KU LEUVEN

FACULTY OF
ENGINEERING SCIENCE

Software integrity checks on open platforms

Performing process attestation on user-controlled ARM TrustZone devices

Oberon Swings

Thesis voorgedragen tot het behalen
van de graad van Master of Science
in de ingenieurswetenschappen:
computerwetenschappen, hoofdoptie
Veilige software

Promotoren:

Prof. dr. ir. F. Piessens
Dr. J.T. Mühlberg

Evaluatoren:

Prof. dr. D. Hughes
Ir. P. Totis

Begeleiders:

Ir. S. Pouyanrad
Ir. F. Alder

Academiejaar 2021 – 2022

Preface

First and foremost I would like to thank my co-promotor Jan-Tobias Mühlberg, and my daily mentors Fritz Alder and Sepideh Pouyanrad. The countless hours they were present to guide me, discuss possible solutions or give feedback are immensely appreciated. I really enjoyed working on my thesis but this could have been very different if my mentors were not as enthusiastic as they were. They gave me a lot of motivation and inspiration during our weekly meetings which kept me going the entire year.

I also want to thank my promotor prof. Frank Piessens. My interest in software security was elevated to a new level during one of his courses I attended. This was the stepping stone to choose one of his thesis topics. During my research endeavor he gave some very interesting and valuable pointers towards my progress which had a strong positive influence on the end result.

Last but not least I am very grateful towards my family, friends and especially my life partner to whom I kept on talking about my research topic and its progress. It was really nice to have so many people listen to my day to day struggles and accomplishments. I felt very appreciated when they tried to understand what I was doing and gave me the feeling it was interesting.

Oberon Swings

Contents

Abstract

Smartphone devices are used more and more for tasks that rely on sensitive data, online banking, e-health and so on. While this is a natural evolution with respect to functionality, the security features of a smartphone are not as extensive as those of a personal computer. Many smartphone devices have an ARM System on Chip, equipped with ARM TrustZone by which the manufacturer attempts to increase the security of these devices. ARM TrustZone is a hardware security solution which provides a Trusted Execution Environment. With this capability, features like secure memory, trusted Input and Output, and process execution isolation are available.

Smartphone manufacturers like Samsung utilize this ARM TrustZone framework to build up a security solution like Samsung KNOX. The downside of these solutions is that the manufacturer stays in control of the smartphone even after it has been sold. They decide which software is allowed to run on the device and which is not. To return the control and ownership to the users, the PinePhone has been introduced, an open smartphone with ARM TrustZone features. To access these Trusted Execution Environment functionalities a kernel is required. For this, there are also open source solutions like OP-TEE. The tools to obtain a secure open smartphone device exist, but they need to be put together along with security implementations to become a complete product.

In this work, a crucial part of Remote Attestation has been looked at, namely measuring the integrity of applications running in the Normal World of the ARM TrustZone framework. Lots of research has been done to isolate applications in the Secure World from the rich Operating System. Also securing the data storage or the Input and Output channels related to these applications are common practice and well understood. Of course, the security of these applications is of utmost importance but this Secure World could also increase the security guarantees for the Normal World. One way of doing this is by allowing the Secure World to attest processes in the Normal World.

Samenvatting

Smartphones worden steeds vaker gebruikt om taken uit te voeren die gebruik maken van sensitieve gegevens zoals online bankieren of het raadplegen van gezondheidsrapporten. Alhoewel dat een natuurlijke evolutie is van de functionaliteit, moet men zich ervan vergewissen dat de veiligheid van een smartphone niet zo uitgebreid is als die van een persoonlijke computer. Vele smartphones hebben een processor van ARM die uitgerust is met TrustZone-mogelijkheden. ARM TrustZone is een veiligheidsmechanisme toegepast in de hardware dat een vertrouwde uitvoeringsomgeving tot stand brengt. Deze omgeving zorgt voor functionaliteiten zoals veilig geheugen, vertrouwde in- en uitvoer en het isoleren van de uitvoering van processen.

Smartphoneproducenten zoals Samsung gebruiken de ARM TrustZone-technologie om een veiligheidsoplossing zoals Samsung KNOX te maken. De keerzijde hiervan is dat de producenten de controle over de smartphone behouden zelfs nadat die verkocht is. Zij beslissen welke software mag draaien op het apparaat en welke niet. Om de controle en het eigenaarschap terug te geven aan de gebruiker is de PinePhone geïntroduceerd. Dat is een open smartphone met ARM TrustZone-mogelijkheden. Om toegang te hebben tot deze functionaliteiten is er nood aan een besturingssysteem. OP-TEE is een voorbeeld van zo'n besturingssysteem waarvan de code vrij beschikbaar is gemaakt. De benodigdheden om een veilige en open smartphone te verkrijgen bestaan. Deze moeten nog samengebracht worden met implementaties van beveiligende software om een compleet product te verkrijgen.

In dit werk is er vooral aandacht besteed aan het meten van uitvoerende processen in de normale wereld van de ARM TrustZone-technologie. Er is al veel onderzoek gedaan naar het isoleren van applicaties in de veilige wereld om ze te beschermen tegen de besturingssystemen van de normale wereld. Daarnaast wordt het veilig opslaan van gegevens of vertrouwelijk maken van de in- en uitvoerkanalen geassocieerd met de applicatie ook zeer vaak toegepast. Natuurlijk is de veiligheid van de applicaties in de veilige wereld zeer belangrijk maar deze veilige wereld kan ook de veiligheidsgaranties van de normale wereld verbeteren. Één manier om dat aan te pakken is door de integriteit van de processen in de normale wereld te testen.

List of Figures

List of Abbreviations

Abbreviations

AES	Advanced Encryption Standard
API	Application Programming Interface
AST	Abstract Syntax Tree
BKEK	Blob Key Encryption Key
CAAM	Cryptographic Acceleration and Assurance Module
CC	Common Criteria
CCC	Confidential Computing Consortium
CoT	Chain of Trust
CPU	Central Processing Unit
DoS	Denial of Service
DMA	Direct Memory Access
DRAM	Dynamic Random Access Memory
DRK	Device Root Key
DSK	Device Sealing Key
ECC	Elliptic Curve Cryptography
GIC	Generic Interrupt Controller
HMAC	Hash-based Message Authentication Code
IMA	Integrity Measurement Architecture
IoT	Internet of Things
I/O	Input and Output
JVM	Java Virtual Machine
LED	Light Emitting Diode
MAC	Message Authentication Code
MBA	Model-based Behavior Attestation
MK	Master Key
NW	Normal World
OCM	On Chip Memory
OCROM	On Chip Read Only Memory
OP-TEE	Open Portable Trusted Execution Environment

OS	Operating System
PDG	Program Dependency Graph
PID	Process IDentifier
PTA	Pseudo Trusted Application
PUF	Physically Uncloneable Function
RA	Remote Attestation
RAM	Random Access Memory
REE	Rich Execution Environment
ROM	Read Only Memory
RoT	Root of Trust
RSA	Rivest-Shamir-Adleman
SCR	Secure Configuration Register
SCU	Secure Cryptographic Unit
SGX	Secure Guard eXtension
SMC	Secure Monitor Call
SML	Secure Measurement Log
SNVS	Secure Non-Volatile Memory
SoC	System on Chip
SRAM	Static Random Access Memory
SW	Secure World
TA	Trusted Application
TCB	Trusted Computing Base
TCG	Trusted Computing Group
TEE	Trusted Execution Environment
TFA	Trusted Firmware-A
TLS	Transport Layer Security
TOC-TOU	Time Of Check to Time Of Use
TPM	Trusted Platform Module
TZASC	TrustZone Address Space Controller
TZMA	TrustZone Memory Adapter
TZPC	TrustZone Protection Controller
VM	Virtual Machine

Chapter 1

Introduction

Smartphones have become an essential part of our daily lives. This can be seen when comparing the number of smartphone users [2] and the worldwide population [3], because in 2020 80% of the entire population owned a smartphone device. These devices can be spotted everywhere: people use them at home, on the bus or even at work. In most cases, these phones are only occasionally used for text messaging or calling but very often for reading e-mails, surfing on the web or even for services like e-banking. Because of this wide range of functionalities, some people may even replace their personal computer with a smartphone entirely. The success of smartphones lies in their ease of use and always being accessible: people just carry it in their pockets. Besides the user having access to his phone all the time, a smartphone also has or can have access to the internet all the time. The internet is the gate to lots of services which are often accessed through mobile devices [4], in 2021 over 90% of internet users accessed the internet through their smartphone.

Devices making lots of connections on the go while also utilizing online services, for which sensitive data is required may introduce security vulnerabilities. The fact that people are using their smartphones for services like online banking, or even consulting health-related reports implies that some sensitive data must be stored on these devices, or at least be present while they are interacting with it. While this data should be protected very well, it is present on an Internet of Things (IoT) device for which lots of security challenges exist [5]. The hardware similarities between smartphones and IoT devices are more prominent than one might expect. Because the architecture of smartphones is often built around the same or very similar System on Chip (SoC) as IoT devices. The main issue here is that IoT devices are designed for performance: they only have a small number of tasks but these need to be executed as fast or as energy efficient as possible. This also applies to smartphones because while the functionality of a smartphone is close to that of a personal computer, the hardware is not. This weak link in the hardware gives rise to multiple different attack strategies that adversaries can utilize to steal sensitive data from smartphone users.

Recently improvements have been made in the area of IoT and smartphone hardware by extending the processors of these devices with features that make it possible to set up a Trusted Execution Environment (TEE). A TEE can increase the security of an IoT device, which is often achieved by utilizing core security services for critical operations. Examples of these critical operations are cryptographic operations, storing data in secure memory, or accessing Input and Output (I/O) through trusted paths. The majority of smartphones use SoC designed by ARM, very often equipped with ARM TrustZone capabilities. Large smartphone manufacturers like Samsung utilize these features to build a security solution on them, Samsung KNOX [6] for instance. The downside to these solutions is the disadvantage that the manufacturer stays in control of the smartphone even after it has been sold. He decides which software is allowed to run on the device and which is not. Recently the PinePhone [7] was introduced, which is an open smartphone platform that also has an ARM SoC with TrustZone enabled. On top of the openness of the platform itself, Linux is the rich Operating System (OS) that runs on this device and the recommended kernel for the Secure World (SW) is OP-TEE [8]. These are both large open source projects with an active community.

Contributions

We attempt to increase the security of a smartphone without closing off the system with respect to the user or third-party software providers. This means that the user should be in control and be able to decide what software can run on his device and what not, instead of the manufacturer of the device. This is achieved by taking integrity measurements of the processes running in the user space and rich OS environment. The measuring process runs inside the TEE to provide strong security guarantees for its execution. This method is based on existing work [1], which focuses on IoT nodes and the performance of the attestation. The focus in this thesis is shifted towards making the solution work on a PinePhone while respecting the openness of the device. To allow for a direct comparison between the performance achieved in the paper and our implementation, some experiments in the paper are redone. To allow others to easily reproduce or review the work that has been done, all code is made available in open source at `https://github.com/SwingsOberon/master_thesis`. Of course, the performance is only a small aspect of the analysis of the solution. To make the security analysis, this solution is compared to similar work. This comparison discusses what solution is the most effective, and what solution achieves the most in terms of defended attacks while keeping in mind the reality and feasibility of the assumptions that are made. Finally, it is evaluated which type of solution is the most promising as the direction for future work based on the comparison with similar alternatives.

Based on the performed work, executed experiments and discussed comparisons the following questions are attempted to be answered:

- What can be used as a Root of Trust (RoT) in such an open system?

- Can attestation be used to increase the security of an open platform without disrupting the openness of the system?
- Does this security solution meet the expectations of all smartphone stakeholders?
- How does this type of security solution compare with the ones in closed systems?

Outline

In the next chapter, more background information about, among other things, Remote Attestation (RA), and ARM TrustZone is given. In the third chapter, the methods to solve the problem are explained. In chapters 4 and 5, the implementation of the integrity measurement program is elaborated upon and the outcome of the experiments are made clear respectively. The sixth chapter includes a discussion about this thesis informing the reader about related work and future work. The final chapter concludes the presented work.

Chapter 2

Background

2.1 Remote Attestation

RA is a method to prove certain properties of a system to a remote verifier. This is achieved by collecting evidence on the attested system and combining it into a report. Based on this the remote verifier should be able to draw conclusions about the desired properties. An example of such property is the integrity of the execution control flow of a process, which could also be generalized to the correct behavior of a process. It is a very difficult task to provide sufficient evidence to allow the remote verifier to prove this property. In this case for instance it is not sufficient to only check the integrity of the executable code of a process, because the process can behave differently from what is expected due to its data structures being tampered with.

The prover (in other words the system that is attested by the verifier) requires certain abilities to correctly and securely gather evidence about itself. First of all, a trusted base enforcing an isolation mechanism is needed to avoid the evidence collecting process is tampered with. This isolation will make sure that the attestation can be executed, even when the target is unreliable due to the presence of malicious code or a compromised rich OS. Another important characteristic of the prover is the ability to gather evidence about desired properties of the system. This is often achieved by measuring certain aspects of the system, for instance, hashing a code page to allow the hash digest to be compared with the original to prove the integrity of that code page. The measurements need to be structured to allow the verifier to draw conclusions from them. Finally, the prover also needs to provide some sort of proof that the attestation report is trustworthy because the verifier relies on the authenticity of the information it receives.

The verifier requests an attestation report from the prover when he desires to check the trustworthiness of the device. This report ideally consists of fresh and securely obtained information from the prover. The freshness is important because the evidence can in the best case only prove the trustworthiness of the device at the

time the measurements were taken. Securely acquiring these measurements is also important to make sure that an adversary is not able to forge a false report to make it look like the device can be trusted while it has been compromised. Based on this information the verifier should be able to identify the state in which the prover is currently situated. The perceived state of the device can then be investigated by the verifier to determine whether the state is secure, whether it is expected or known and, finally, based on these findings, whether the verifier trusts the state of the prover and therefore the prover itself. These decisions are referred to as attestations. It is not unusual for a verifier to keep track of past attestations of a prover and use this history for more elaborate investigations.

