -new -out client.csr -key client-key.pem -config client.cnf -sha256
openssl x509 -sha256 -req -in client.csr -CA root-crt.pem -CAkey root-key.pem \
        -CAcreateserial -days 365 -out client-crt.pem
```

Listing 4.13. *OpenSSL certificate and key generation script*

Implemented methods

- **int do_pagefault()**
  This function opens the sysfs interface of the helper Linux kernel module, and writes a command to instruct it to load the process binaries into memory. This ensures that during the checksum calculation, the memory segments of every process can be hashed. The return value is 0 on no error.

- **Status ProcessList(context, request, reply)**
  This is the callback function of the ProcessList gRPC service. This function receives a ProcessListRequest and forwards it to the TA through the shared memory API. The matching TA function ID is `RA_CMD_GET_PSLIST`. This command receives no arguments and it sends back a stream of strings containing the process binary paths as part of the `ProcessListReply`.

- **Status ProcessHash(context, request, reply)**
  This is the callback function of the ProcessHash gRPC service. The function receives a ProcessHashRequest and forwards it to the TA through the shared memory API. The matching TA function ID is `RA_CMD_GET_PROC_HASH`. The command receives a single argument containing the full path of the requested binary, and it sends back a stream of strings containing the checksums of the found processes as part of the `ProcessHashReply`.

- **TEEC_Result InvokeTee(cmd_id, req_meta, res_meta, out, args, argsize)**
  This function registers a previously allocated memory segment for the shared memory API, and sends the given request (req_meta) with the given command (cmd_id). The results are stored in the res_meta parameter. If the command needs input argu-

ments it can be specified in the args array which has to be argsize long. The return value is `TEEC_SUCCESS` on no error.

- **int GetCredentials(ssl_opts)**
  The function tries to load the required keys and certificates into the ssl_opts object. The client certificate requirements can be set with the `ssl_opts.client_certificate_request` field.

4.5.8 Linux kernel module

Our kernel module provides a **sysfs**[39] interface to ensure the tasks' code segments are loaded into the memory. This is needed for calculating the checksums of the code segments, because Linux uses demand paging which is not configurable. The sysfs interface is under the `/sys/kernel/pslist` node. Currently only one interface function is implemented which is called `all`. The interface can be triggered by writing arbitrary data into this node (e.g. `echo 1 >> /sys/kernel/pslist/all`). The interface provides no output and it does not process any user input data. The kernel module should compile with any version of Linux between 4.6.3–4.12.8 (the module was tested only with these two versions).

The pagefault_task_range function

The function gets an address as a parameter and tries to map count bytes into the memory of the specified task. This is achieved by calling the `get_user_pages_remote` function without specifying a destination pointer. In this case the kernel ensures the pages are loaded into memory, and page table of the memory range does not contain zero pages. The address should be a virtual address in the context of the task's memory.

```
1  long pagefault_task_range(struct task_struct *task,
2                            unsigned long start_address, size_t count)
3  {
4      size_t page_count;
5      long user_pages;
6  #if LINUX_VERSION_CODE >= KERNEL_VERSION(4, 9, 0)
7      int lock = 1;
8  #endif
9      if (!task || !count)
10         return -EINVAL;
11     page_count = (count / PAGE_SIZE) + 1;
12 #if LINUX_VERSION_CODE < KERNEL_VERSION(4, 9, 0)
13     user_pages = get_user_pages_remote(task, task->mm, start_address,
14                                        page_count, 0, 1, NULL, NULL);
15 #else
16     user_pages = get_user_pages_remote(task, task->mm, start_address,
```

[39] `https://www.kernel.org/doc/Documentation/filesystems/sysfs.txt` (last visited: 2018-10-22)

```
17                                          page_count, 0, NULL, NULL, &lock);
18 #endif
19     if (IS_ERR_VALUE(user_pages))
20         return user_pages;
21     if (page_count != user_pages)
22         return -EFAULT;
23     return 0;
24 }
```

Listing 4.14. *pagefault_task_range function*

The pagefault_pslist function

This function iterates through the `task_struct` list which has its entry point as the `task` parameter and ensures that the code segment of every task is loaded into the memory by calling the `pagefault_task_range` function for each task it founds. If multiple errors occur during the iterations, only the last error code is returned.

```
1  long pagefault_pslist(struct task_struct *task)
2  {
3      struct list_head *list;
4      struct list_head *head;
5      struct task_struct *child;
6      struct mm_struct *mm;
7      long res = 0;
8      long tmp_res;
9      if (!task)
10         return -EINVAL;
11     mm = task->mm;
12     head = &(task->children);
13     for (list = head->next; list != head; list = list->next) {
14         child = list_entry(list, struct task_struct, sibling);
15         tmp_res = pagefault_pslist(child);
16         if (IS_ERR_VALUE(tmp_res))
17             res = tmp_res;
18     }
19     if (!mm)
20         return res;
21     tmp_res = pagefault_task_range(task, mm->start_code,
22                                    mm->end_code - mm->start_code);
23     if (IS_ERR_VALUE(tmp_res))
24         res = tmp_res;
25     return res;
26 }
```

Listing 4.15. *pagefault_pslist function*

4.5.9 Generating kernel structures

Several data structures on Linux are platform and configuration dependent and they can change from one kernel version to another. This means that memory forensics tools must deal with every compiled kernel differently. Our tool provides an easy way to generate a C header file with the needed structures, which can be used to extract usable data from the memory directly. Volatility framework[40] does parse kernel structures, but it is not fitting for our purpose because it cannot generate a C header from the parsed structures.

The common thing that both our tool and Volatility does is it extracts DWARF[41] debug information from a compiled binary[42] which has the necessary data structures. This binary can be either compiled on the target, or it can be compiled as a dummy kernel module for example in Buildroot.

Our python script is based on the open-source **pyelftools**[43] package with some modifications to support our target platform (ARM). The script is called **dwarfparse.py**[44], which shows a help text if the `-h` command line argument is given.

Script documentation

The script needs an input file which is an ELF binary (in our example it is a dummy kernel module which includes the needed kernel headers). The script can either output the header to the standard output or it can generate C header files if the `-o` parameter is supplemented. This parameter expects a directory path where the header files will be written to. If the `-s` parameter is supplemented, the script will not generate the headers, it will only parse the debug data and print out statistics about it.

The output files are named `cu_<xx>.h` and `cu_<xx>.json` for every compilation unit[45] it founds in the debug information. The `.h` files are the C headers, and the `.json` files are generated as a more readable debug info format.

4.6 Secure Communication

In this section, we will introduce our improvements on the SKS proposal, by going over the various problems we have encountered while trying to use it with OpenSSH.

[40] `https://github.com/volatilityfoundation/volatility` (last visited: 2018-10-22)
[41] `https://en.wikipedia.org/wiki/DWARF` (last visited: 2018-10-22)
[42] `https://github.com/volatilityfoundation/volatility/wiki/Linux#creating-vtypes` (last visited: 2018-10-22)
[43] `https://github.com/eliben/pyelftools`(last visited: 2018-10-22)
[44] `https://github.com/realmoriss/dwarfparse` (last visited: 2018-10-22)
[45] `https://www.cs.auckland.ac.nz/references/unix/digital/AQTLTBTE/DOCU_015.HTM` (last visited: 2018-10-22)

4.6.1 Finding missing functions with pkcs11-tool and SSH

As mentioned in Section 3.6.1, the PKCS#11 standard is rather big, so often only parts of it are implemented. This is even more true for SKS, since it is still heavily under development. Because of this, we decided to enumerate the necessary additions, by trying to use ssh with SKS and checking where it would fail. Then implement or fix the specific functionality that caused the error and try again. This way we did not have to understand the whole codebase at once, or delve into the OpenSSH implementation either, rather we were able to gradually build up knowledge about these things as certain issues came up. As most of the API calls were already implemented in SKS, there were only a few cases where we had to implement a function from the ground up. In most of the cases, we only had to improve on already existing functions that were not working correctly in some situations. Of course, we collaborated with Etienne while working on this project, and contributed these additions and improvements to the project on GitHub. It might be worthy of mention here, that while we described the above process specifically with SKS in mind, our plans would have been the same, even if SKS did not exist. We would have patched a soft token, to print what PKCS#11 API calls are made while we use ssh and select a subset of the API to implement based on that. Doing this would have been likely harder to implement, so it is great that we had an already existing project to build on.

Specifically, we inserted the following line into the beginning of the handling function of each of the API calls, to easily get a sense of what is happening in the background when we give a terminal command:

```
1     printf("Function %s Entered\n", __func__);
```

Listing 4.16. *Print name of function that was called*

where `__func__` is an implicitly declared identifier in C that expands to a cstring that has the name of the current function inside. A similar print can be placed at the end of the function to see when the function exits.

The following sections are the issues we encountered with this technique, roughly in chronological order. In each section, we detail the problem encountered and any relevant parts of the standard, then explain the solution we implemented for it.

4.6.2 Fixing the function list

One of the first problems manifested right at the beginning, when we started to explore the SKS soft token. The pkcs11-tool provides commands to get information about the devices that can be used with PKCS#11. The command that had problems was `--list-slots`. Here is the output that it generated, slightly edited for clarity:

```
# pkcs11-tool --module /lib/liboptee_cryptoki.so -L
    Function C_GetFunctionList Entered
    Function C_GetFunctionList End
    Function C_Initialize Entered
    Function C_Initialize End
    Function C_GetSlotList Entered
    Function C_GetSlotList End
    Function C_GetSlotList Entered
    Function C_GetSlotList End
Available slots:
Slot 0 (0x0):
    Function C_GetSlotInfo Entered
    Function C_GetSlotInfo End
OP-TEE SKS TA
    Function C_InitToken Entered
    Function C_InitToken End
C_GetTokenInfo() failed: rv = CKR_HOST_MEMORY
Slot 1 (0x1):
    Function C_GetSlotInfo Entered
    Function C_GetSlotInfo End
OP-TEE SKS TA
    Function C_InitToken Entered
    Function C_InitToken End
C_GetTokenInfo() failed: rv = CKR_HOST_MEMORY
Slot 2 (0x2):
    Function C_GetSlotInfo Entered
    Function C_GetSlotInfo End
OP-TEE SKS TA
    Function C_InitToken Entered
    Function C_InitToken End
C_GetTokenInfo() failed: rv = CKR_HOST_MEMORY
    Function C_Finalize Entered
    Function C_Finalize End
```

Listing 4.17. *PKCS#11 functions called when listing slots*

Based on the output, it seemed that for some reason, the pkcs11-tool thinks that it is calling the `C_GetTokenInfo`, but in reality the `C_InitToken` function is getting called. The reason for this error is rooted in the way the client applications acquire the function pointers to the PCKS#11 API, with the `C_GetFunctionList` method. This method returns a pointer to a `CK_FUNCTION_LIST` structure, which in turn contains function pointers to all the PKCS#11 API methods that the library has—even the unimplemented ones have to have stubs. According to the standard, this is inteded to be used in a way, so that applications can use shared PKCS#11 libraries easier and faster and to enable using more than one PKCS#11 library at once. In our case, some of the function pointers in the `CK_FUNCTION_LIST` structure were registered in the wrong order, which caused the pkcs11-tool to mix up the calls. The reason that this issue did not surface in the regression tests is probably that the same, out of order structure was used in the tests as well, meaning it could work together with the faulty SKS implementation, since they were wrong in the same way. Correcting the order of the functions in the structure corrected this behaviour.

4.6.3 Determining required buffer size

This problem came up while we were trying to gather information about the SKS soft token with the pkcs11-tool as well. When trying to list the available slots or mechanisms (`C_GetSlotList` and `C_GetMechanismList`), the program displayed the following error:

```
error: PKCS11 function C_GetSlotList(NULL) failed:
    rv = CKR_BUFFER_TOO_SMALL (0x150)
Aborting.
```

Listing 4.18. *Output of pkcs11-tool when listing slots*

After investigating a bit, we found that the pkcs11-tool was calling the respective API functions with a buffer pointer of `NULL` and zero length. According to the standard, this is done, in order to determine what size the buffer needs to be, in order for the result of the request to fit in it. The token should return with `CKR_OK` and set the buffer length for the necessary value. SKS already sets the length to the correct value, the only mistake was the wrong return value. We modified the logic, so that it returns with `CKR_OK` if the buffer pointer is `NULL` and which resolved the problem.

4.6.4 Key type inference in C_GenerateKeyPair

After exploring the soft token, the first step toward using for authentication with ssh is generating a new key to use on the token. We chose to use an RSA key, because we already had a little preliminary experience with RSA and it was also supported to some extent in SKS. We used the pkcs11-tool with the following command to generate a 2048 bit RSA key with pkcs11-tool:

```
pkcs11-tool --module /usr/lib/libsks.so --label testtoken      \
    --login --pin 12341234 --keypairgen --label testkey        \
    --key-type rsa:2048
```

Listing 4.19. *Generate RSA key pair with pkcs11-tool*

However, the command resulted in pkcs11-tool displaying an error, stating that the `C_GenerateKeyPair` returned with `CKR_TEMPLATE_INCOMPLETE`. After some investigation, we found that the reason for this is that pkcs11-tool is not setting the `CK_KEY_TYPE` parameter for the call. This is because the standard states that it is not necessary to set this parameter, since it is implicit in the key generation mechanism, that is set with the `CK_MECHANISM` parameter. The reason for the error, is that this inference was not implemented in SKS.

Originally, we wanted to imlement the inference in the TA, however this was not possible, because the checks that determine whether an API call is well formed (i.e., in our case if it has every necessary parameter) are among the very first to run when the TA is called,

and by the time we could insert the missing parameter, the call would already be rejected. Thus, we had to implement this in the shared library in Normal World.

In particular, what we decided to do, is scan the API call, before forwarding it to the TA, and if the `CK_KEY_TYPE` parameter is not specified in it, try to infer the missing information and insert it, but leave it as it is, if it is already present. Currently, the only inference is for the RSA key type, but other inferences can be very easily added, by inserting a new case in the switch case block that handles this logic.

4.6.5 C_FindObjects*

After generating the key pair we will use with ssh, we need to export the public key, so we can specify it for the server to accept it. At first we exported the RSA public key with the pkcs11-tool, but after applying all the changes listed here, it is also possible to use ssh-keygen to do this, which is also more convenient, because ssh-keygen takes care of the format conversions. Here is the commands necessary to export with pkcs11-tool:

```
pkcs11-tool --module /usr/lib/libsks.so -r --type pubkey    \
            --label testkey > pub.key
openssl pkey -inform DER -pubin -in pub.key -out pub.pem
```

Listing 4.20. *Extract public key with pkcs11-tool*

And directly using ssh-keygen:

```
ssh-keygen -D /usr/lib/libsks.so -e >> .ssh/authorized_keys
```

Listing 4.21. *Extract public key with ssh-keygen*

In order to extract the public key, we first need to acquire a handle for the public key object inside the token. This can be done with the `C_FindObjectsInit`, `C_FindObjects` and `C_FindObjectsFinal` calls. The search parameters can be specified in `C_FindObjectsInit`, which also initializes the search. With each call to `C_FindObjects` the client can query one result of the search. To finish the active search, `C_FindObjectsFinal` has to be called.

In our case, the client application uses `C_FindObjectsInit` with no search parameters supplied, in which case all objects should be matched. This edge case was not correctly implemented in SKS, which meant that the client application could not find the public key. After adding code that correctly handled this case, the objects were returned to the client.

4.6.6 Reading the public key with C_GetAttributeValue

Once they have the handle to the right object, the programs in the previous section need to extract the value of the public key object. This can be done with the `C_GetAttributeValue`

function. This API call was not implemented before in SKS, so we had to build it from the ground up. When implementing this, there were two main aspects that had to be considered in order for this function to work correctly.

First, it is very important, that it is impossible to recover any values that are sensitive or unextractable. To accomplish this, we read the relevant parts of the standard, and also consulted the SoftHSM source code for a reference implementation, and created a function in the SKS TA, that decides whether a certain attribute can be disclosed.

The second is more related to our particular circumstances: when passing parameters in SKS from the Normal World to Secure World, they are serialized, and certain constants are translated from their respective numerical values in the PCKS#11 standard to the ones used internally by SKS. This serialization was implemented, however the deserialization and translation back to Normal World values was not, so it had to be added. We implemented these functionalities based on what their counterparts did, essentially reversing the serializing algorithm to get the deserializing algorithm.

In order to fully understand how the `C_GetAttributeValue` function works, a bit more information about PKCS#11 objects is necessary. In essence every object in PKCS#11 is a collection of attributes, for example `CKA_EXTRACTABLE` is a boolean attribute, and defines whether the given object is extractable. Another example is `CKA_MODULUS` that holds the modulus N in an RSA public key object. To define it more precisely, the `C_GetAttributeValue` function obtains the value of one or more attributes of an object, in our case the modulus and public exponent attributes. The specification gives an algorithm to describe how to implement the `C_GetAttributeValue` function. Using the functionality described in the previous two paragraphs we implemented this algorithm, and successfully exported the public key.

4.6.7 Authenticating with OpenSSH

Helping OpenSSH find the private key

Having impelmented all of the above, we can set up an environment where we can actually start trying to use OpenSSH to establish a connection. We have a private key in our token, that is unextractable and we have inserted the public counterpart of that key into the `.ssh/.authorized_keys` file, so the server will accept it for authentication. For the sake of simplicity, we tested this on a single host, by connecting to localhost:

```
ssh root@localhost -I /usr/lib/libsks.so
```

Listing 4.22. *Connect as ssh client using SKS*

However, at first, when we gave this command, we were greeted with a password prompt. This meant, that the ssh server didn't accept any of the keys that the client offered, so it defaulted to authentication with a password. After some investigation, it became clear, that the OpenSSH client could not find the key pair on the SKS soft token. There are two

reasons for this: the first is related to the login functionality in PKCS#11. It is possible to log in to a PKCS#11 token by supplying a PIN, and as one would expect, a logged in user can do some things that an unauthenticated user cannot. One of these differences is, that when searching for objects with the `C_FindObjects*` functions, the private key is only among the results for an autenticated user. However, when we entered the ssh command, we were not prompted for a PIN number. OpenSSH decides whether it should prompt for a login PIN for the token, based on a flag in the token called `CKFT_LOGIN_REQUIRED`. If this flag is set, it means that the token has functionality that is only available to authenticated users. This flag was not set for the SKS token; after adding it, OpenSSH correctly prompted for the PIN and logged in to the token.

The second problem was caused by the method that is used by OpenSSH to identify what public keys and private keys on the token form key pairs. In PKCS#11, RSA key objects have an attribute called `CKA_ID`, that was not set by the `C_GenerateKeyPair` function in SKS, because the specification does not require it to be set. However, this is what the specification does say about `CKA_ID`:

> The CKA_ID attribute is intended as a means of distinguishing multiple public-key/private-key pairs held by the same subject (whether stored in the same token or not). (Since the keys are distinguished by subject name as well as identifier, it is possible that keys for different subjects may have the same CKA_ID value without introducing any ambiguity.)
>
> It is intended in the interests of interoperability that the subject name and key identifier for a certificate will be the same as those for the corresponding public and private keys (though it is not required that all be stored in the same token). However, Cryptoki does not enforce this association, or even the uniqueness of the key identifier for a given subject; in particular, an application may leave the key identifier empty. [31]

Reading this, we assumed that OpenSSH must be trying to use the `CKA_ID` to pair the public and private key object, but since the attribute is not set in our generated key pair, it discards these objects. To correct this, we amended the code in the SKS TA that handled generating key pairs with a section, that generates a random number and sets it as `CKA_ID` for both of the keys that are generated.

The CKM_RSA_PKCS mechanism

An OpenSSH client creates a cryptographic signature with the private key of the user to authenticate to the server. In PKCS#11 this can be accomplished with the `C_SignInit`, `C_Sign`, `C_SignUpdate` and `C_SignFinal` API calls, depending on whether the the signing is done as single-part or as multi-part, with updates. The scheme that will be used is specified in the `C_SignInit`, with a `CK_MECHANISM_TYPE` type parameter. OpenSSH uses

the CKM_RSA_PKCS mechanism, which is a multi-purpose mechanism based on the RSA PKCS#1 standard [24]. Among other things, it supports single-part encryption, decryption, signatures and verification. From here on, we will be writing about the case when it is used for signing. The exact scheme that this mechanism is based on is the RSASSA PKCS#1 v1.5, which in turn uses EMSA PKCS#1 v1.5 encoding. In the PKCS#1 specification, a rough outline of signing would be the following: hash the data that is to be signed, then use the paddings defined in the standard to generate the encoded message, where the interesting part for us is that the resulted encoded message will contain (with padding) the identifier of the hash algorithm used and the hash itself. Then apply the necessary cryptographic primitives to the encoded message to get the signature itself. However, the PKCS#11 specification states, that the CKM_RSA_PKCS mechanism only corresponds to the part of PKCS#1 v1.5 that involves RSA and does not compute a message digest. In accordance with this, it takes a message digest (generated by the client application) as input, and it does not and can not include the identifier of the hash algorithm that was used. In SKS, signing and verification with the CKM_RSA_PKCS mechanism was not implemented, because there is no corresponding function in the GPD TEE Cryptographic Operations API. There are algorithm identifiers for signing a message digest with PKCS#1 RSASSA, but those also require a hash algorithm, since the format of these is TEE_ALG_RSASSA_PKCS1_V1_5_<hash_algorithm>, so for example TEE_ALG_RSASSA_PKCS1_V1_5_SHA1 would expect a message digest created by SHA1 and include its algorithm identifier in the signature.

OP-TEE extension TEE_ALG_RSASSA_PKCS1_V1_5

Fortunately, since OP-TEE is open source, it is possible to modify the source code and extend the API defined in the GPD specification. Behind the internal core API in OP-TEE, the cryptographic operations are implemented using LibTomCrypt[46], an open source cryptographic library written in C. So the solution is self evident from here: define a new algorithm identifier in the API and connect it to the right LibTomCrypt function. The only problem was, that after looking at the source of LibTomCrypt in OP-TEE, it seemed that there was no option to do this in LibTomCrypt either, since even the LibTomCrypt API required a hash algorithm to be specified. At the same time, we noticed, that the timestamps in the LibTomCrypt source that was included in OP-TEE were somewhat outdated, so we inspected the GitHub repository of LibTomCrypt, where we found that the newer version does support the same mode of operation that we need. To be exact, they added a new possible value for the padding parameter in the rsa_sign_hash_ex function, called LTC_PKCS_1_V1_5_NA1 that allows this. At first, we wanted to update the whole LibTomCrypt library in OP-TEE to the latest version, but after comparing the differences between the two versions, we realized that OP-TEE had many minor modifications (e.g., for optimization purposes) that would have made the update process very lengthy and prone to errors, so we opted to only update rsa_sign_hash_ex and rsa_verify_hash_ex

[46] https://www.libtom.net/LibTomCrypt/ (last visited 2018. 10. 22.)

functions, while being very careful to keep the minor changes added by the OP-TEE developers in them. With the updated version of LibTomCrypt, we were able to add the new algorithm identifier to the Cryptographic Operations API and in turn connect it with the `CKM_RSA_PKCS` mechanism. And with that, OpenSSH was finally able to authenticate and connect to the server.

We created a regression test case for the newly added TEE API method, by modifying test vectors already used by the OP-TEE test suite (originally from the US NIST Computer Security Resource Center[47]). Specifically, we kept the plaintext and key pair of the test vector and generated the signature with a python script. Then we inserted these values into their place and added them to the list of test vectors.

As a side note here, the question has arisen in us, whether it makes any sense to even have such a mechanism as `CKM_RSA_PKCS`, since if the hash algorithm is not included in the signature, how would the receiver of the message know how to verify the the message with the signature, without knowing how to create the right message digest from it. A simple possibility that gets around this problem, is if both parties in a communication follow a protocol that specifies what hash function they have to use, so the hash function—although not explicitly specified in the signature—is known to the receiver. We have also been able to find some real life examples of cases where this functionality is required. The first one is mentioned in the commit message of the change[48] in which the `LTC_PKCS_1_V1_5_NA1` option was added to LibTomCrypt and according to the author, the early versions of the SSL protocol did not set the hash algorithm identifier in `SERVER_EXCHANGE_MESSAGE` messages, hence the feature was added to LibTomCrypt to support the format. The same feature was requested[49] by someone else who was also implementing a PKCS#11 soft token. The last example is in the `pyca/Cryptography` python module, where it was requested in an issue[50] on GitHub by a contributor for the TOR project, because it was necessary for compatibility reasons.

[47] `https://csrc.nist.gov/projects/cryptographic-algorithm-validation-program/digital-signatures#rsavs` (last visited 2018. 10. 22.)

[48] `https://github.com/libtom/libtomcrypt/commit/aa4bae5ae9a2` (last visited 2018. 10. 22.)

[49] `https://lists.randombit.net/pipermail/botan-devel/2008-November/000696.html` (last visited 2018. 10. 22.)

[50] `https://github.com/pyca/cryptography/issues/3713` (last visited 2018. 10. 22.)

Chapter 5

Evaluation

5.1 Secure Boot

5.1.1 Basic functioning

The implemented secure boot process was tested to ensure that it works properly and as described above. The test methodology was to conduct such test cases, which together cover every possible path with every possible decision in all of the flowcharts and in the source files.

The first test case checks if there is no firmware image, then U-Boot prints a message, saying that the device is bricked, and resets it continuously, waiting for a valid image and preventing interactive mode.

The second test case checks if the signature is broken on the firmware image, then U-Boot refuses to boot it, and resets the device continuously, waiting for a valid image and preventing interactive mode.

The third test case checks if the signature is correct on the firmware image, then U-Boot boots it.

The fourth test case checks if the kernel is booted but there is no root file system, then the operating system prints a message, saying that the root file system is missing, and reboots.

The fifth test case checks if the kernel is booted but the integrity of the root file system is broken, then the operating system prints a message, saying that the integrity of the root file system is broken, and reboots.

The sixth test case checks if the kernel is booted and the integrity of the root file system is correct, then the operating system switches from the initial RAM file system to the root file system, and continues to run.

All of the test cases succeeded, meaning that the described secure boot process is correctly implemented.

5.1.2 Performance

Security features always have a negative impact on performance. Since both the whole firmware image and the whole root file system have to be verified, the boot time heavily depends on their sizes. On the Raspberry Pi 3 Model B with a micro SD HC CLASS 10 card in it and with a minimalist Linux based operating system together with OP-TEE the following results were experienced. U-Boot boots the firmware image just barely noticeably slower than the same firmware image without verification. And the verification of the root file system (hashed with SHA-256) is done under ten seconds.

5.1.3 Security

Given any kind of reliable hardware root of trust, and that the signing private keys are kept secure, the described secure boot process only boots the device into a known and secure state, owing to the fact that if the verification of any component fails, the boot process is halted.

5.1.4 Limitations

As already mentioned before, the Raspberry Pi platform misses a lot of security functions, including the hardware root of trust, so this proof of concept implementation is cannot be considered secure without one. Furthermore, by the time of writing, U-Boot only supports SHA-1 hashing, and 2048-bit RSA signing key pairs, which are not as secure as their other variants.

5.2 Secure Firmware Update

5.2.1 Basic functioning

The implemented secure firmware update process was also tested to ensure that it works properly and as described above. The test methodology was the same as by the secure boot process, to conduct such test cases, which together cover every possible path with every possible decision in all of the flowcharts and in the source files.

The first test case checks if there is no firmware image, then U-Boot prints a message, saying that the device is bricked, and resets it continuously, waiting for a valid image and preventing interactive mode.

The second test case checks if there is no update firmware image, but both the self-test and the signature are correct on the stable firmware image, then U-Boot boots the stable firmware image.

The third test case checks if the version-number is missing from the update firmware image, but both the self-test and the signature are correct on the stable firmware image, then U-Boot boots the stable firmware image.

The fourth test case checks if there is no boot-log, then U-Boot prints a message, saying that the device is bricked, and resets it continuously, waiting for a valid boot-log and preventing interactive mode.

The fifth test case checks if the self-test, the watchdog and the signature are correct on the update firmware image, then U-Boot boots the update firmware image.

The sixth test case checks if the self-test is correct, but the watchdog failed on the update firmware image, but both the self-test and the signature are correct on the stable firmware image, then U-Boot boots the stable firmware image.

The seventh test case checks if the self-test and the watchdog are correct, but the signature is broken on the update firmware image, but both the self-test and the signature are correct on the stable firmware image, then U-Boot adds an indicating parameter to the boot-arguments and boots the stable firmware image.

The eighth test case checks if the integrity check failed on the root file system of the update firmware image, but both the self-test and the signature are correct on the stable firmware image, then U-Boot adds an indicating parameter to the boot-arguments and boots the stable firmware image.

The ninth test case checks if the self-test failed on the update firmware image, but both the self-test and the signature are correct on the stable firmware image, then U-Boot boots the stable firmware image.

The tenth test case checks if the self-test and the watchdog are correct, but later the signature is broken on the update firmware image, and by that time the stable firmware image should not exist, then U-Boot prints a message, saying that the device is bricked, and resets it continuously, waiting for a valid image and preventing interactive mode.

The eleventh test case checks if the kernel is booted but there is no root file system, then the operating system prints a message, saying that the root file system is missing, writes the boot-log accordingly, and reboots.

The twelfth test case checks if the kernel is booted but the integrity of the root file system is broken, then the operating system prints a message, saying that the integrity of the root file system is broken, writes the boot-log accordingly, and reboots.

The thirteenth test case checks if the kernel is booted and the integrity of the root file system is correct, but the self-test failed, then the operating system switches from the initial RAM file system to the root file system, prints a message, saying that the self-test failed, writes the boot-log accordingly, and reboots.

The fourteenth test case checks if the kernel is booted and both the integrity of the root file system and the self-test are correct, then the operating system switches from the initial

RAM file system to the root file system, writes the boot-log accordingly. The operating system deletes and logs any different firmware, then starts the update-manager, and continues to run.

The fifteenth test case checks if the kernel is booted and both the integrity of the root file system and the self-test are correct, and the indicating parameter of an integrity failed update firmware is in the kernel command line (including the boot-arguments), then the operating system switches from the initial RAM file system to the root file system, writes the boot-log accordingly and deletes the integrity failed update firmware. The operating system deletes and logs any different firmware, then starts the update-manager, and continues to run.

All of the test cases succeeded, meaning that the described secure firmware update process is correctly implemented.

5.2.2 Performance

The secure firmware update process builds on the secure boot process, therefore it is affected by the same kind of performance decrease, as well as by its own slowing factors. Again it was tested on the Raspberry Pi 3 Model B with a micro SD HC CLASS 10 card in it and with a minimalist Linux based operating system together with OP-TEE. In U-Boot the slowest boot scenario is, when two firmware images have to be checked, but it is still within a few seconds. The verification of the root file system (again, hashed with SHA-256) however, takes more time, since it has to be copied from the download place to the active root file system partition (the firmware image must be copied to the boot partition before the reboot), with the above setup it takes around a half minute. As mentioned before the xtest test suite of OP-TEE is used as self-test. And since xtest runs for a few minutes, it slows down the startup process significantly.

5.2.3 Security

In addition to the secure boot process' security requirements, the logs must be kept secure and must have limited write access, in order to achieve the described secure firmware update process.

5.2.4 Limitations

The limitations are the same as detailed by the secure boot process, because a physical write protection for the logs can also be derived from a hardware root of trust.

5.3 Security hardened firmware/operating system

5.3.1 Basic functioning

Since testing was a core part of the implementation process, there was no need to conduct additional test cases, as the implementation was only considered done, when the whole system worked as wanted.

5.3.2 Performance

Besides the compatibility problem, the main reason, why most of the security hardening features are not widely used, is the performance drop they cause. Although to notice their performance drop, definitive use cases/applications are needed, which are out of the scope of this project, as this project focuses on securing a general embedded platform.

5.3.3 Security

The reason to use the security hardening features, which are designed and implemented by experts, is to improve a systems security.

5.3.4 Limitations

The other core part of the implementation process was to exclude those security hardening features, that are somehow not compatible with other parts of the project.

5.4 Remote Attestation and Integrity Monitoring

We tested the Remote Attestation and Integrity Monitoring algorithms, Trusted Application, Server Application, and remote client application throughout their development process. The tests discussed in this section are mostly using the remote client application to interact with the Server Application and the TA. In the test environment the Server Application and the TA are running on the Raspberry Pi 3 Model B embedded device and the remote client application is located on one of the development computers. These two devices are connected on a Local Area Network, although our implementation also works over the Internet. Throughout the shown tests, the host name of the Raspberry Pi 3 is `rpi`.

5.4.1 Basic functioning

The testing methodology for the functional tests—namely for process listing and process hash calculation—are described below, followed by the conducted tests. We tried to cover every aspect of the Trusted Application and the client software functionality with our tests.

Process listing is tested, by first listing the running processes in Normal World, on the Raspberry Pi 3, with the following command:

```
# ps -o comm,vsz,stat | awk '$2 != 0 && $3 !~ "Z"'
```

This lists every running process, except kernel threads (and zombie processes—hence the "`!~ "Z"'`" argument). Then the list is also generated in Secure World with the TA, by using the remote attestation client software. Note: `ps` and `awk` should be omitted, since they are running only during the listing.

Process hash calculation is tested, by calculating the hash of the code segment of a binary on the Raspberry Pi 3 with the following commands:

```
$ size=$(readelf -l out/target/bin/busybox | egrep "LOAD *0x000000" | awk -Wposix
    '{printf("%d", $5)}')
$ dd if=out/target/bin/busybox bs=1 count=$size | sha256sum
```

The code section size of a binary can be acquired using the `readelf` command. First, we list the different sections and get the size (`FileSiz`, the 5th field) for the code section—the one with offset `0x0` and type `LOAD` [27]. The offset is zero, since the section—corresponding to the one specified in the `mm_struct` by the `code_start` and `code_end` fields—is located at the beginning of the binary, and the section type is loadable, as the kernel loads it.[1] Then `dd` can be used to pipe the code segment into `sha256sum` to calculate the SHA256 hash. We then compare the digest with the one calculated by the TA in Secure World, returned by the remote attestation client software.

Getting the process list

The initial process list on the board is the following (`ps` and `awk` omitted):

```
# ps -o comm,vsz,stat | awk '$2 != 0 && $3 !~ "Z"'
COMMAND          VSZ  STAT
init             2364 S
syslogd          2364 S
klogd            2364 S
tee-supplicant   1968 S
sshd             4616 S
udhcpc           2364 S
sh               2492 S
attest_server     54m S
```

This test checks if the process list shows the running processes correctly, by comparing the initial process list state with the one generated after a new process is started. Getting the process list with the client program:

[1] `www.skyfree.org/linux/references/ELF_Format.pdf` (last visited 2018–10–22)

```
$ ./remote-attest-client.py --host rpi pslist
Binary: /usr/sbin/sshd (1 entries)
Binary: /usr/sbin/tee-supplicant (1 entries)
Binary: /usr/bin/attest_server (1 entries)
Binary: /bin/busybox (5 entries)
```

Starting a new process on the board : `nano &`
Getting the current process list on the board:

```
# ps -o comm,vsz,stat | awk '$2 != 0 && $3 !~ "Z"'
...
nano             2076 T
```

Getting the process list again with the client program:

```
$ ./remote-attest-client.py --host rpi pslist
...
Binary: /usr/bin/nano (1 entries)
```

Calculating a process hash

The test checks if the TA correctly calculates the hash for the code segment of BusyBox. The result is checked with the locally generated hash of the binary. Generating the hash of the BusyBox binary at the remote client side:

```
$ size=$(readelf -l out/target/bin/busybox | egrep "LOAD *0x000000" | awk  -Wposix
    '{printf("%d", $5)}')
$ dd if=../build/out/target/bin/busybox  bs=1 count=$size | sha256sum
...
373fce300530245c66a25b6ea37fe160f31bd4470248b223829e3db8c9225154  -
```

Getting the hash with the remote attestation client program:

```
$ ./remote-attest-client.py --host rpi pshash /bin/busybox
Binary: /bin/busybox, Hash:
373fce300530245c66a25b6ea37fe160f31bd4470248b223829e3db8c9225154 (5 entries)
```

Sending invalid request to the TA

The test checks the TA behavior when invalid requests are sent. The `buf_num` metadata representing the request argument number is changed to a larger value (`50`) in the remote attestation server running on the board. The expectation is that the TA does not process the request. Sending a hashing request with the remote attestation client program:

```
$ ./remote-attest-client.py --host rpi pshash /usr/sbin/sshd
status = StatusCode.CANCELLED
```

The serial output of the TA on the board (error code is `TEE_ERROR_BAD_PARAMETERS`):

```
D/TC:0 tee_ta_invoke_command:625 Error: ffff0006 of 4
```

Too long path in request

The test checks the TA behavior when the process path in the request is longer than the maximum defined in `RA_MAX_PROC_PATH_LEN` (`512`). The expectation is that the TA

does not process the request. Sending a hashing request with the remote attestation client program:

```
$ ./remote-attest-client.py --host rpi pshash $(python2 -c 'print "A"*1024')
status = StatusCode.CANCELLED
```

The serial output of the TA on the board (error code is `TEE_ERROR_BAD_PARAMETERS`):

```
REQ: (ver|fname|args|nonce)
(1|pshash|AAAAA...AAAA|4770481557905811)
sha256(REQ)
d238badfe0c9c46859263f6dc24adbb60a0988b81d1df14a5de054c0f5934156
D/TC:0 tee_ta_invoke_command:625 Error: ffff0006 of 4
```

Key Provisioning

The test checks if the TA correctly stores and uses the stored RSA keys. Removing the already existing keys from Secure Storage on the board:

```
# rm -rf /data/tee/*
```

Calling the Key Provisioning function of the TA on the board:

```
# remote_attestation
...
D/TA:  Saving key (ra_keypair) to secure storage...
D/TA:  Saving key (ra_attester_pubkey) to secure storage...
```

Requesting process hash with remote attestation client program:

```
$ ./remote-attest-client.py --host rpi pshash /bin/busybox
Binary: /bin/busybox, Hash: 373
    fce300530245c66a25b6ea37fe160f31bd4470248b223829e3db8c9225154 (5 entries)
```

Unknown process hash/path

This test checks the behavior of the client when an unknown process hash/path is sent back by the server. The result is checked against the known process hashes/paths listed in the `process_db.csv` file. The client shows a warning message if there is no matching entry in the database:

```
./remote-attest-client.py --host rpi3 pshash /usr/bin/attest_server
```

Output:

```
Warning: Could not find matching digest for /usr/bin/attest_server
Binary: /usr/bin/attest_server, Hash: 3094[..]dee0 (1 entries)
```

Requesting a process hash for an invalid path

If the client requests a process hash for a program which is not running, the server will send back an empty list. The output shows an error message for this:

```
./remote-attest-client.py --host rpi3 pshash /usr/bin/attest_server_wrong
```

Output:

```
Error: Could not find the binary called /usr/bin/attest_server_wrong.
```

Public keys do not match

The public key of the server is stored on the client side. If the public key does not match the private key, the response can not be verified. The following test shows this scenario:

```
./remote-attest-client.py --host imx pshash /bin/busybox
```

Output:

```
Error: Signature verification failed.
```

Multiple hash values for a single binary

An instance of `nano` is started on the server, then the binary is deleted while the program is running. Another binary is copied to the same path and it is started too. The server should send back two different digests and the client should prompt a warning:

```
./remote-attest-client.py --host imx pshash /usr/bin/nano
```

Output:

```
Binary: /usr/bin/nano, Hash: a80a[..]865f (1 entries)
Warning: Could not find matching digest for /usr/bin/nano
Binary: /usr/bin/nano, Hash: 61e4[..]99e4 (1 entries)
Warning: Multiple hash values exist for /usr/bin/nano.
```

5.4.2 Performance

We tested the performance of the Trusted Application, by measuring the time it takes to assemble the list of running processes and to calculate the hash of a process.

The robustness of the solution was also tested multiple ways. First, as shown in section Large number of processes, many processes were started on the board, and their hash digest and process list was requested. Our TA and Server Application successfully handled and executed the request, and sent back around 300 hash values and binary paths in the response.

The other test aimed to simulate multiple incoming requests in a short time frame. When our implementation relied on global variables to hold large structures (e.g., `task_struct`), the TA operation was unreliable, the requests were not processed, the TA exited with panic messages. After we eliminated the usage of global variables, the TA successfully executed every request it received. Around 1000 requests was sent with 10 ms delay between each of them (we used normal shell scripts to send the requests).

We could not perform scalability testing, since we did not have access to a significant number of Raspberry Pi devices, and since we did not port the TA to QEMU, it was

not an option to use virtual machines. Otherwise, other possible problems can arise when dealing with multiple devices. For example, key provisioning must be solved efficiently—depending on the use case, the key pairs for the devices must be generated and installed somehow, probably during manufacturing. Also, other type of communication protocols might be more feasible for large scale deployment, like publish-subscribe or message broker solutions (e.g., RabbitMQ).[2]

Large number of processes

The test checks if the TA correctly works with large number of running processes (>= 200). The only limitation is the shared memory buffer given to the TA (by default 4 KB), it is increased to 20 KB in the current case. Starting processes on the board:

```
# for i in $(seq 1 300); do nano & >/dev/null ; done
```

Getting the process list with the remote attestation client program:

```
$ ./remote-attest-client.py --host rpi pslist
...
Binary: /usr/bin/nano (300 entries)
...
```

Getting the hash of `nano` with the remote attestation client program:

```
$ ./remote-attest-client.py --host rpi pshash /usr/bin/nano
Binary: /usr/bin/nano, Hash:
    a80a2f69b368314b2d3d1cdb1c1e93c011df4afa2f47f9353afaf13adb22865f (300 entries)
```

5.4.3 Security

The current implementation in OP-TEE, the Trusted Applications and the Secure Storage objects are stored in the Normal World file system as encrypted files owned by the root user. Their integrity are checked by OP-TEE kernel each time they are accessed or loaded.[3] Should any integrity check fail, OP-TEE does not start the applications or use the stored objects. Also, we changed the key pair used for signing the TAs (the default is located in the OP-TEE git repository, and should be used for testing only). We tested if the TA only uses keys that are not modified, as described in section Basic tampering with stored keys.

The remote attestation protocol is a proof of concept protocol which is not well suited for practical application. A more robust protocol should be implemented with better key management, client authentication and a more sophisticated defense against message replay attacks. The protocol is not proved to be secure so it is important to put more work on this aspect of the protocol design.

[2] `https://www.rabbitmq.com/` (last visited on 2018–10–22)

[3] `https://github.com/OP-TEE/optee_os/blob/master/documentation/optee_design.md#normal-or-secure-storage-trusted-applications` (last visited on 2018–10–22)

Basic tampering with stored keys

The test checks the TA behavior when the existing RSA keys in the Secure Storage are modified (or missing). The expected behavior is that the TA does not perform the requests sent to it. Modifying the first byte of already existing (and previously tested) keys in Secure Storage on the board:

```
# echo -ne \\xFF | dd conv=notrunc bs=1 count=1 of=/data/tee/1
# echo -ne \\xFF | dd conv=notrunc bs=1 count=1 of=/data/tee/2
```

Requesting process list with remote attestation client program:

```
$ ./remote-attest-client.py --host rpi pslist
status = StatusCode.CANCELLED
```

The serial output of the TA on the board:

```
E/TC:0 syscall_storage_obj_open:301 Object corrupt
D/TA:  Key (ra_attester_pubkey) not found in Secure Storage
D/TA:  Failed to init RSA op
D/TA:  Request verification failed!
D/TC:0 tee_ta_invoke_command:625 Error: f0100001 of 4
```

5.4.4 Limitations

The OP-TEE documentation, Porting Guidelines[4], lists the requirements for making a real secure platform. Although, OP-TEE developers aim to create solutions usable on a wide range of devices, on some platform this is not achievable, due to their proprietary nature.

At the time of writing, none of the boards supported by OP-TEE, have publicly available implementation of real Secure Storage. Encryption keys used in OP-TEE are derived from a Hardware Unique Key (HUK), which should be unique for each device. The keys for Secure Storage are derived from the HUK too. However, the device manufacturers do not make the documentation of how to access the HUK publicly available (under a license compatible with the license of OP-TEE[5]), thus in the source code, the functions responsible for providing the HUK are just stubs, and return a hard coded constant. This potentially makes Secure Storage objects readable to attackers (with the right NW user privileges), when these values are not changed. The best solution would be HUKs, that are not even directly accessible from software.

The memory areas of OP-TEE and the TAs should be accessible only from the Secure World. Thus, the separation of secure and non-secure memory areas should be configured properly. For example, this can be done using TrustZone Address Space Controller (TZASC) defined by ARM.[6] The documentation, mentioned at the beginning of this section, describes TZASC as the following:

[4] `https://github.com/OP-TEE/optee_os/blob/master/documentation/porting_guidelines.md` (last visited on 2018–10–22)

[5] `https://github.com/OP-TEE/optee_os/blob/master/LICENSE` (last visited on 2018–10–22)

[6] `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0431b/index.html` (last visited on 2018–10–22)

TZASC can be used to configure DDR memory into separate regions in the physical address space, where each region can have an individual security level setting. After enabling TZASC, it will perform security checks on transactions to memory or peripherals.

Unfortunately, on our platform, the Raspberry Pi 3 B, the before mentioned features are not available.[7] A Hardware Unique Key is not accessible (since the documentation of the Broadcom SoC it uses is not public), and the TZASC is not part of the SoC architecture. Although, the platform is great for testing and developing Trusted Applications, it should not be considered secure enough to be used as a gateway in industrial environments. Thus, making our implementation only a proof of concept, however, the solutions, described in our paper, can be ported to devices which have mature security capabilities.

5.5 Secure Communication

5.5.1 Basic functioning

In our paper, we have extended the SKS OP-TEE proposal, making it possible to use it with an OpenSSH client to authenticate and log in to an ssh server. This can be achieved for example by using the commands mentioned through Section 4.6:

```
# pkcs11-tool --module /usr/lib/libsks.so --init-token  \
              --label testtoken --so-pin 12341234
Using slot 0 with a present token (0x0)
Token successfully initialized

# pkcs11-tool --module /usr/lib/libsks.so --label testtoken  \
              --login --so-pin 12341234 --init-pin --pin 12341234
Using slot 0 with a present token (0x0)
User PIN successfully initialized

# pkcs11-tool --module /usr/lib/libsks.so --label testtoken  \
              --login --pin 12341234 --keypairgen            \
              --label testkey --key-type rsa:2048
Using slot 0 with a present token (0x0)
Key pair generated:
Private Key Object; RSA
    label: testkey
    ID: 01bebf99
    Usage: decrypt, sign, unwrap
Public Key Object; RSA 2048 bits
    label: testkey
    ID: 01bebf99
    Usage: encrypt, verify, wrap