Exemplary techniques

Integrity Measurement Architecture (IMA) [9] implements attestation by measuring the code of programs before running them on the target. The prover is responsible for measuring the executable code pages of a process, for instance, by hashing them when the program is loaded into Random Access Memory (RAM) for execution. After all hash digests have been obtained, they need to be sent to the verifier which is required to have access to a database containing a mapping between the executable code pages of all the possible programs that can run on the prover, and their hash digest. The verifier can check whether the program's code has been modified, by comparing the newly calculated digests with the ones in its database. There is not much recent work that specifically mentions IMA, as it is often combined with other techniques or used in very specific use cases. In one paper [10], for instance, Intel Secure Guard eXtension (SGX) is used to assess code integrity (along with other aspects) of a running Virtual Machine (VM). A very similar paper [11] describes the method to measure the code integrity (and dynamic characteristics) in a TEE provided by ARM TrustZone. According to [12], IMA is inflexible and static. On top of that, it is perceived as a very limited solution, [13] claims similar shortcomings of the reflection method [14] which is very similar to IMA. First of all, it is inflexible or static because, if new software is introduced or software is updated, the verifier needs to update its mapping because the hash digests will certainly change. This is rather complex to manage because not all provers update at the same time. Some measurements will need to be compared to the old value, and some to the new one. Furthermore, it is not known how long this old value should be kept for attestation purposes. IMA is seen as limited because there are a variety of ways a program can misbehave without the code base being tampered with. One example of such misbehavior without the code base changing is the well-known attacks related to buffer overflows and return-oriented programming.

Attestation on program execution goes beyond merely measuring the integrity of the executable code pages, it measures the dynamic behavior of the process. [15] proposes to observe the system calls the program makes to verify whether it adheres to the permitted control flow. This is achieved by introducing hooks on the system calls to allow the attestation procedure to intercept them. During this interception of

the system call, its parameters and its time of arrival are all analyzed and afterwards stored to provide it to the verifier later. Based on the Abstract Syntax Tree (AST) and the Program Dependency Graph (PDG), the verifier should be able to verify whether a certain system call was possible. If it can correctly identify the state in which the attested program was. Although this method is a big improvement, there are still weaknesses. For example, the granularity of a system call might not give enough details about the behavior of the system. Furthermore, there is more to the behavior than the system calls alone. Many similar solutions that focus on the dynamic behavior of code have been proposed, all with their weaknesses and shortcomings. RIPTE [16] combines dynamic measurement and encrypts return addresses with a Physically Unclonable Function (PUF) key and it can protect software integrity at runtime. [17] uses machine learning algorithms to verify the dynamic behavior of the prover. [18] provides runtime measurement and RA for Java bytecode integrity to prevent the Java Virtual Machine (JVM) from being bypassed.

As mentioned earlier when describing IMA, different techniques are often combined to protect against a wider variety of vulnerabilities. Model-based Behavior Attestation (MBA) was proposed by [19], which is again mentioned by the same researchers in their analysis of existing techniques [12], on which these exemplary techniques are based. The platform is expected to enforce a certain security model, or have its submodules enforce a certain security model. This division helps to precisely identify the behavior of each module, but it also makes sure the high-level behavior policy does not need to be made too general. This behavior policy dictates constraints and conditions which need to be adhered to by the prover. Finally, the enforcement behavior is compared to the expected behavior to check the trustworthiness of the prover. These behavior policies consist of certain properties that the module possesses. It is still very hard to map configurations of the platform to certain properties, but it does provide lots of flexibility in terms of attestation. [20] also combines a variety of approaches to attest trust of IoT devices and implements multiple modules that attest certain platform properties based on different measurements. Some of these properties are OS data structures, available resources, event occurrence and event timing.

2.2 Trusted Execution Environment

Based on the papers of the Trusted Computing Group (TCG) [21] and Confidential Computing Consortium (CCC) [22], a TEE can be defined as the following. A TEE is a standardized isolation environment for SoCs that provides a high level of assurance in terms of data integrity, data confidentiality and code integrity. This isolation is enforced by hardware architecture and the boot sequence which uses a hardware RoT, making it highly robust against software and probing attacks. Code running in the TEE and using protected resources (known as Trusted Applications (TA)) is cryptographically verified prior to execution, leading to high integrity assurance. On

top of this, it is also common for a TEE to provide RA functionalities that prove its trustworthiness for third parties.

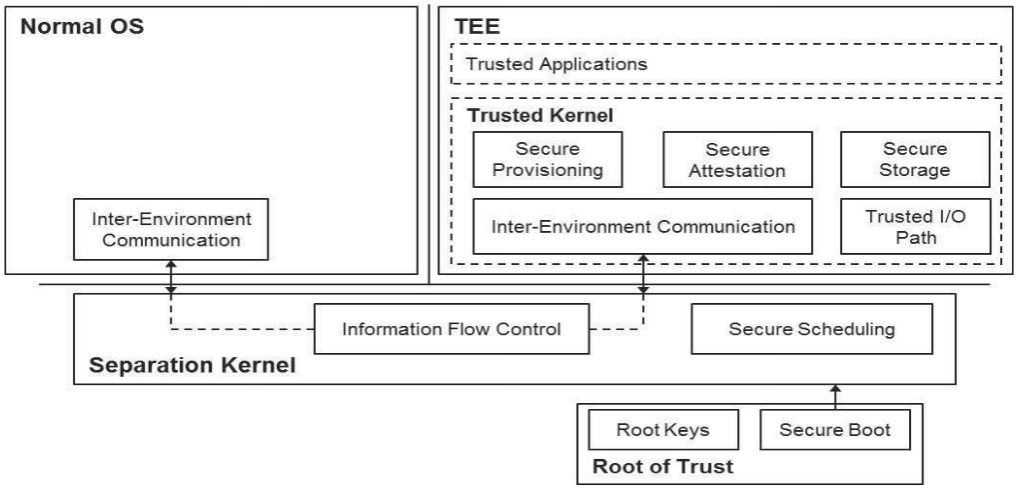

Figure 2.1: Trusted Execution Environment structure (Decomposition)

Source: [23]

In image 2.1 from [23], the different software modules in a common TEE realization are shown. Each of these components has its responsibility and fulfills its own specific duty to guarantee the security of the TEE.

The secure boot makes sure the device starts up in a known state. This is achieved by verifying that the integrity of the loaded code is intact. This booting process often consists of multiple stages where the system is bootstrapped by first running the boot loader, which calls the secondary program loader, which in turn starts the SW and so on. During this process each stage decrypts the code of the next one; when the code is loaded into memory the current stage checks the integrity of the next one by checking a signature or comparing the hash digest. After the current booting phase has finished and the verification of the next phase was successful, the following phase gets control and starts executing. Since the secure boot starts from the RoT and checks the integrity of all intermediate steps before continuing, a Chain of Trust (CoT) is constructed. This CoT ends when the SW or in general the TEE has been given control. Afterwards the Normal World (NW), along with the rich OS, will receive control and here the trust is lost due to the uncertainty of the rich OS containing security vulnerabilities. This RoT is often implemented using some trusted hardware component, for instance, a Secure Cryptographic Unit (SCU) [24] or a Trusted Platform Module (TPM) [25], or finally, using a PUF [26].

The separation kernel has multiple responsibilities, which is logical because it is the coordinator between the Rich Execution Environment (REE) and the TEE. Secure scheduling is the first important aspect to watch over, because a balanced distribution of time slices for each environment needs to be assured. This scheduler is often implemented preemptively to avoid the TEE from affecting the responsiveness of the rich OS. Furthermore, inter-environment communication is very crucial. A strict interface is defined to enable the TEE to communicate with the rest of the system. Communication is indispensable because the TEE cannot provide any added value when it is not able to receive input or produce output. Communication also introduces a lot of threats like message overloading, control data corruption and so on. These dangers can be minimized by implementing a standardized communication mechanism like the GlobalPlatform TEE Client API [27]. These interfaces have been thoroughly tested and worked on by many people and they guarantee reliability in terms of memory and time isolation, providing protection of communication structures and minimum overhead by avoiding unnecessary data copies and context switches. Despite all the effort that has gone into these APIs, flaws can still arise when they are put into practice. In a paper from our University [28], several sanitization vulnerabilities were found, in another paper [29], cache vulnerabilities based on the ARM TrustZone cache architecture are explained, and finally, BOOMERANG [30], which enables NW user programs to abuse the TEE and gain control over the rich OS.

The trusted kernel is responsible for key features that the TEE provides. This kernel is the Trusted Computing Base (TCB) of the TEE, which means that the security relies on the correctness of the code and the absence of security vulnerabilities in it. Some of these key features are secure memory and trusted I/O, which are important to allow the TEE to be used in a secure manner. Secure memory ensures confidentiality, integrity and freshness of stored data. The trusted I/O protects the authenticity and confidentiality of communication between TEE and peripherals. Besides these general security functionalities, the TAs also reside in the TEE. A TA can be developed by a third party that wishes to utilize the secure and trusted functionality of the TEE kernel, or to protect its application from the rich OS, or both. A TA can be called by applications in the REE with a very constrained interface to avoid as many threats as possible. During this execution, the TA has similar privileges to other code running in the TEE. The TEE often enforces multiple levels of privilege, a TA is less privileged than a trusted kernel process, similar to how a user application has fewer privileges than a rich OS process.

Trust

Trust is a core aspect of a TEE, therefore, it is important to establish a notion of what it means to trust something. Trust is a firm belief in the reliability, truth, or ability of someone or something. In computer sciences, there are multiple types of trust with respect to devices. Besides there being differences in types of trust, it is also important to be able to quantify this trust to compare different types and levels

of trust. The Common Criteria (CC) [31] are an international standard that specifies seven evaluation assurance levels, where higher levels envelop all requirements of the preceding levels.

First of all, there is static trust: this type of trust is evaluated only once and assumed to never change during the lifetime of the device. The analysis often takes place before the device is deployed to ensure no malicious code or security vulnerabilities can be introduced by an adversary. For this type of trust, the evaluation assurance levels from the CC are often used. Dynamic trust, on the other hand, is based on the state of the system and this state changes continuously. In this latter case, the trust needs to be measured continuously to have an up to date understanding of it. Generally, a TEE has both types of trust: it needs static trust for the code running inside the TEE (TAs and the trusted kernel), but it also needs static trust when booting up which is provided by the RoT. During execution the TEE of course needs to maintain its level of trust, and this needs to be checked with the methods for measuring dynamic trust.

To perform these measurements, a trusted module or device is required, because if the trust level of an object is measured with an untrusted tool, these measurements can never be fully trusted. With this analogy in mind, it can be derived that a RoT as mentioned earlier is required to start a CoT up to the measuring component. In a CoT the trust from one intermediate component is transferred to the next, because this following component does not have any inherent trust. Components are not inherently trusted because if the environment in which they are started is malicious, it is very hard if not impossible to guarantee the reliability or security of this component. Transferring trust from one component to the next is done by each component checking the next one on authenticity, integrity and reliability, and based on this evaluation, the next component is given control to do its task.

2.3 ARM TrustZone

ARM TrustZone [32] implements the TEE on the processor level, which means it runs below the hypervisor or OS [33]. This approach divides the system into two main partitions, namely the NW and the SW. The processor executes either in the NW or in the SW, as these environments are completely isolated in terms of hardware. The SW is more privileged to make sure a maliciously behaving NW can not damage the SW. The partition between these worlds gives rise to new and better security solutions for applications running on these types of SoCs. An example of this is a reliable on/off switch of device peripherals [34]. Fidelius [35] protects sensitive user input from a malicious browser, MIPE [36] protects sensitive data and code in the SW and is formally proven by the B method, and finally, Komodo [37] attempts to provide similar security guarantees as Intel SGX on ARM TrustZone.

While providing additional security, ARM TrustZone only has a minimal impact on performance due to it being implemented on hardware [38] [39]. ARM TrustZone makes sure that there is a TEE available with capabilities like secure memory, trusted I/O and many others. The trusted kernel is responsible for making partitions of the memory only accessible to the SW, which should then be able to provide secure data storage services to NW applications. Secure data storage is necessary to make sure that data from one application cannot be read or modified by another one. Trusted I/O paths, on the other hand, allow the user application to request I/O features from the SW instead of the rich OS. Because these connections go through the SW, the rich OS is not able to inspect or modify the data that is transmitted to the I/O peripheral, which is important in cases where an adversary has gained control over the rich OS.

Implementation

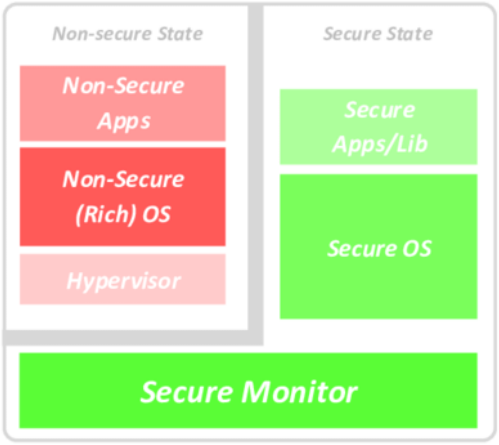

Figure 2.2: ARM TrustZone architecture (Decomposition)

Source: [33]

The NW and the SW are the two main environments in which the processor will be executing code. In image 2.2, these environments are called non-secure state and secure state respectively. The NW shelters the rich OS (like Linux). It is mainly due to the size of these operating systems that they cannot be trusted to run in the SW. The risk of including implementation bugs that introduce security risks, is too high. Also, user-level applications run in the NW, for access to peripherals they rely on the rich OS and, depending on the service, the rich OS relies on the SW. Some services can also be requested from the user application directly to the SW. The SW is where the trusted kernel runs. The implementation of this component is kept very minimal and needs to be designed and implemented very securely to avoid vulnerabilities.

The secure monitor, which can be seen at the bottom of figure 2.2, is responsible for keeping these two execution environments isolated from each other. The NS-bit is an additional bit that flows through the entire pipeline and can be read from the Secure Configuration Register (SCR) to identify the world in which the operation is being executed. The processor has a third state, which is the monitor state. This is necessary to preserve and sanitize the processor state when making transitions between NW and SW states. The new privileged instruction Secure Monitor Call (SMC) allows both worlds to request a world switch, the monitor state will make sure this is handled correctly. The only other way of getting into the monitor state is with exceptions or interrupts from the SW.

This secure monitor of course needs to have an overview of which environment is allowed to access certain memory regions, this is nicely described in [33]. For this, the TrustZone Address Space Controller (TZASC) is introduced. The TZASC can be used to configure specific memory regions as secure or non-secure, such that applications running in the SW can access memory regions associated with the NW, but not the other way around. Making these partitions is also performed by the TZASC, which is made available through a programming interface only available from within the SW. A similar approach is taken for off-chip Read-Only Memory (ROM) and Static Random Access Memory (SRAM), which is implemented using the TrustZone Memory Adapter (TZMA). Whether these components are available or not and how finely grained the memory can be partitioned, depends on the SoC, because they are optional and configurable. Besides the memory, there are also the peripherals which need to be monitored, and depending on the execution environment, access needs to be denied. The TrustZone Protection Controller (TZPC) is in the first place used to restrict certain peripherals from worlds, for instance, to only allow the SW to access them. It also extends the Generic Interrupt Controller (GIC) with support for prioritized interrupts from secure and non-secure sources. This prioritization is important to avoid Denial of Service (DoS) attacks on the SW.

2.4 PinePhone

Many smartphone manufacturers use ARM SoC for their devices. Thanks to the introduction of ARM TrustZone, these companies are extending their security solutions with these TEE capabilities. An example of this is Samsung KNOX [6], but large companies like Google are also able to increase their security guarantees using these functionalities, for instance, Android KeyStore [40]. The drawback of this is that the security solutions of these major companies are not made public. What is even worse, is that these devices cannot be used to experiment with the TEE. On the one hand, this is logical because additional software running in the SW could compromise the security of the entire device, but on the other hand, having these large companies decide which software may run on the device and which cannot, brings about a very closed and exclusive community. To provide the academic world and hardware security enthusiasts with an open platform on which

these ARM TrustZone features are available, PinePhone developed a smartphone device for which all hardware developments are open to the public, and they also take into account the development ideas of their community [7]. Not only the hardware that is used is made open source, but the main operating system is Linux [41] which enables the user to control every nook and cranny of the device. A large community working with the PinePhone also helps to solve issues, or find related solutions when working with such a device.

2.5 OP-TEE

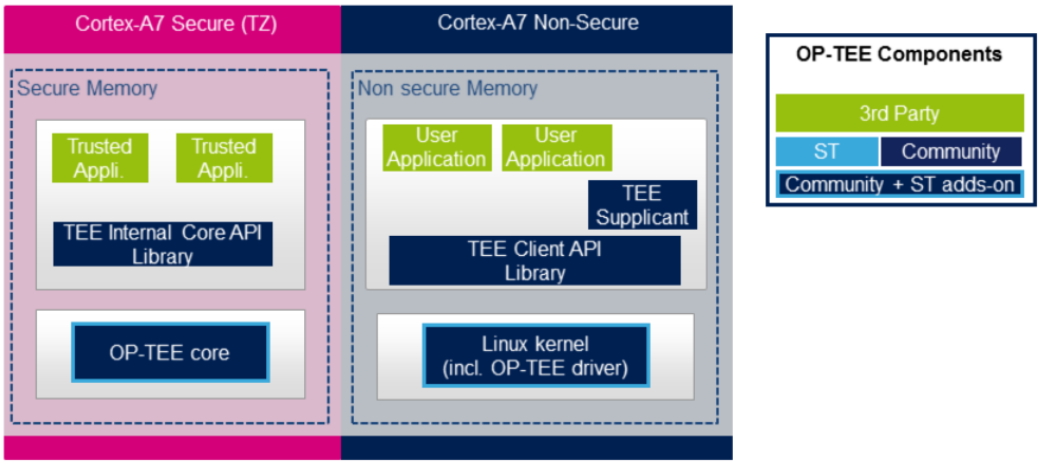

Figure 2.3: OP-TEE structure (Process)

Source: [42]

OP-TEE stands for Open Portable TEE [8]: it is an open-source implementation with ARM TrustZone in mind, but it applies to other realizations of TEEs as well. It implements two Application Programming Interfaces (APIs) defined in the GlobalPlatform API specifications: TEE internal core API v1.1 [43] is exposed to trusted applications, and TEE Client API v1.0 [27] defines how to communicate with a TEE. The main design principles for OP-TEE are isolation, small footprint and portability, as the name describes. As is required for a TEE, it provides isolation from the non-secure OS and shields the TAs from each other. The TEE should also remain small enough to reside in the On-Chip Memory (OCM) of the ARM-based devices. OP-TEE aims at being easily pluggable to different architectures, and has to support various setups like multiple OSs or TEEs.

As can be seen in figure 2.3, OP-TEE allows for third parties to write TAs that can enjoy the added security guarantees the SW has to offer. Despite these trusted applications residing inside the SW, they are not able to compromise the security

guarantees of other TAs running on the same platform. This fact is very important, especially when advocating an open platform where software of different stakeholders can run simultaneously. For the academic community, this is also a very interesting architecture, because the OP-TEE core is open source, meaning it can be extended. These extensions can easily be implemented by creating a Pseudo Trusted Application (PTA), which is similar to a TA but with the same privileges as the trusted kernel. Caution is required, because software bugs in the OP-TEE kernel could introduce security vulnerabilities in the TEE, which defies the added security it provides. Nonetheless, it is very important to experiment with these features and produce research explaining the best practices, pitfalls and innovative ways in which the TEE can be extended.

2.6 Secure boot, trusted boot and remote attestation for ARM TrustZone-based IoT Nodes [1]

The paper [1] about secure boot, trusted boot and RA attempts to enforce system integrity on ARM TrustZone based IoT devices. To achieve this, they need to guarantee system integrity during load time, and during runtime. The load time integrity is made possible with what they call a hybrid booting approach, which consists of a secure boot up to the SW and proceeds with a trusted boot for the rest of the system. To achieve runtime integrity a RA scheme is used, which checks the integrity of executable memory pages of processes running in the NW. With this approach, they endeavor to protect IoT devices against malicious pre-installed programs or malware injection during runtime.

System overview

The threat model assumes that attackers have physical access to the IoT device and can launch a wide variety of attacks, like software and OS attacks. They are able to tamper with the images of the SW and NW (including that of the OS) before the device is booted up. The adversary can also inject malware into the NW during runtime and tamper with the NW applications. Bus snooping, cold boot and cache side-channel attacks are not considered, even though these are possible with physical access. Only the security of the text section of programs is considered. On the other hand, it is assumed that programs that reside in On-Chip Read Only Memory (OCROM) are secure, because this type of memory is hard to tamper with. Lastly, the execution environment of the SW and the RA server are assumed to be trustworthy and secure.

The solution that is proposed uses a hybrid booting method to ensure the load time integrity of the system and RA to ensure the runtime integrity. Secure boot is used to load everything up to the kernel of the SW. This provides strong guarantees that the SW starts in a secure and known state. The NW is booted with what is called trusted boot, which uses attestation to provide proof of the integrity of the

image that is being started. Before the control is given to the rich OS, its image is measured and after it has started, it should send this to the RA server to verify the measurement. The RA service is implemented in the SW. The memory pages of the rich OS are periodically measured using a hash function. The hash digest is encrypted by an attestation key and sent to the RA server for verification.

Hybrid booting

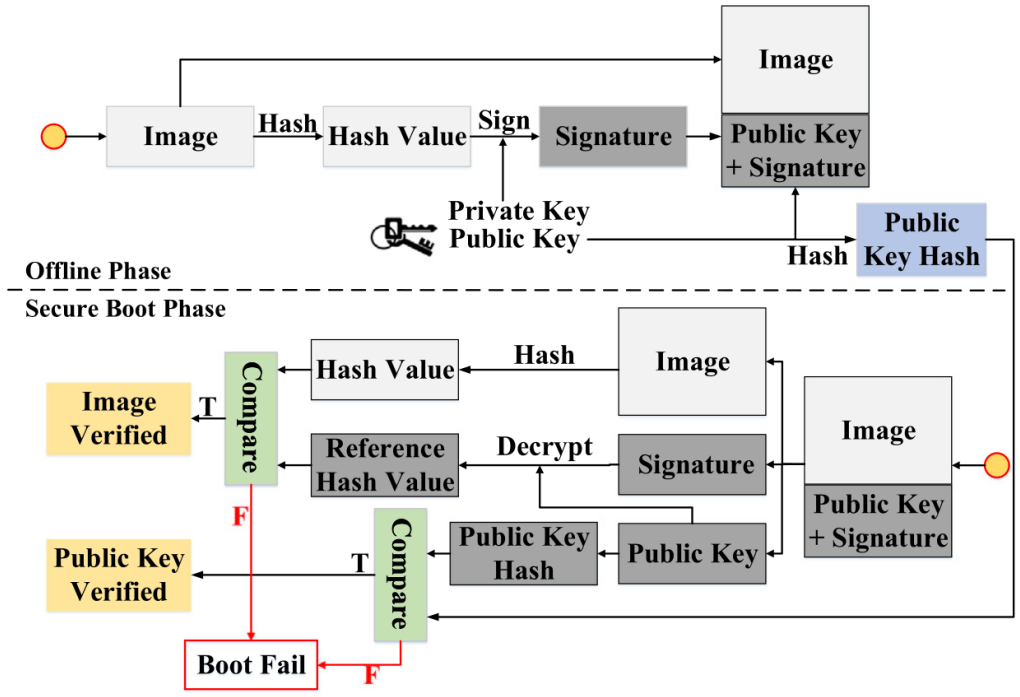

Figure 2.4: Secure boot process sequence

Source: [1]

The setup of secure boot starts with a RoT, which in this case is achieved by using the OCROM and eFuse of the IoT device. The first-stage bootloader is encrypted with a private key and stored on the OCROM to verify the integrity during the boot phase as can be seen in figure 2.4. The hash of the public key is stored in the eFuse to verify its integrity during the boot phase. The images of the second-stage bootloader and secure kernel are measured and signed before deploying the device. These measurements and signatures are stored in the flash memory while the hash of the public key of the secondary bootloader is stored in the eFuse. The hash of the public key of the trusted kernel is stored in the secondary bootloader to achieve an incremental chain.

The secure boot phase starts with the first-stage bootloader which locates the second-stage bootloader, the public key and its signature. Secondly, the first-stage bootloader calculates the hash of the public key and verifies the integrity of the public key. After successful verification, it uses the public key to obtain the measurement result for the second-stage bootloader. Finally, the first-stage bootloader calculates the hash of the second-stage bootloader and verifies its correctness. The second-stage bootloader does this entire process for the secure kernel to complete the secure boot phase.

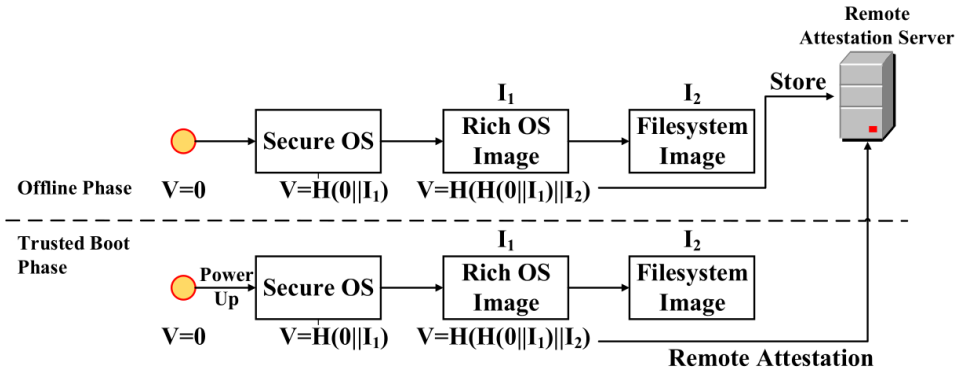

Figure 2.5: Trusted boot process sequence

Source: [1]

Trusted boot is set up by producing a hash chain: the image of the secure OS is hashed first and concatenated with the image of the rich OS, and hashed and concatenated with the image of the file system, and hashed again. This final hash value is stored in the RA server, during runtime this hash value will need to be securely sent to the RA server. To achieve secrecy a symmetric key is used (the attestation key): storing this in the IoT device is not trivial, so this is solved with the following method. The Cryptographic Acceleration and Assurance Module (CAAM) is used to execute cryptographic functions in a secure environment. This module is used to generate a 256-bit blob key, which is used to encrypt the attestation key that has been agreed on with the RA server. A Message Authentication Code (MAC) is calculated from the attestation key to ensure its integrity. The blob key is itself encrypted with a Blob Key Encryption Key (BKEK), which is derived from the Master Key (MK) by the CAAM. The MK is stored in Secure Non-Volatile Storage (SNVS), which is assumed to be secure by default. To prevent the NW from ever gaining access to the attestation key, the CAAM is configured to be only accessible from the SW using the TZPC, as discussed earlier.

The trusted boot phase starts with the NW attestation client application establishing a Transport Layer Security (TLS) connection with the RA server, and requesting

a nonce. The measurement TA restores the attestation key from the blob, using the CAAM. The TA measures the rich OS and filesystem images, appends the nonce to the final hash value and encrypts this combination with the attestation key. The encrypted text is put in shared memory to allow the client application to send it to the RA server. On the RA server, the cyphertext is decrypted and verified to check for integrity violations and replay attacks.