# ssh-keygen -D /usr/lib/libsks.so -e > .ssh/authorized_keys
# chmod 0600 .ssh/authorized_keys

# ssh root@localhost -I /usr/lib/libsks.so
The authenticity of host 'localhost (127.0.0.1)' can't be established.
```

[7] https://connect.linaro.org/resources/las16/las16-111/ (last visited on 2018–10–22)

```
ECDSA key fingerprint is SHA256:KETZVgw2Otax58AKVxy2pWHB/w8mvsywB2EG9nO3iGM.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'localhost' (ECDSA) to the list of known hosts.
Enter PIN for 'testtoken':
# ls /
bin etc lib32 mnt root sys var
data init linuxrc opt run tmp
dev lib media proc sbin usr
# exit
Connection to localhost closed.
#
```

Listing 5.1. *Using SKS with OpenSSH*

Alternatively, for the public key extraction pkcs11-tool could also be used. We have also tested this using the ssh-agent to handle the token:

```
# ssh-add -s /usr/lib/libsks.so
Enter passphrase for PKCS#11:
Card added: /usr/lib/libsks.so

# ssh root@localhost
# exit
Connection to localhost closed.
```

Listing 5.2. *Using SKS with ssh-agent*

5.5.2 Performance

As usual, adding security features creates an overhead and hinders performance. In the case of SKS, the overhead is the context change between Normal World and Secure World and the serialization and deserialization required to pass the parameters. However, because of the nature of the ssh protocol, this overhead is limited to the key generation and the authentication process [46], since after authentication temporal keys are used for communication, so it will not affect overall system performance that much. Testing on QEMU, the time required for authentication was not noticably different: connecting without SKS took around 1.6 seconds while it took 3 seconds with using SKS, but it's hard to take an exact measurement, since the program requires user input. On the other hand, the key generation takes significantly longer with SKS: it takes around 5 seconds to generate a 2048 bit RSA key with `ssh-keygen`, it takes as long as 53 seconds to do the same through the pkcs11-tool with SKS. This could possibly be because SKS is generating its own randomness, but we have not done any investigation to find out what takes so long during key generation. The measurements were taken with the `time` linux utility.

5.5.3 Security

The security of SKS depends on TrustZone and OP-TEE working correctly, and having correct configurations for the device that is being used. If all these are right, the secret key

can never be extracted from the SKS soft token. So even if an adversary takes complete control over Linux, she can only use it for certain cryptographic operations, but can not recover it. Of course, the PKCS#11 standard is fairly big, and as mentioned in **??**, there have been vulnerabilities found in it before. So, to further limit the attack surface, one could evaluate what PKCS#11 functions the client applications used in their system need, and disable the rest in the SKS TA.

5.5.4 Limitations

SKS is still heavily under development, and not yet merged into the mainline OP-TEE repository, so it has to be added by hand if we want to use it. Another limitation is, that we only tested it with OpenSSH and pkcs11-tool, so if someone would want to use a different application that has PKCS#11 support, she would likely discover some issues, just like we did. Another limitation, that is a result of using OpenSSH, since in our implementation, only the private key of the client is protected by SKS, but the fingerprint of the ssh server is stored in Normal World, so if an adversary were to take over the REE side of the system, it could modify it and impersonate the ssh server.

Chapter 6

Conclusion

In this paper, our aim was to create a highly secured IoT gateway considering the possibility that it may even be physically compromised, as a result of its location being likely unsecurable. Low price was also an objective, in order to raise the likeliness of adoption, so security co-processors were ruled out. Assuming some kind of physical tamper protection and building on known hardening techniques, U-Boot's fit image verification, TrustZone technology, OP-TEE and the SKS proposal we implemented and demonstrated a proof of concept system with numerous mechanisms for protection.

The discussed secure boot process builds up a chain of trust, and the solutions applied in it, like U-Boot's FIT image verification [16, 15] and dm-verity [12], are based on cryptographic algorithms, and together ensure that after any reset, the device boots into a known and secure state.

A secure firmware update was achieved by building on the secure boot process as a base with added versioning, and by extending that process with fault detection and fallback mechanisms, which made it also fail-safe, and by developing managing mechanisms, which support both aspects.

Based on the recommended settings page [35] of the Kernel Self Protection Project [34], and picking the right options, a security hardened operating system was achieved, while still supporting the desired features. A security hardened operating system has a reduced attack surface, and is more resistant to typical known attacks.

With integrity monitoring implemented in a Trusted Execution Environment, the detection algorithms and their results can be trusted, and possible run-time compromises, failures, misconfigurations and break-in attempts can be detected. The resulting device state consists of the extracted process list and the calculated cryptographic hash digests of their code in memory. Using these, we can detect malicious software, whether it be new installations or modified existing system components. We also presented a basic provisioning technique for storing asymmetric keys in a TEE based Secure Storage.

For securing communications, we added support to the OP-TEE SKS proposal for OpenSSH, in order to be able to create secure connections, with the secret key stored in SKS and

accessed only through the PKCS#11 API. Thus, the secret key is as well protected as if it was on a hardware token, without the extra cost associated with it.

As future work an implementation utilizing a hardware root of trust on a platform, that has one, could be done. Which would be a real, working secure boot process. Support and usage for more secure algorithms in U-Boot would improve the security of the boot process.

An implementation using a hardware root of trust could derive a physical write protection for the logs besides the benefits for the secure boot process. Also implementations along the other use cases, described in the implementation, could be done in the future.

The security hardened system could be tested with definitive use cases/applications, in order to find out whether they are compatible with the system, or the setting have to be altered.

There are many more interesting integrity monitoring techniques that we can implement in the Trusted Application. First, the detection of rootkits— both in the Linux kernel and user mode—could be a great feature to discover persistent malicious software. There are existing memory forensics concepts and solutions to detect shared library and shellcode injections, process hollowing, Global Offset Table overwrites, and other techniques used by attackers to gain a persistent foothold on a system [27]. We could try to utilize and implement these in the Trusted Execution Environment. Second, by monitoring network connections, suspicious network activity could be discovered. Third, the process hash calculation in the Trusted Application is only performed for the code segment of the processes, but the same principle could be applied to other binary segments whose hash values must not change during run time or at rest (depending on the type and purpose of the programs). Fourth, to increase the security of the Pseudo Trusted Application, we could also implement the access control ideas, discussed in section 4.5.2. Finally, for more flexibility, the PTA and TA interfaces could be extended to enable the specification of the hash function used for digest calculations.

As already mentioned, SKS still needs much more development. As a continuation of the approach in this paper, it would be interesting, to ensure compatibility with a VPN solution, since that is a functionality that an IoT gateway would be likely to need. Another aspect that requires further work, is the contribution of this work into mainline OP-TEE. The OP-TEE API extension in Section 4.6.7 is already being reviewed on GitHub as a pull request[1], and after that is done, it would be nice to help SKS get to the point where it will be accepted into the mainline OP-TEE repository as well.