Page-based attestation

The idea is to measure the code segments of the programs in the NW on the IoT device. It is assumed that this code base does not change in the lifetime of the device. The SW is trusted but the NW is still vulnerable to attackers, therefore attesting the code in the NW would increase the security of the applications running in the NW. The measurement is done on pages of 4 KB at a time so programs will have multiple tuples of the form *{process-name, page-hash}* which will later be used to verify the integrity of the process. The first measurement is done before deploying the IoT device, and the results are stored on the RA server to be able to compare the future measurements with.

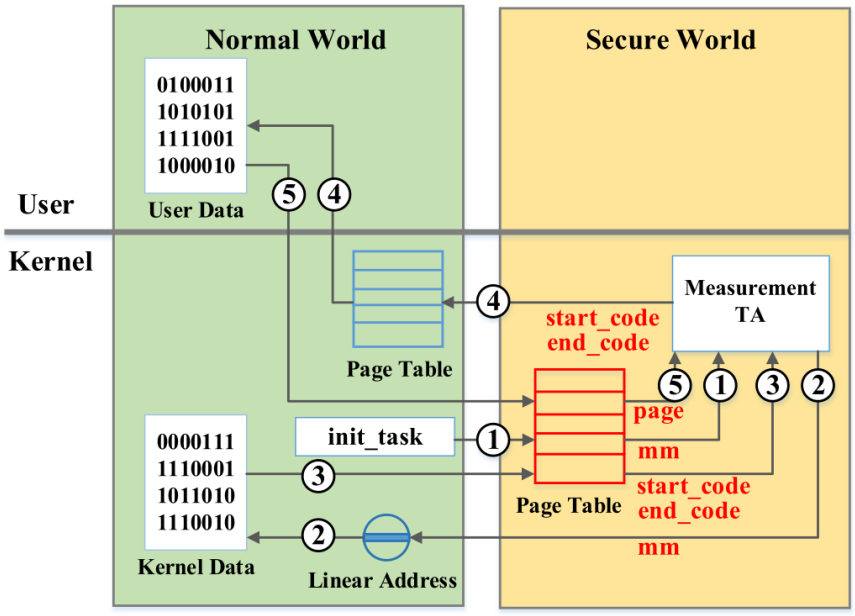

Figure 2.6: Process measurement sequence

Source: [1]

Process integrity measurement starts with the kernel of the NW providing the *task_struct* of the initial process *init_proc* to the measurement TA (1). The

task_struct is a data structure containing all information about a process, for instance, the virtual address where it resides (2). This address is sent back to the kernel of the NW, because only there it can be investigated to find out about the boundaries of this memory region (3). The fourth step (4) is to search for this memory region in the kernel of the NW from where the process can be accessed. Next (5), the executable code pages of the NW process are sent to the SW, which is achieved by storing them in shared memory space. Based on the *init_proc*, the TA is able to iterate over all processes and measure their code pages. These *task_struct*s have the structure of a doubly-linked list, so it enables the TA to find all processes during the measurements.

Process integrity attestation uses the measurement of the process integrity measurement stage. First, the IoT device requests a nonce from the RA server, with which a TLS connection is established. The measurement TA encrypts the measurement results concatenated with the nonce, using the attestation key. Based on the assumption that the attestation key is kept secure, and the execution control flow of the measurement TA is secure, also guarantees the freshness of the measurements. If any of those two assumptions were violated the attestation response could be forged by an adversary, because that the nonce is only concatenated with the hash digest. The measurements of the processes are encrypted individually, meaning that a set of cypher texts is sent to the RA server. The RA server decrypts the measurements of the processes and checks whether integrity violations can be found, which means that there is new software or old software has been modified.

Evaluation

The effectiveness of the secure boot process is measured by whether the secure boot phase can detect any violations against the integrity of the images, the signatures or the public key. All of these inconsistencies are exposed by the secure boot process correctly. For the trusted boot the focus lies on whether the RA server can identify an abnormal system status. This is the case because NW programs can still be executed, but the RA server will verify the system state. The process integrity attestation is tested by tampering with existing programs and inserting additional programs. Both cases are picked up on by the attestation.

Performance of the boot procedures is measured by comparing the mean time of 30 iterations with secure and trusted boot, and 30 iterations without it. The secure boot adds little overhead on the second-stage bootloader, while the trusted boot almost doubles the time it takes for the secure kernel to boot, which comes down to a little over a second being added. The main reason why the secure kernel takes this long is that the image of the filesystem and rich OS is rather large and takes some time to measure. The overhead of the measurement TA and the attestation CA is measured by calling rich OS services, while these modules are running and with them disabled. The overhead these modules introduce lies between -0.55% and $+0.67\%$.

Security analysis is executed on the hybrid booting approach and the page-based process attestation. In the booting method a CoT is constructed from the RoT residing in the eFuse and OCROM. A successful secure boot ensures that the SW can be seen as the secure base from which the NW can be booted. If the NW image is tampered with, the RA server will pick up on this threat. The execution of the measurement TA and the results it generates are both secure because of the isolation in the SW. The results pass through the NW, but they are encrypted at this stage; the encryption key is also securely stored in the SW, giving the NW no opportunity to get hold of the information. The main drawback of this approach is that the method relies on the rich OS to access the paging structure and process management kernel objects.

In [1] it is correctly mentioned that the proposed solution does not detect self hiding malware, for example, transient rootkits. There are, however, more shortcomings when the solution is investigated. The attestation method is not sufficient to detect all software attacks due to it not checking the integrity of key aspects of the system like data structures, system calls, etc. The same reasoning can be applied to OS and firmware attacks. The fact that they state: *"We only consider the security of the code section of a program"* is honest, but alarming nonetheless.

Chapter 3

Method

3.1 Detailed Problem

When people interact with their bank or hospital through online services accessed from a personal computer or a smartphone, they expect their data is secure. In other words: confidentiality, integrity, and authenticity are guaranteed. On a personal computer, these requirements have become evident, and many technologies for cyber security exist [44], [45]. The problem with systems like smartphones and IoT devices is that their hardware is not as extensive as that of a personal computer, making many of these technologies unfeasible. So, while people often have the same expectations towards security, in both cases this is still a challenge when it comes to smartphones. Therefore, the security of a system like a smartphone is very crucial because lots of people are unaware of the possible threats they face.

Adversaries have many possibilities when it comes to stealing sensitive data from smartphone users while the data is being used during execution or stored on the device. [46] for instance, propose AlphaLogger, which infers the letters being typed on a soft keyboard based on the vibrations, and [47] were able to achieve identity theft through data cloning of auto-login credentials. Besides custom attacks, there are plenty of well-known risks as well. [48] have executed a review of various malicious software threats and mitigations, [49] researched software attacks that take advantage of hardware resources to conduct fault injection or side-channel analysis, and [50] reviewed the possible threats that can be introduced by smartphone providers altering the Android OS to add their signature flavor. These attacks are possible due to smartphones being inferior to personal computers in terms of hardware. As similarly to IoT devices, a lot of focus lies on the performance of the final product. This performance is often hard to achieve because the resources want to be kept to a minimum to lower the price or keep the device small. Security is still seen as a performance killer, which is very unfortunate because it should play an essential role in the design and implementation of a system.

3.2 System Model

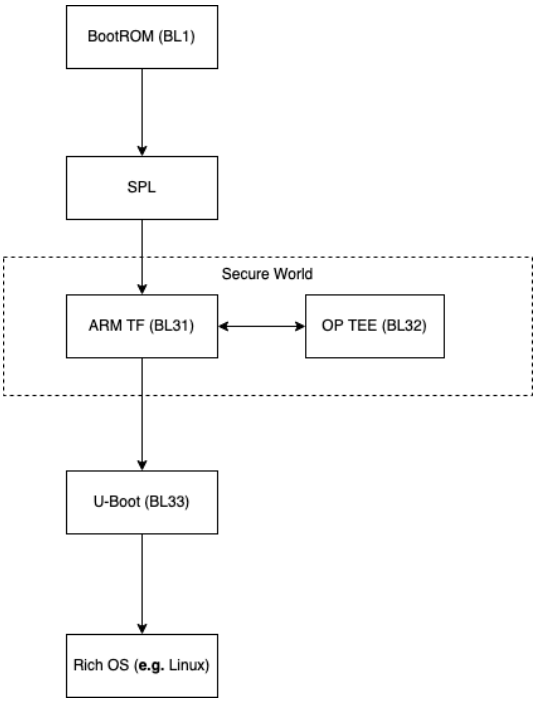

Figure 3.1: PineA64 boot flow for OP-TEE and Linux

Source: [51]

The system model used in this thesis is an open system. With an open system, we mean that the owner is not constrained by any means to how they desire to use their device. These restrictions can go from limitations on what software can run on the device, to certain functions that cannot be controlled by the user in any way. The openness discussed here is not the same as open-source systems, but it shares many properties. In an open-source system, all the stakeholders of the system can inspect every detail about the system they want. Using this technique the stakeholders can gain lots of knowledge about the system, which could facilitate interoperability with certain software or devices. The stakeholders could also gain confidence in the qualities and capacities of the system if they can investigate these details. This is the type of confidence and understanding our open system should also provide, but it is mainly focused on not constraining the owner in their capabilities while using the device.

The concrete system is a smartphone that is owned by a user but for which multiple software providers offer applications. These providers don't necessarily trust each other but they do want guarantees that the execution of their software will not be

interfered with by software of other providers. The user of the device is also its owner, with full control over what software can be installed and run on the platform. In an open system like the PinePhone, these values are pursued to minimize the constraints the user experiences while using his device. This is very different from large smartphone companies where the company has a signature key which is kept secret. Programs that are not signed with this signature key will be rejected for installation on the device.

The smartphone, in this case, is a PinePhone which is equipped with 4 ARM Cortex A53 Cores that are TrustZone enabled. Lots of smartphones have ARM processors as SoC, which means that ARM TrustZone is the ideal candidate for the implementation. For this TEE to work correctly, a trusted kernel is required. OP-TEE is used, which provides the interfaces to communicate with the TEE and call TAs from the NW. The NW runs a Mobian Linux distribution as kernel which is specifically written for smartphone devices. The system can be put together as described in [51], and shown in figure 3.1 using the source of the Mobian distribution [52], OP-TEE [53], U-boot [54] and the ARM Trusted Firmware [55]. Unfortunately, this was not realized in this thesis, so the QEMU emulator [56] was used to test the code and run the experiments. QEMU allows running OP-TEE on a desktop while emulating the TEE. Because OP-TEE can run on the PinePhone, the code and experiments should be reproducible on the PinePhone if correctly configured with OP-TEE.

3.3 Attacker Model

The adversary we are trying to protect against intends to get access to sensitive data from the device or execute arbitrary code on the device. By achieving this, he could abuse the personal information of the victim or his hardware respectively. The attacker is assumed to have physical access to the device, which could be achieved by stealing the device or through the evil maid attack, where the device is unattended during a period of time and the attacker has temporary physical access to it. He could also launch OS attacks which means he exploits a security vulnerability in the rich OS, which allows him to take control of it and with this control try to maliciously modify other parts of the system. Exploiting a security vulnerability inside the OS and taking control over it by itself, is also undesirable behavior. Lastly, the attacker can execute software attacks which comprise insertion of malware, tampering with the control flow of executing processes, and so on. The attacker is not able to break encryption algorithms deemed to be secure by the literature and cannot violate the security guarantees a TEE provides.

Physical access brings along lots of risks because the adversary has a variety of possible attacks he could launch from this position. The TEE can, when combined with some specific hardware, make these attacks ineffective or a lot harder to execute. For instance, reading from the hardware memory becomes infeasible because

everything on there that is sensitive is encrypted by the TEE. Only the TEE can decrypt this information. The decryption key is stored in a secure key store or a TPM or can be derived from the RoT. Tampering with memory can be prevented by using secure boot. It ensures that the TEE is started up in a secure and known state from which a trusted base can be ensured. With this trusted base, attestation could be run on the memory to check whether inconsistent memory pages can be found before they receive control in case they are code pages. Another hardware attack is one where a physical back door is exploited. This is assumed to be impossible because the processor has been designed for security purposes and thus no back doors should be available. The main advantage defenders have in terms of hardware attacks, is that they are often very hard to execute. There are still lots of hardware attacks for which TEEs are vulnerable: manipulation of RAM and eFuse bypassing secure boot [57], micro-architectural structures leaking information [58], and electromagnetic analysis of side-channels [59]. These attacks are very advanced and would be hard to execute, which is the reason why they are not taken into account in this thesis.

OS/Firmware attacks can compromise all user-level applications because the OS is the trusted layer on which these user-level applications rely. This is the main area where ARM TrustZone and other TEE implementations make a very big difference in security. ARM TrustZone, for instance, is implemented below the rich OS, which means that the trusted kernel has more privileges than the latter. The rich OS has of course more functionality, but to achieve this functionality, it will in some cases rely on the trusted kernel. In case an OS attack has succeeded, a TEE increases the security by shielding the user-level applications from this OS. This is done by allowing the user-level application to request services like trusted I/O and secure memory from the TEE itself, without interference from the OS. Examples of this are a container using ARM TrustZone [39], securing camera and location peripherals [60], and checking whether the OS executes system services correctly [61]. It does not make claims about preventing OS attacks, because the vulnerabilities in the rich OS are still present. The NW could be attested which would allow the detection of an OS attack, but the TEE doesn't have special protection mechanisms to avoid it from happening. This is not surprising of course, since OS attacks are often a consequence of software bugs in the implementation that give rise to security vulnerabilities. These bugs are very hard to avoid due to the large number of lines of code in these rich OS implementations.

Software attacks try to tamper with the control flow of certain program executions, or get hold of sensitive data through malicious code. Most of these attacks can be defended against by a TEE implementation, or with the help of a TEE. Applications can be isolated from each other, making it a lot more difficult for a malicious application to tamper with the execution or data of another one. This isolation can be enforced using virtualization or specialized TAs, making sure the application can only be interacted with, using a very well-defined interface. The virtualization needs to be implemented correctly to make sure no data is leaked in between the partitions.

In the case of the TA, it can use the TEE functionality to store its data securely. Software attack defense mechanisms based on ARM TrustZone come in many forms, isolate application and secure communication [62], control flow integrity scheme [63], and reverse engineering protection [64].

3.4 Solution

First things first: to make sure the TEE can provide the security it guarantees, it needs to be started up using secure boot. This ensures that the device starts in a known secure state, which can therefore be trusted. To achieve this, a RoT is needed from which a CoT can be constructed. The RoT is often implemented by using a TPM along with hardware memory specifically designed for secure storage, for instance, an eFuse or OCROM. As discussed in [65] the PinePhone provides write-once memory in which the hash of a public key can be placed. This key can be used to serve as RoT during trusted boot on the device. Depending on how this memory is used, a discussion about *"What can be used as a RoT in such an open system?"* can be had.

When the RoT and subsequently the trusted boot are taken care of, we need to utilize the TEE to improve the security of the smartphone. The ultimate goal would be to guarantee the integrity of the control flow, code, and data on the device, which is a very strict requirement. An achievable target would be to detect violations to this integrity. Attestation can be used to check the integrity of an application or a running system depending on what properties are looked at to measure the reliability of the target. RA is not logical in the context of a smartphone, but the user (who in our case is seen as the owner of the device) could be alerted about the integrity checks. This train of thought will be followed and partially implemented to attempt to answer the question *"Can attestation be used to increase the security of an open platform without disrupting the openness of the system?"*.

The user plays a critical role in this solution. This is due to the requirement that the device should be secure while the user stays in control. The user can decide which software he considers secure and allow to run on his device. On the other hand, the software providers do not want another process to interfere with the execution control flow of their programs. This can be achieved by having the attestation process run within the TEE on the device, making it tamper-proof against software and OS attacks. The interests of all parties need to be taken into account during the evaluation of the solution to be able to answer the question *"Does this security solution meet the expectations of all smartphone stakeholders?"*.

Ensuring the openness of the system is not violated by the proposed security solution, has consequences on the achieved results. These consequences are thoroughly looked at during a comparison with similar security approaches that do not take into account the openness of the system. Besides these comparisons, also possible

extensions are discussed that have been implemented by other work but don't require the system to be closed, so these could also be seen as directions for future work. These comparisons attempt to answer the question *"How does this type of security solution compare with the ones in closed systems?"*

Chapter 4

Implementation

The goal of the implementation is to guarantee that the executable memory pages that are loaded into the memory are not tampered with. In general, this is achieved through RA where the device proves its trustworthiness to a remote verifier, in this case, it is the trusted base of the device itself that checks the integrity of the REE. This integrity check is achieved by measuring the code pages of the processes in advance, for instance, during the installation of the particular code. This measurement can be done in a variety of ways. In this thesis, hashing was chosen because of its common use and it being well understood by the community. After these initial measurements have been taken, they need to be stored in a secure place. The secure memory of the PinePhone provided by ARM TrustZone was chosen because it can guarantee the integrity and confidentiality of the data. Last but not least, these measurements need to be retaken by a trusted base (SW) during execution time, and they need to be compared with the initial measurements to verify whether the running code has been tampered with. These runtime checks need to happen continuously to guarantee no modifications are introduced, or that the changes are observed before they can damage the system too much. Measuring code all the time is of course not feasible. Periodic measurements are more appropriate, but they should be scheduled unpredictably to avoid Time Of Check to Time Of User (TOC-TOU) attacks.

4.1 NW application

The NW application (CA in figure 4.1) begins with looking up the processes that are currently running by examining the */proc* directory (1). This directory stores all relevant information concerning processes in Linux. In this directory there is a collection of directories named with numbers. These numbers represent the Process IDentifiers (PID). The PID is used within Linux to allow the operating system to differentiate between the different processes and manage them. It is of course necessary to attest all these processes. However, the explanation will focus on just one, for now, to keep things clear. Take for instance the directory */proc/1*, which will always exist since PID 1 is associated with *proc_init* which is the initial process the

Linux OS forks. Within this directory there are two important files: *pagemap* and *maps*. The *maps* file gives an overview of the different memory regions associated with the process. Besides the virtual address range, other information is also present, for instance, whether the pages are executable or writable, and where their symbolic origin is in the file structure. The *pagemap* file, on the other hand, is necessary to translate from virtual address to physical address.

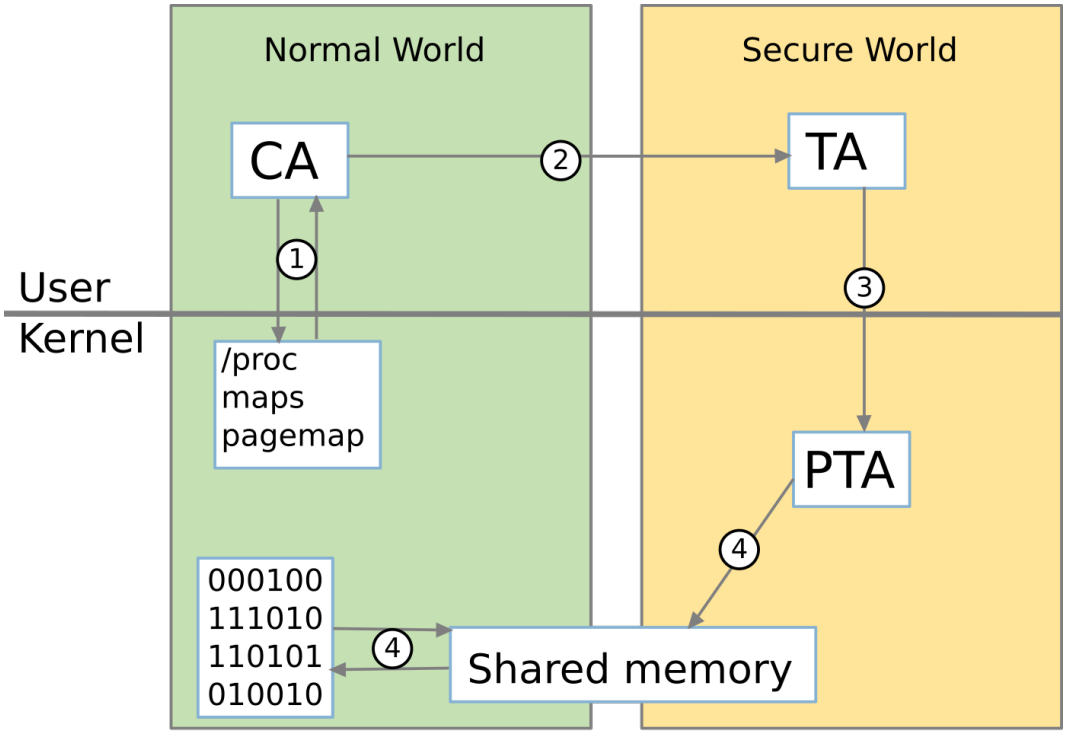

Figure 4.1: Process measurement sequence

Translating the virtual addresses that are found in the *proc/pid/maps* file into physical addresses, is based on a solution from [66]. The solution investigates the *proc/pid/pagemap* file and generates a *pagemap_entry* from it. This *pagemap_entry* is used to structure the information associated with one entry in the *maps* file. Based on this *pagemap_entry* the physical address can be derived from the virtual address found in the *maps* file. Do note, that the *maps* file provides a contiguous virtual memory region. It is not guaranteed that the physical memory will also be contiguous, so the first virtual address of every page is translated into a physical address. Also, the virtual addresses in the *maps* file are virtual addresses in the Linux environment, as seen on the bottom left side of figure 4.1. Pages are 4 KB large, so lots of virtual addresses are translated into physical ones. After the physical addresses are obtained, they are put together into a list, and a memory reference of this list is sent to the SW for further investigation.

4.2 Measurement PTA

The PTA could be called directly if it is configured correctly but to keep things simple here the communication goes through a normal TA (2). The measurement PTA receives a memory reference with inside the buffer all the physical memory addresses of the pages that need to be attested (3). Before the SW is able to access these memory pages, they need to be mapped into its virtual memory space (4). The mapping is done using the *core_mmu_add_mapping* function from the OP-TEE kernel [53]. When this mapping is successful, the virtual address in the TEE (the right part of image 4.1) can be obtained using *phys_to_virt*. The function *phys_to_virt* returns the virtual address in the PTA where the memory can be accessed. With the PTA having access to the memory pages, the actual measurements can start. The hashing algorithm used is SHA-256, which can easily be substituted by another algorithm provided in the OP-TEE library. The hashing algorithms are provided by the OP-TEE framework and are easily usable from within the PTA.

In the initialization phase of the integrity checking, the hash digests are stored in the secure memory of the device. The files written from the PTA to secure memory are only accessible by this PTA and are protected against everything in the NW. In this file, the hashes are sorted based on which *pagemap_entry* they come from and the page number within this region. Later, this allows to uniquely identify the hash value that will be used for comparison. After the initial hash values have been stored, the initialization phase is complete. This implies that the security of the stored measurement values is guaranteed, under the assumption that the SW does indeed protect the secure memory against access (read and write) from outside the SW.

During the attestation phase of the integrity checking, all the same steps will be taken as the initialization phase has taken up to this point, except for storing the calculated hash values. Instead, during this phase the newly calculated hash values will be compared with the hash values that are stored in the protected file where the initialization phase has written the measurement results. For the comparison, the hash value and the initial value are put into 4 *uint64_t* data types each. The first one from the hash is then compared with the first one of the initial value, the second one with the second one and so on, using the standard integer comparison functionality. The number of comparisons that fail is saved and printed in the debugging output stream of the SW. Ideally, the user should be notified about these faults, and which processes may experience an impact from this, based on whether the process uses the code for which the attestation has failed. Based on the information the user receives from the attestation, he can decide what action to take as the owner of the device.

4.3 Improvements and extensions

First and foremost, it is very important to avoid relying on the rich OS for the translation of addresses, especially because this functionality is only possible with root privileges. This is because the data structures that contain this information are owned by the NW OS, and the necessary files are privileged which causes the accessibility issue. If possible this should be improved because an attacker in control of the rich OS could alter these data structures to hide the changed processes from the attestation PTA. The adversary could, for instance, keep an unchanged process hierarchy loaded in memory while the device is actually running on a different malicious process hierarchy. As long as it cannot be guaranteed that the addresses the SW receives are the addresses of the actual processes that are running, the attestation can be bypassed.

Secondly, the attestation currently only measures the executable pages present in RAM. The *pagemap_entry* only provides physical memory addresses for the pages that are loaded into RAM. This is sufficient during runtime, but for the initialization phase all the possible executable pages need to be measured to have a reference value to compare with. To achieve that, the entire code base of the process needs to be measured during the initialization phase. The binary files of the modules can be looked up to solve this issue. If it is possible to access these rich OS files from within the SW, they could also be attested as if they were loaded in memory. By measuring from these files, the initialization phase is not bound by the executable memory pages present in RAM anymore and can attest all pages. Having an initial value for all code pages is important to provide strong additional security guarantees. Measuring a memory page for the first time while the device is already deployed is bad practice, because there is no guarantee that the memory page has not been tampered with already.

A significant extension to allow this proof of concept to be turned into a complete attestation solution is to notify the user via trusted I/O. When informing the user about the results of the measurements, it is important to keep in mind the security of the channel on which this is achieved. ARM TrustZone provides trusted I/O paths which can be used for this goal. It could be used to inform the user of the problem, which program has been tampered with, or it could also take on a more coarse-grained approach like a Light Emitting Diode (LED) signal that some piece of software has failed the attestation, which is easier to achieve but less useful to the user. What details can be derived from this communication is not necessarily of the greatest importance. It is important how this communication works. To make sure the NW cannot interfere with the communication from the attestation PTA to the user, it should be implemented using the provided APIs of ARM Trusted Firmware-A (TFA) [55]. Secure I/O paths are an extensively researched topic in the field of TEEs, for instance, to protect user data from compromised browsers [35] or to have a general trusted I/O path between the user and trusted services [67]. Using secure I/O is necessary to build a fully functional solution based on the

proof of concept presented in this chapter. The secure I/O was not included in this work because no new functionality would be showcased and it would divert too much from the main focus of the implementation, namely checking the integrity of the NW processes.

The second major extension focuses on the added security guarantees the attestation process provides. To achieve great security guarantees it is necessary to do more than just code attestation. Lots of software attacks are based on the used data structures and do not impact the text section of programs. As discussed in the background there are a variety of methods to detect software attacks based on used data structures and a great example of an extensive attestation method is [20]. These forms of attestation are of course more complicated to execute and take a great amount of implementation effort. The attestation for this work was kept relatively simple to show that this form of attestation is achievable on the PinePhone. When designing a final product, it is of course encouraged to use more advanced techniques to increase the security guarantees of the code execution on the device.

Chapter 5

Evaluation

The main purpose of the experiments executed for this thesis is to provide a better understanding of the practical feasibility of the solution for its stakeholders. The performance experiments are based on those of the paper [1] discussed in the background section. To be able to compare the results of this work and that of the paper, the experiments need to be made as similar as possible which will be clearly visible. Secondly, for the security evaluation the solution of this work will be evaluated and the differences with the security evaluation of [1] will be highlighted.

5.1 Experiment setup

As was mentioned in the system model the experiments were not executed on the PinePhone itself, but on the QEMU emulator [56]. These emulations were run on a general-purpose laptop (Lenovo Thinkpad P50). The execution of the experiments themselves was rather straightforward. First, the emulation needed to be started, here all changed files were recompiled and the whole environment was put together. In the QEMU environment, we had to log in as root to enable the client application of the solution to access the necessary Linux files. When this was successful the solution had to be started, which was achieved in a similar way to how the OP-TEE example TAs are started up. This is because these examples were used as a source of inspiration, for how to put together a working OP-TEE application with TA (and PTA) interactions.