[1] `https://github.com/OP-TEE/optee_os/pull/2524` (last visited 2018. 10. 22.)

Acknowledgment

We would first like to thank our thesis advisor Dr. Levente Buttyán of the CrySyS laboratory at the Budapest University of Technology and Economics. Dr. Buttyán provided us with valuable advice, guidance and encouragement. We would also like to thank Etienne Carriere for allowing us to use the SKS proposal and for reviewing our modifications to it.

Bibliography

[1] Arm security technology – building a secure system using trustzone technology. Technical Report PRD29-GENC-009492C, 2009.

[2] Manos Antonakakis, Tim April, Michael Bailey, Matthew Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J. Alex Halderman, Luca Invernizzi, Michalis Kallitsis, Deepak Kumar, Chaz Lever, Zane Ma, Joshua Mason, Damian Menscher, Chad Seaman, Nick Sullivan, Kurt Thomas, and Yi Zhou. Understanding the mirai botnet. In *Proceedings of the 26th USENIX Conference on Security Symposium*, SEC'17, pages 1093–1110, Berkeley, CA, USA, 2017. USENIX Association.

[3] Ahmed M Azab, Peng Ning, Emre C Sezer, and Xiaolan Zhang. Hima: A hypervisor-based integrity measurement agent. In *Computer Security Applications Conference, 2009. ACSAC'09. Annual*, pages 461–470. IEEE, 2009.

[4] Ahmed M. Azab, Peng Ning, Jitesh Shah, Quan Chen, Rohan Bhutkar, Guruprasad Ganesh, Jia Ma, and Wenbo Shen. Hypervision across worlds: Real-time kernel protection from the arm trustzone secure world. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 90–102, New York, NY, USA, 2014. ACM.

[5] Ahmed M Azab, Peng Ning, Zhi Wang, Xuxian Jiang, Xiaolan Zhang, and Nathan C Skalsky. Hypersentry: enabling stealthy in-context measurement of hypervisor integrity. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 38–49. ACM, 2010.

[6] Akshay Bhat. Secure Boot and Encrypted Data Storage. `https://www.timesys.com/security/secure-boot-encrypted-data-storage/`. Accessed October 28, 2018.

[7] Akshay Bhat. Software / Firmware Update Design Considerations. `https://www.timesys.com/security/software-firmware-update-design-considerations/`. Accessed October 28, 2018.

[8] Christian Cachin and Nishanth Chandran. A secure cryptographic token interface. In *Proceedings of the 2009 22Nd IEEE Computer Security Foundations Symposium*, CSF '09, pages 141–153, Washington, DC, USA, 2009. IEEE Computer Society.

[9] George Coker, Joshua Guttman, Peter Loscocco, Amy L. Herzog, Jonathan Millen, Brian O'Hanlon, John Ramsdell, Ariel Segall, Justin Sheehy, and Brian T. Sniffen. Principles of remote attestation. *Int. J. Inf. Sec.*, 10:63–81, 06 2011.

[10] Tim Cooijmans, Joeri de Ruiter, and Erik Poll. Analysis of secure key storage solutions on android. In *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices*, SPSM '14, pages 11–20, New York, NY, USA, 2014. ACM.

[11] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers, 3rd Edition.* O'Reilly Media, Inc., 2005.

[12] cryptsetup. dm-verity. `https://gitlab.com/cryptsetup/cryptsetup/wikis/DMVerity`. Accessed October 28, 2018.

[13] A Micro Devices. Amd64 architecture programmer's manual volume 2: System programming, 2006.

[14] DENX Software Engineering. Das U-Boot – the Universal Boot Loader. `https://www.denx.de/wiki/U-Boot/WebHome`. Accessed October 28, 2018.

[15] DENX Software Engineering. U-Boot FIT Signature Verification. `https://git.denx.de/?p=u-boot.git;a=blob;f=doc/uImage.FIT/signature.txt`. Accessed October 28, 2018.

[16] DENX Software Engineering. U-Boot Verified Boot. `https://git.denx.de/?p=u-boot.git;a=blob;f=doc/uImage.FIT/verified-boot.txt`. Accessed October 28, 2018.

[17] A. Francillon, Q. Nguyen, K. B. Rasmussen, and G. Tsudik. A minimalist approach to remote attestation. In *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1–6, March 2014.

[18] Xinyang Ge, Hayawardh Vijayakumar, and Trent Jaeger. Sprobes: Enforcing kernel code integrity on the trustzone architecture. *arXiv preprint arXiv:1410.7747*, 2014.

[19] Simon Glass. Verified U-Boot. `https://lwn.net/Articles/571031/`. Accessed October 28, 2018.

[20] Simon Glass. Verified boot in chrome os and how to make it work for you. In *Embedded Linux Conference Europe*, 2013.

[21] GlobalPlatform. *TEE Client API Specification*, 2010. Version v1.0.

[22] GlobalPlatform. *TEE Internal Core API Specification*, 2016. Version v1.1.2.

[23] James Greene. Intel trusted execution technology. *Intel Technology White Paper*, 2012.

[24] J. Jonsson and B. Kaliski. Public-key cryptography standards (pkcs) #1: Rsa cryptography specifications version 2.1, 2003.

[25] Georgios Kambourakis, Constantinos Kolias, and Angelos Stavrou. The mirai botnet and the iot zombie armies. In *Military Communications Conference (MILCOM), MILCOM 2017-2017 IEEE*, pages 267–272. IEEE, 2017.

[26] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. Comprehensive formal verification of an OS microkernel. *ACM Transactions on Computer Systems*, 32(1):2:1–2:70, February 2014.

[27] Michael Hale Ligh, Andrew Case, Jamie Levy, and Aaron Walters. *The Art of Memory Forensics: Detecting Malware and Threats in Windows, Linux, and Mac Memory.* Wiley Publishing, 1st edition, 2014.

[28] Robert Love. *Linux Kernel Development.* Addison-Wesley Professional, 3rd edition, 2010.

[29] Hyungon Moon, Hojoon Lee, Jihoon Lee, Kihwan Kim, Yunheung Paek, and Brent Byunghoon Kang. Vigilare: Toward snoop-based kernel integrity monitor. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 28–37, New York, NY, USA, 2012. ACM.

[30] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), April 1965.

[31] OASIS. *PKCS #11 Cryptographic Token Interface Base Specification*, 2016. Version 2.40 Plus Errata 01.

[32] Siani Pearson. *Trusted Computing Platforms: TCPA Technology in Context.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 2002.

[33] Nick L. Petroni, Jr., 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, Berkeley, CA, USA, 2004. USENIX Association.

[34] Kernel Self Protection Project. Kernel Self Protection Project. `https://kernsec.org/wiki/index.php/Kernel_Self_Protection_Project`. Accessed October 28, 2018.

[35] Kernel Self Protection Project. Kernel Self Protection Project/Recommended Settings. `https://kernsec.org/wiki/index.php/Kernel_Self_Protection_Project/Recommended_Settings`. Accessed October 28, 2018.

[36] The Chromium Projects. Chromium OS File System/Autoupdate. `https://www.chromium.org/chromium-os/chromiumos-design-docs/filesystem-autoupdate`. Accessed October 28, 2018.

[37] The Chromium Projects. Chromium OS Firmware Boot and Recovery. https://www.chromium.org/chromium-os/chromiumos-design-docs/firmware-boot-and-recovery. Accessed October 28, 2018.

[38] The Chromium Projects. Chromium OS Firmware Verified Boot Crypto Specification. https://www.chromium.org/chromium-os/chromiumos-design-docs/verified-boot-crypto. Accessed October 28, 2018.

[39] The Chromium Projects. Chromium OS Verified Boot. https://www.chromium.org/chromium-os/chromiumos-design-docs/verified-boot. Accessed October 28, 2018.

[40] The Chromium Projects. Chromium OS Verified Boot Data Structures. https://www.chromium.org/chromium-os/chromiumos-design-docs/verified-boot-data-structures. Accessed October 28, 2018.

[41] Mohamed Sabt, Mohammed Achemlal, and Abdelmadjid Bouabdallah. Trusted Execution Environment: What It is, and What It is Not. In *14th IEEE International Conference on Trust, Security and Privacy in Computing and Communications*, Helsinki, Finland, August 2015.

[42] Peter Jay Salzman, Michael Burian, and Ori Pomerantz. *The Linux Kernel Module Programming Guide*. iUniverse, Incorporated, 2001.

[43] Sean W. Smith. Outbound authentication for programmable secure coprocessors. *International Journal of Information Security*, 3(1):28–41, Oct 2004.

[44] Hailun Tan, Wen Hu, and Sanjay Jha. A tpm-enabled remote attestation protocol (trap) in wireless sensor networks. In *Proceedings of the 6th ACM Workshop on Performance Monitoring and Measurement of Heterogeneous Wireless and Wired Networks*, PM2HW2N '11, pages 9–16, New York, NY, USA, 2011. ACM.

[45] Jiang Wang, Angelos Stavrou, and Anup Ghosh. Hypercheck: A hardware-assisted integrity monitor. In *International Workshop on Recent Advances in Intrusion Detection*, pages 158–177. Springer, 2010.

[46] T. Ylonen and Ed. C. Lonvick. The secure shell (ssh) authentication protocol, 2006.