5.2 Performance

The trusted boot experiment in [1] is based on the attestation of the NW before giving it control. The paper states a performance of 107 MB of file system image being measured in 1.276 seconds. This could be a valuable starting point to compare with the performance achieved in this thesis. Because the implementation is focused on measuring code pages, there will be a difference in results. While it can be somehow assumed that the memory space of the file system in [1] is contiguous, or at least the mapping can be found rather easily, the code pages in the implementation

of this work are spread across multiple processes which need to be considered one by one. Looking up the right memory pages to measure them incurs additional overhead which needs to be taken into account. That is why we opted for three different experiments that relate to this trusted boot experiment from [1]. One experiment will measure all the code pages of all the processes in the simulation, the second will measure all code pages of one process and the last experiment will measure one contiguous memory region of code pages from one process.

The overhead of running the attestation module is measured in [1] by executing system services from the Linux kernel and running this experiment with and without the attestation module being active. The results in the paper of this experiment shows overhead between -0.55% and $+0.67\%$ on different system calls. In our opinion, these results do not give much insight into the matter, because some key information is missing to correctly assess the results. First of all, it is not mentioned how often the attestation is running while these system calls are made. Secondly, it can be assumed that the attestation does not get priority over the system calls but this is also not explicitly mentioned. Last but not least, the range of -0.55% and $+0.67\%$ does not really tell much, as this could just be standard deviation. For this experiment, it was not mentioned how often it was executed, which gives the impression that this is a one-time measurement. However, the authors of the paper are explicit about calling each service 1000 times, with intervals of 250 ms between each call, and they do this with 7 different system services, which, as they also include, adds up to almost 30 minutes.

5.3 Performance Evaluation

The total amount of loaded code pages in RAM is around 1,850. The time it takes to measure these pages twice (once for initialization and once for attestation) is about 25 seconds. A plot is shown in figure 5.1 which provides the generated data points from the experiment being executed 10 times. Also a linear regression line can be found to provide the general correlation between the number of pages measured and the time it takes.

To provide insight into how long it takes to measure one process (twice), the last experiment was executed on the *init_proc* process. The results of this experiment can be seen in figure 5.2.

In the first two experiments the amount of pages that are loaded into RAM fluctuates a bit. Which probably has to do with pages being swapped in and out of memory. In this last experiment, only one executable memory region of the *init_proc* is measured and the amount of pages stays constant for all 10 iterations. The amount of pages measured is 88 and the average execution time is 1.319 seconds, the maximum execution time is 1.353, and the minimum 1.299.

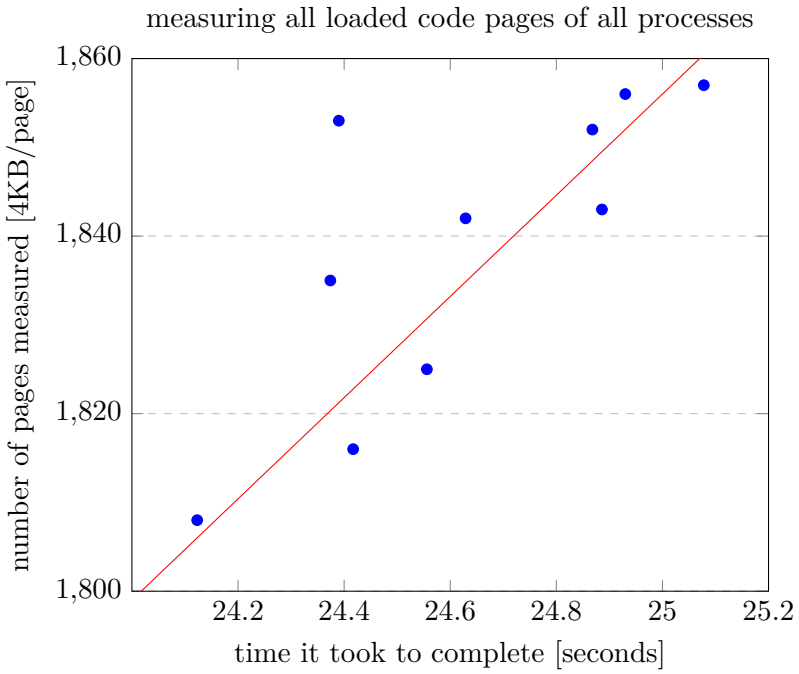

Figure 5.1: Measurement plot for all processes

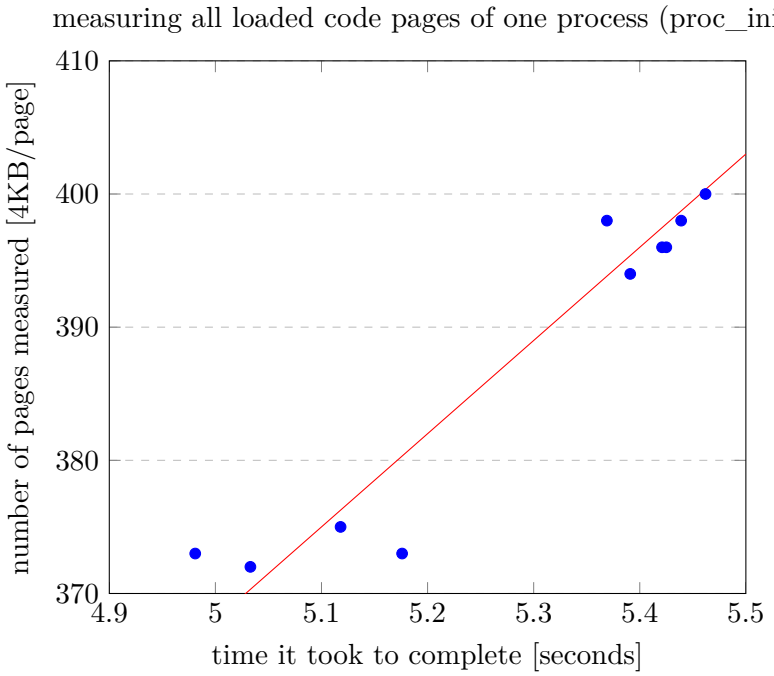

Figure 5.2: Measurement plot for one process

Last but not least, the three experiments are plotted in one graph in figure 5.3 to show how they relate to each other. The mean values of every experiment have been taken to fit each experiment in one data point. They would be very close to one another anyway. This plot clearly shows that the amount of time the measurements take is proportional to the number of pages that are measured. The reason for this is that hashing the memory pages is the most compute-intensive task of the program, and the file overhead is similar, because for every process two files need to be read. This graph allows us to compare the performance of our experiments to that of [1], because we can extrapolate the results to the necessary amount of pages that were measured in the paper. In the paper, 107 MB (of file system image) was measured in 1.276 seconds. We achieve a similar execution time for only 88 executable pages. As explained these were measured twice: once for the initialization phase and once during the attestation phase. This comes down to 352 KB measured twice or 704 KB, which is less than 1% of their measured memory. If we extrapolate based on the number of pages, it is best to start from the largest experiment to have the least rounding errors. The mean of this first experiment comes down to 1,839 pages measured in 24.625 seconds, 107 MB equals 26 750 pages. Measuring this many pages would require our implementation a little less than 360 seconds or 6 minutes.

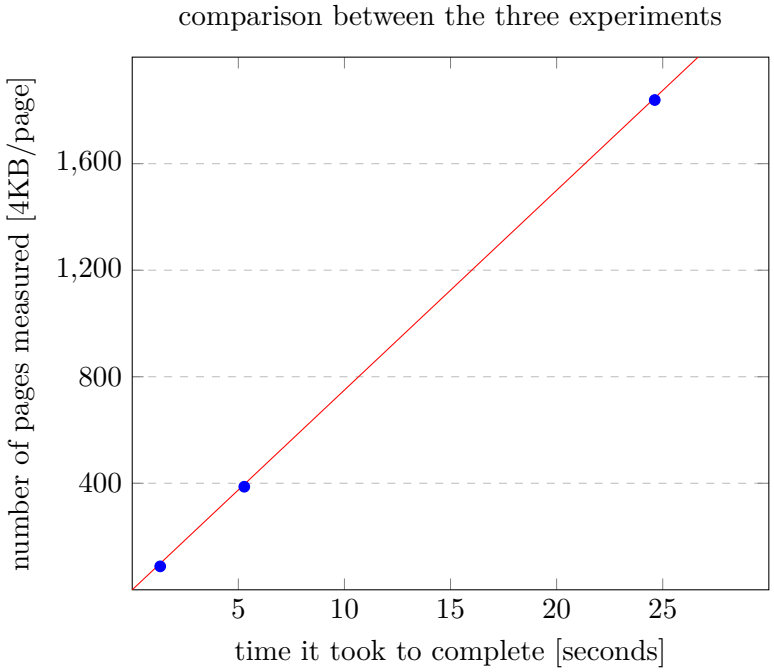

Figure 5.3: Comparison of the three measurement experiments

As can be seen in the three executed experiments, the performance of the implementation of this work differs a lot from that of [1]. Some aspects need some explaining, however. First of all, the implementation is not written nor fine-tuned for

performance, it is merely written as a proof of concept. This is due to the memory regions that are attested in this work being only the executable memory pages of a process. These pages need to be identified using multiple files, which generally takes more time compared to the case with one large chunk of memory that is attested. It is also important to take into account that executable pages take up less memory than a file system would generally do. Lastly, these experiments are executed on a Qemu emulation of OP-TEE on a general-purpose laptop (Lenovo Thinkpad P50), while the experiments in [1] are executed using a hardware prototype.

A good balance between performance overhead and security assurance depends on the use case. In the case of a smartphone, we believe even with these results that attesting the executable pages that are loaded in RAM every 30 minutes has not too much impact on the user experience. Depending on the amount of newly loaded memory pages when an application is started, it could even be considered to attest these pages before starting to execute them. This could impact the user experience much more because this happens while the user is waiting for the application to open. While on the other hand, running the attestation in the background on the already loaded pages in RAM, should have minimal impact on the user. These considerations do not take into account the energy consumption of running the attestation, because we didn't have the means for these kinds of experiments.

5.4 Security Properties

The attestation method presented in this work focuses on measuring the code pages of processes loaded in RAM. A code page can only gain control or be executed when it is loaded in RAM first. Keeping this prerequisite in mind, it should suffice to only attest those pages to ensure no corrupted code page gets executed. To put it more mildly, the measurements should run periodically, so a code page may have been executed before the changes are noticed by the attestation PTA.

The integrity of the measurement execution is of utmost importance when it comes to attestation. In RA, the critical code runs on a hardened server, which guarantees that it is infeasible to tamper with the execution control flow. In the case of the PinePhone, the RA method is not used. Instead, user-controlled attestation is applied, which allows the user to verify the trustworthiness of his device. The critical code which compares the measurements runs in the SW. The TEE provided by ARM TrustZone does guarantee that the NW cannot influence the execution of the SW in any way, as long as the device was started up successfully using secure boot.

Secure storage of results is very important for a liable attestation method. If an adversary were able to tamper with these values, they could make every attestation attempt fail due to the changed initial value. In case a bad hashing algorithm is chosen and the attacker can read the initial hash digest, he could also try to forge a collision attack [68] where he tampers with a page in such a way that its hash

digest does not change but the code in the page does something the attacker desires. To make sure the reference values are not tampered with, they need to be stored in secure memory which should not be accessible from outside the SW. In the SW, these values should only be written during the initialization phase and afterward only read. This needs to be stated less strictly. In case there are software updates or additional software to be installed on the device, the initial values will have to be overwritten. The initialization phase could be executed again to update or add the measurement results for those code pages.

5.5 Security Evaluation

Security guarantees that can be made, are the integrity of the measurement execution control flow and the integrity and confidentiality of the results that are being stored. These are achieved thanks to secure boot process enabling the TEE of ARM TrustZone. These are key assumptions in the field of RA, and thus are also very important in this case of user-controlled attestation. Of course, when discussing the security guarantees of a certain solution, it is not only about the execution and storage of the measuring application itself, but also about what additional security guarantees it provides to the system overall. The most important guarantee that an attestation solution tries to provide, is being able to detect modifications or unexpected possibly malicious changes to the aspects of the system it attests. In our case, the code pages of processes are being attested, we only measure the ones that are present in RAM, because only then they are able to harm the system. As stated in [1] there is still an unsolved problem, the rich OS needs to provide the physical addresses of the code pages. This means that malware capable of self hiding or transient rootkits are still possible threats that could stay unnoticed by this solution. If an adversary has taken control of the OS, there are countless ways in which the attacker could deny the attestation PTA to obtain the necessary addresses, which would be very hard to identify and alert the user.

OS/firmware attacks are present in the attacker model of [1] on which this work is based. We believe that with this method the code pages of all processes, including those of the rich OS, can be attested and this will enable the user to be notified about any tampering with this code base. This does not mean, however, that tampering with these code pages has become harder. Besides tampering with code, there are also lots of different methods to perform an OS/firmware attack. We believe that this work can be extended to thoroughly attest the NW, but in its current state it only allows detection of a small portion of the OS/firmware attack surface.

It is also claimed by [1] that software attacks are protected against or detectable in the case of attestation. Again, with their addendum about only securing the code section of processes, this is valid but the integrity of the code is only a small portion of the attack surface. In their measurement method, the data structures are not checked, and those are the main target for very well-known attacks that have been around for

decades, like buffer overflows and return-oriented programming. Extensions to this solution need to be realized to detect software attacks, OS/firmware attacks likewise.

Chapter 6

Discussion

6.1 Related work

The paper on RA, secure and trusted boot on IoT Nodes [1] is the paper on which the implementation and experiments of this thesis are based, and is extensively explained in the background section. To provide a better understanding of the effectiveness of their solution, it is compared to three related works. Preceding this comparison, these three papers are briefly explained with in depth explanations on the parts that are most relevant for the comparison. The main focus of this comparison will be on the added security guarantees and the assumptions that have been made with respect to the RA. After the comparisons of the related work, the weaknesses and possible improvements of the work performed in this thesis will be discussed.

SecTEE: A Software-based Approach to Secure Enclave Architecture Using TEE

The solution described in the paper about SecTEE [69] aims to implement a framework using ARM TrustZone to achieve similar security guarantees as a hardware-based secure enclave architecture. The authors provide the following contributions with their paper. Their first contribution is the implementation of SecTEE, the new secure enclave architecture which achieves *"the highest level of security"* using ARM TrustZone. Secondly, a locking mechanism is introduced which makes sure enclave pages cannot be accessed while the enclave is running to prevent cross-core side-channel attacks. The TEE OS is also extended to provide functionality like identification, RA, and sealing sensitive data. Lastly, an implementation of SecTEE based on OP-TEE is provided along with experiments showing the performance of the proposed system.

Their wording of *"the highest level of security"* means that it should be resistant to privileged host software attacks, board-level physical attacks, page fault based side-channel attacks, and cache based side-channel attacks. The hardware attacks on

which is focused are cold boot attacks [70], bus monitoring attacks [71], and Direct Memory Access (DMA) attacks. Attacks against the internal state of the SoC are not considered, because those are assumed to be very sophisticated and require expensive equipment. To achieve this security, certain requirements are necessary. First of all, a Device Sealing Key (DSK) needs to be present. This is a symmetric key only known by the device itself and used to protect secrets related to the device. Also needed is a Device Root Key (DRK), which is an asymmetric key pair necessary to identify and authenticate the device. Lastly, the manufacturer's public key needs to be hard coded on the device to make sure it is able to verify the signature on software updates from the manufacturer.

An important aspect of the SecTEE architecture is the method that is applied for memory protection. SecTEE protects enclaves from physical attacks by using a similar approach as the OP-TEE pager. It is an on-demand paging system which runs the entire TEE system on OCM. Whenever a page leaves this OCM, it is encrypted to ensure confidentiality and integrity of the data while it is stored on the Dynamic Random Access Memory (DRAM). Another key feature of SecTEE is its side-channel resistance. Side-channel attacks from the SW are avoided by using a page coloring mechanism. Different enclaves can never share the same cache set, which ensures that one enclave will never be able to evict cache lines of another one. Of course, side-channel attacks can also come from the NW, especially because the NW and SW share the same cache in the ARM TrustZone architecture. In the NW, there are of course certain limitations to what is possible with the cache lines due to privilege restrictions. Nonetheless, the prime and probe method is still relevant in this case. SecTEE cleans and invalidates all cache levels when the Central Processing Unit (CPU) switches from the SW to the NW. This, however, is not sufficient because cross-core side-channel attacks can launch the attack, while a core of the CPU is still executing in the SW. To handle this problem, the cache set of the enclave is locked, which guarantees that the cache of the enclave cannot be probed or manipulated by the NW. Some final aspects that deserve some highlighting are enclave identification, measurement, and RA in SecTEE. Enclaves are published along with the public key of the author and the signed integrity value of the image. With this information the enclave can be verified and identified before it is run. During runtime, the enclave keeps track of important aspects like enclave identifier, integrity, and a flag indicating whether it is privileged. For the RA a specific quoting enclave is created. This is a privileged enclave that can make the system calls related to the management of attestation keys, other enclaves are not allowed to do this. These keys are used to sign the report data of the attestation together with the runtime measurement to provide proof to the verifier that it comes from the correct enclave.

First of all, the number of lines of code are measured because these implementations extend the TEE kernel. This is the TCB and needs to be kept minimal because bugs could easily introduce security vulnerabilities. Next, the overhead of the trusted computing features is identified and discussed. This overhead is acceptable in case

Elliptic Curve Cryptography (ECC) with keys of 256 bits are used, some system calls take too long in case Rivest-Shamir-Adleman (RSA) keys of 2,048 bits are used. The performance of certain enclaves was measured and the xtest benchmark of OP-TEE was executed. SecTEE was about 4 times slower than OP-TEE on the benchmark and 40 times slower than OP-TEE on the enclave execution. It is argued that the memory protection mechanism is the cause of the greatest performance overhead in SecTEE. Finally, to test the effectiveness of the side-channel defense, the effectiveness of the attack on plain OP-TEE is compared to the case of SecTEE. In case of a NW attack, there is a clear difference between prime and probe timings in OP-TEE while both memory operations take an equal amount of time when using SecTEE. This makes it impossible for an attacker to learn anything about the cache behavior of the victim enclave in SecTEE. For the SW attack an Advanced Encryption Standard (AES) key could be recovered after 256,000 encryptions in OP-TEE, while in SecTEE the attacker could not learn any information about the used key.

TrustShadow: Secure Execution of Unmodified Applications with ARM TrustZone

TrustShadow [61] is designed to protect existing applications from untrusted OSs without the need to modify the applications themselves. This is achieved by using TrustShadow as a lightweight runtime system that shields the sensitive information of the application from the OS. This system does not provide system services itself but requests them from the OS. It does, however, guarantee security through encryption, context switching and verifying return values. To ensure that the OS cannot get hold of the encryption keys, sensitive data, and cannot interfere with the security verification, TrustShadow runs in the SW of the ARM TrustZone processor.

The solution described in the paper about TrustShadow is mainly focused on protecting applications from an untrusted OS. This means that the attacker is able to launch DMA attacks, modify interrupted process states, change program execution control flow, hijack system services, and control communication channels [72]. DoS attacks and side-channel attacks such as timing and power analysis are not taken into account in this solution.

The memory of the different stakeholders is isolated from each other to make sure the rich OS is not able to tamper with the memory of the applications and with the runtime system. TrustShadow utilizes the TZASC to make three partitions in memory: one unprotected for the applications, one protected for the applications and one protected for the runtime system itself. The virtual memory space is equally distributed between the OS and the runtime system TrustShadow. The runtime system also maps the physical address space holding the OS to efficiently locate shared memory. Another aspect that needs to be taken into account, is that the OS is used to handle exceptions. When an exception is thrown, the state of the application is stored and the state of the OS is restored. The runtime system then reproduces the exception in this OS state to allow the OS to handle it without the need to get

sensitive details about the application. There are two exceptions that the runtime system implemented in the SW, namely, the random number generation, and floating point operations. Since the cryptography relies on the security of these exceptions they are security critical. System calls are similar to exceptions: the OS needs some information about the application requesting the system call. TrustShadow sanitizes this information to make sure only the necessary data is received by the OS. After receiving the result from the system call, the runtime system also checks this result on correctness before forwarding it to the application. An example of sensitive information that an untrusted OS should not have access to is the memory mapping from virtual to physical address space. This information is stored in the page tables. TrustShadow makes use of the OS page fault handler to update and retrieve the page table entries that are stored in the SW. This ensures that the rich OS does not require the information because it is controlled by the runtime system. After the memory location of a page has been found, this can be loaded for execution. Before this loading is finished, however, the page is checked on its integrity. TrustShadow verifies the executable pages of an application and the executable pages of the shared libraries based on a manifest file related to the application.

TrustShadow was evaluated with a variety of measuring methods. Microbenchmarks using LMBench, file operations using Sysbench, HTTP and HTTPS request handling using Nginx, and finally, the performance overhead on data analytic applications. The performance overhead caused by the runtime system was negligible across all tests. In terms of security, the OS still has a lot of control: it could, for instance, invoke a system call with the original *execve* instead of the new *tz_execve* which largely bypasses TrustShadow. Another concern is the possibility to manipulate or forge manifest files which could allow an attacker to tamper with the code of an application without the runtime system being able to detect it.

TZ-MRAS: A Remote Attestation Scheme for the Mobile Terminal Based on ARM TrustZone

In this paper [73], a TrustZone based Mobile Remote Attestation Scheme (TZ-MRAS) is proposed. There are three main challenges the authors tackle, the first one is building a RoT for measuring, storage, and reporting of the attestation results. The second challenge is to execute binary-based attestation on hardware constrained devices while defending against TOC-TOU attacks. The last challenge consists of the performance optimization of storing measurement logs in a hash tree which has considerable overhead during the construction and updating phases.

The assumptions of the TrustZone model are followed, meaning that the processor is trusted and a CoT is achieved to trusted applications. The SW is the TCB and it provides features to attest the integrity of the NW. The attacker, on the other hand, is assumed to have kernel level access, meaning he can tamper with the REE kernel and applications in any way he desires. DoS and side-channel attacks are not considered in this paper.

To tackle the first challenge related to the root of trust, the functionality of a TPM chip is provided by the TEE of TrustZone. These functionalities are implemented in TA's running in the SW to ensure isolation from the NW. The measurement module keeps track of REE and dynamically detects integrity changes in the kernel, but can also update the Secure Measurement Log (SML). Secondly, the TOC-TOU attacks are protected against by updating the SML with a dynamic update mechanism, namely ProbeIMA. Probes are added to all operations that are able to modify the code pages of the kernel, meaning that any attempt to execute any of these operations will be detected. Lastly, the structure of the SML is optimized by storing the root node in a hardware protected register and the rest of the hash tree in secure memory. Also, the design related to the construction, insertion and revocation algorithms avoids too frequent construction and update operations to increase performance.

The performance overhead in terms of execution time difference, is only touched on shortly. Because this is less than 10% in the worst case, which is very acceptable. Besides the performance overhead, the security analysis is backed up with experiments that show the solution works, and is able to detect the attacks used. The simulated attacks are a rootkit that hijacks the kernel function of the system call table and a TOC-TOU attack that modifies the code pages, and restores them again in between integrity checks. Lastly, the depth of the hash tree is measured for a varying amount of updated nodes, mainly to prove that the construction and updates on those trees have a limited impact on performance.

6.2 Comparison of Approaches

Effectiveness

All solutions focus on measuring the binaries of the executable files to verify the integrity of the runtime system. This is achieved by having a TA in the SW which does the measurements, stores the results, and encrypts them before sending them to the RA server. Running the attestation inside the SW shields the attestation process from being tampered with by the NW. But it does not guarantee that the execution control flow of the attested applications cannot be tampered with. SecTEE seems to keep track of more data for the attestation process, but nothing is said about the integrity of data structures or other critical parts of the system.

Except for the paper on SecTEE, all solutions have a very similar goal and their solutions are also very comparable. All three solutions (TrustShadow, TZ-MRAS and the trusted boot) mainly focus on the attestation of NW processes, but these are not very well shielded or isolated from the rich OS. SecTEE uses RA as one of the many tools to protect the applications or at least identify violations thereof. Due to the effort the authors of SecTEE put into the memory protection scheme, the page coloring mechanism and the way enclaves need to be implemented, it achieves better security guarantees for the enclaves. An enclave has to be implemented as a

TA, which by itself introduces additional security guarantees like, it being infeasible for the rich OS to tamper with the execution control flow of the enclave.

None of the solutions mentioned give much attention to the openness of the secured system. SecTEE, for instance, requires a private public key pair from the manufacturer to function, which entirely closes the solution towards the user and third parties that may also provide useful software. The DRK used in that architecture could serve as the RoT of the device. The other examples do not explicitly mention how the RoT is achieved, only that it is stored in tamper resistant storage hardware. This is, however, a very important detail when validating the openness of the final system.

Assumptions

All solutions assume an IoT device or smartphone device meaning, the solutions need to be adapted to devices with limited hardware specifications. Based on this assumption, all compared solutions chose for ARM SoC with TrustZone capabilities. In this context the SW of ARM TrustZone, along with its TAs, are all assumed to be trusted and secure and isolated from the rich OS. In the case of SecTEE, the additional assumption is made that the enclaves (isolated applications) are written as TAs, which may be feasible for an open platform. Although when these enclaves need to be signed by a manufacturer's private key, it achieves an entirely different goal. On the other hand, the other solutions claim to attest the NW and provide similar protection against the rich OS, which is unfeasible if only the code pages of an application are measured for integrity violations.

SecTEE claims to protect against certain side-channel attacks and hardware attacks like DMA and cold boot attack. These claims seem to be met thanks to the SW being instantiated in a trusted manner with secure boot and the protected code running in the SW. The papers about TrustShadow and TZ-MRAS acknowledge that side-channel attacks are not dealt with, but DMA attacks are prevented by protecting the address space mapping and encrypting the stored data. The reproduced solution does not give specific details about side-channel attacks or DMA attacks, it only mentions hardware, software, and OS attacks. These claims seem excessive, because only the attestation TA itself is protected against these kinds of attacks, due to the TEE and trusted boot. The applications in the NW are still vulnerable to many attacks of which only a few (changing the executable code of the application) are detected by the solution. Last but not least, OS attacks cannot be prevented using this method. An attack can in some cases (depending on what is modified) be detected with RA, but none of the solutions make an effort to make attacking the OS harder. This is due to it not being considered when discussing the applications running in the NW.

6.3 Reflection

First of all, a consideration needs to be made with respect to the RoT of the system. The discussed related work does not take into account the openness of the system, and thus does not discuss this in detail. The PinePhone is equipped with tamper resistant storage hardware, which could accommodate the RoT. This detail on what exactly is stored in this memory and how this information is generated, has a large impact on whether the system remains open. If the user is allowed to put a public key in the secure storage, of which he and only he possesses the secure counterpart, the solution remains open. The security, of course, depends on how well the user is able to protect his private key, but this is out of scope for this work. This is, however, how we believe a RoT can be achieved for an open system. In case the device is sold second-hand, there is of course a problem, because the buyer needs to trust the first owner in order for this deal to work. Since the original user possesses the private key, and it is hard to prove that when the buyer receives this key, the initial owner does not have access to this key anymore. This is interesting, but goes beyond realizing a secure RoT for an open platform.

Our solution avoids the trusted third party that needs to be convinced that the integrity of the software running on the device is not violated. The only party that needs to be assured about the integrity of the system is the user. Based on our implementation we can claim that the NW process integrity checking works. To become a complete security solution, certain extensions will need to be made which are discussed in the future work. Taking these considerations into account, we believe that user-controlled attestation can be used to increase the security of an open platform without disrupting the openness of the system.

The evaluation gives insight in how well this solution meets the expectations of the smartphone user. The performance is not ideal as elaborately discussed earlier, but this was not the main goal of the proof of concept. Making sure the solution meets the expectations of the user is one thing, but the software providers need to be confident about their guarantees as well. Right now the user decides which software can run on the device. By doing so he is assumed to trust the software provider. A software provider is only guaranteed that, other software running on the platform comes from providers that are trusted by the user. This guarantee is not very strong, since there is a possibility that the user trusts a malicious provider which could try to tamper with the processes of other providers. If software providers need additional guarantees, the attestation solution should be extended to provide these security measures, some examples of these measures are given in the future work. Even though guaranteeing the software providers correct execution control flow of their processes seems evident, it does have implications on the user's freedom to operate the device. This leans more towards the philosophical side of the open platform idea so we will not dive in too deep about this.

The solution in this thesis differs from [1], in the sense that the initial measurement results are stored in the secure memory of the SW instead of at a RA server. Storing these values on the device itself, allows for the solution to be extended, and allow the user to attest the device. Depending on how the user reacts to this information, similar security guarantees can be provided as the closed systems. All related work that we have discussed, relies on the integrity and confidentiality of the secure memory of ARM TrustZone. Furthermore, the execution control flow of the TAs is always assumed to be protected against adversaries whether they attack with software attacks, OS attacks, or certain hardware attacks. The solution presented in the implementation also relies on these features to store the initial hash digests and perform the comparisons with the newly measured values, respectively. Due to these assumptions already being present they do not weaken the security of the proposed system compared to the closed systems.

6.4 Future Work

Considerations

One important sensitivity point is to decide whether every page in an executable memory region gets its own hash value or whether all the pages in this memory region get one combined hash value. The fine-grained option allows to only measure the code pages that are currently loaded into the RAM. This is sufficient, because the executable page can only start being executed when it is located in RAM. If, on the other hand, all pages of the region need to be measured, the unused pages also need to be loaded into RAM to be able to measure them. The downside to the fine-grained option is that more initial values need to be stored, since every page has its own value.

Another trade-off that could be put forth, is the hashing algorithm that is used to measure the memory pages. The hashing algorithm needs to adhere to certain requirements like avoiding collisions, but a Hash-based Message Authentication Code (HMAC) algorithm that uses a key during the hashing process, is definitely excessive. The hashing mainly needs to be fast, because lots of memory pages need to be attested and this needs to happen relatively often to guarantee improved security. The hash digests do not need to be secure in the sense of being unforgeable. They are stored in secure memory which cannot be tampered with, meaning the attacker is not able to replace the initial values with a malicious one. With the initial values assumed secure, the hash digests are merely used to detect whether the hashed information has changed or not, for which a normal hashing algorithm is sufficient due to its collision avoidance.

Lastly, it was found out that in a recent framework update of OP-TEE an attestation PTA was added to the kernel [74]. The PTA in that solution attests the SW processes (TAs), which is different from the implementation presented in this

thesis, since we focus on checking the integrity of the processes in the NW. This also implies that the added PTA in the kernel provides asymmetric key encryption to communicate with a remote verifier in a secure manner. The update notes clearly state that RA was intended, so these differences are definitely in line with the expectations. Even though there are certain differences, it is definitely possible to change the implementation 'slightly' to make it fit into this provided attestation PTA and reuse some of its already present functionality. This could be the beginning of a more standardized form of NW integrity checks if the shortcomings have been dealt with.

Shortcomings

In the proposed solution, there is still a strong dependency upon the rich OS. This dependency mainly comes from the files that are used to determine the physical addresses of pages related to the processes that are measured. These files are part of the data structures the rich OS owns. A solution to this problem could be to move the responsibility of the memory mapping into the SW. The downside to this is that it increases the TCB, because the SW is often seen as the TCB. This can however be achieved in a relatively lightweight fashion, as demonstrated in the TrustShadow paper where the page table is stored inside the SW, and the SW handles page faults that are generated in the rich OS.

Partial attestation introduces a fake sense of security, because not all possible attacks are checked. We have mentioned that the execution control flow of an application can be tampered with through the manipulation of data structures. There are lots of ways similar attacks can be achieved to make applications behave differently than intended. Some solutions protect better against some of these attacks than others, like checking the results of a system call executed by the rich (untrusted) OS. [20] describes a variety of possible aspects for which integrity checks are useful to increase the level of trust one can have in an application. These properties vary from OS data structures, available resources to event occurrence and timing. This is probably harder to achieve when trying to protect against a rich OS like Linux compared to the Contiki OS that was used, but it looks like a promising direction for future work.

The attestation measurements are performed, software integrity checks in our case, but there is no way to interact with this information yet. To allow the user to perform user-controlled attestation, he needs to be informed about the results of these measurements. To make sure this is achieved in a secure manner, the trusted I/O channels in ARM TrustZone can be used. Through these channels the user could also command the system to take certain actions with respect to the processes that have been tampered with. By doing this the user is in control with respect to how software integrity violations are handled.

Finally, if software providers need to be guaranteed that the execution control flow of their programs is not tampered with, an extension is necessary. For instance, the ability to identify applications violating security measures like trying to tamper with other applications or the rich OS. The user should be notified about this and be strongly advised to remove those malicious applications from his device. A solution could be opted for where the process is terminated if any inconsistencies are detected. This would increase the guarantees that can be provided with respect to the proper execution of programs from certain software providers. It would of course be better if it were infeasible to tamper with another process, but we believe this is not achievable with user-controlled attestation or any form of attestation alone.

Chapter 7

Conclusion

The goal of this thesis is to come up with security measures that can protect an open platform. The use case of a smartphone is specifically looked at, because currently the large companies that produce these devices, disrupt the openness of the system while implementing security mechanisms. Thanks to the introduction of the PinePhone, a hardware platform with the necessary security features and openness has become available. Using this hardware, it is reviewed what RoT could be used. This proves to be a sensitivity point towards the openness of the system. Furthermore, attestation is investigated as a security system for the platform, taking into account the user expectations. Finally, the achieved security guarantees are evaluated to support our claims. Reviewing the RoT is attempted by implementing secure boot on the PinePhone, but turned out to be more challenging than expected. Therefore this is pushed back to the discussion. There the nuances of what needs to be taken into account to keep the system open, are elaborated. The attestation on the other hand, is implemented based on existing work: the solution is modified to fit better into the use case. Evaluating the user expectations is touched on during the experiments by giving a performance and security analysis. The achieved security is compared with related solutions that do not take into account the openness of the system. The implementation of the attestation focuses on measuring the executable memory pages that are present in RAM and in use by running processes. The details of this process are elaborately explained in the implementation. The experiments show that although the measuring and thus checking the integrity works, it performs very slowly compared to the work on which this implementation is based. More effort needs to be put into optimizing the code to achieve results that are feasible. Based on the comparison with closed system, it can be stated that the proposed solution provides similar security guarantees, given that the RoT was instantiated in a secure manner.

Bibliography

[1] Z. Ling, H. Yan, X. Shao, J. Luo, Y. Xu, B. Pearson, and X. Fu, "Secure boot, trusted boot and remote attestation for arm trustzone-based iot nodes," *Journal of systems architecture*, vol. 119, p. 102240, 2021.

[2] S. O'Dea, "Number of smartphone subscriptions worldwide from 2016 to 2027," 2022. Last accessed on 30th of May 2022.

[3] A. O'Neill, "World Total population from 2010 to 2020," 2022. Last accessed on 30th of May 2022.

[4] statista, "Mobile internet usage worldwide," 2022. Last accessed on 30th of May 2022.

[5] M. Litoussi, N. Kannouf, K. El Makkaoui, A. Ezzati, and M. Fartitchou, "Iot security: challenges and countermeasures," in *Procedia Computer Science*, vol. 177, pp. 503–508, Elsevier B.V, 2020.

[6] Samsung, "KNOX stay connected, protected, and productive," 2022. Last accessed on 26th of May 2022.

[7] PINE64, "PINE64 pinephone," 2021. Last accessed on 26th of May 2022.

[8] OP-TEE, "OP-TEE about op-tee," 2022. Last accessed on 26th of May 2022.

[9] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn, "Design and implementation of a tcg-based integrity measurement architecture.," pp. 223–238, 01 2004.

[10] J. Duan, G. Cai, K. Xu, and S. Guo, "Integrity measurement based on tee virtualization architecture," in *2020 5th International Conference on Mechanical, Control and Computer Engineering (ICMCCE)*, pp. 370–376, IEEE, 2020.

[11] M. Kucab, P. Borylo, and P. Cholda, "Remote attestation and integrity measurements with intel sgx for virtual machines," *Computers & security*, vol. 106, p. 102300, 2021.

[12] M. Alam, T. Ali, S. Khan, S. Khan, M. Ali, M. Nauman, A. Hayat, M. Khurram Khan, and K. Alghathbar, "Analysis of existing remote attestation techniques," *Security and communication networks*, vol. 5, no. 9, pp. 1062–1082, 2012.

[13] S. F. J. J. Ankergård, E. Dushku, and N. Dragoni, "State-of-the-art software-based remote attestation: Opportunities and open issues for internet of things," *Sensors (Basel, Switzerland)*, vol. 21, no. 5, pp. 1–23, 2021.

[14] D. Spinellis, "Reflection as a mechanism for software integrity verification," *ACM transactions on information and system security*, vol. 3, no. 1, pp. 51–62, 2000.

[15] L. Gu, X. Ding, R. Deng, B. Xie, and H. Mei, "Remote attestation on program execution," in *Proceedings of the 3rd ACM workshop on scalable trusted computing*, STC '08, pp. 11–20, ACM, 2008.

[16] Y. Qin, J. Liu, S. Zhao, D. Feng, and W. Feng, "Ripte: Runtime integrity protection based on trusted execution for iot device," *Security and communication networks*, vol. 2020, 2020.

[17] T. Ali, R. Ismail, S. Musa, M. Nauman, and S. Khan, "Design and implementation of an attestation protocol for measured dynamic behavior," *The Journal of supercomputing*, vol. 74, no. 11, pp. 5746–5773, 2017.

[18] H. Ba, H. Zhou, J. Ren, and Z. Wang, "Runtime measurement architecture for bytecode integrity in jvm-based cloud," in *2017 IEEE 36th Symposium on Reliable Distributed Systems (SRDS)*, vol. 2017-, pp. 262–263, IEEE, 2017.

[19] M. Alam, X. Zhang, M. Nauman, T. Ali, and J.-P. Seifert, "Model-based behavioral attestation," in *Proceedings of ACM Symposium on Access Control Models and Technologies, SACMAT*, SACMAT '08, pp. 175–184, ACM, 2008.

[20] J. T. Mühlberg, J. Noorman, and F. Piessens, "Lightweight and flexible trust assessment modules for the internet of things," in *Computer Security – ESORICS 2015*, vol. 9326 of *Lecture Notes in Computer Science*, pp. 503–520, Cham: Springer International Publishing, 2016.

[21] Trusted Computing Group, "Tpm mobile with trusted execution environment for comprehensive mobile device security," 2012. Last accessed on 28th of May 2022.

[22] Confidential Computing Consortium, "Confidential computing: Hardware-based trusted execution for applications and data," 2021. Last accessed on 28th of May 2022.

[23] M. Sabt, M. Achemlal, and A. Bouabdallah, "Trusted execution environment: What it is, and what it is not," in *2015 IEEE Trustcom/BigDataSE/ISPA*, vol. 1, pp. 57–64, IEEE, 2015.

[24] T. MATSUMOTO, M. IKEDA, M. NAGATA, and Y. UEMURA, "Secure cryptographic unit as root-of-trust for iot era," *IEICE transactions on electronics*, vol. E104.C, no. 7, pp. 262–271, 2021.

[25] G. Fotiadis, J. Moreira, T. Giannetsos, L. Chen, P. B. Rø nne, M. D. Ryan, and P. Y. A. Ryan, "Root-of-trust abstractions for symbolic analysis: Application to attestation protocols," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 13075 of *Lecture Notes in Computer Science*, pp. 163–184, Cham: Springer International Publishing, 2021.

[26] S. Zhao, J. Lin, W. Li, and B. Qi, "Research on root of trust for embedded devices based on on-chip memory," in *2021 International Conference on Computer Engineering and Application (ICCEA)*, (Piscataway), pp. 501–505, IEEE, 2021.

[27] GlobalPlatform, "Tee client api specification," 2010. Last accessed on 28th of May 2022.

[28] J. Van Bulck, D. Oswald, E. Marin, A. Aldoseri, F. Garcia, and F. Piessens, "A tale of two worlds: Assessing the vulnerability of enclave shielding runtimes," in *Proceedings of the 2019 ACM SIGSAC Conference on computer and communications security*, CCS '19, pp. 1741–1758, ACM, 2019.

[29] P. Guo, Y. Yan, C. Zhu, and J. Wang, "Research on arm trustzone and understanding the security vulnerability in its cache architecture," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 12382 of *Lecture Notes in Computer Science*, pp. 200–213, Cham: Springer International Publishing, 2021.

[30] A. Machiry, E. Gustafson, C. Spensky, C. Salls, N. Stephens, R. Wang, A. Bianchi, Y. R. Choe, C. Krügel, and G. Vigna, "Boomerang: Exploiting the semantic gap in trusted execution environments," in *NDSS*, ndss symposium, 2017.

[31] Common Criteria, "Security assurance components," 2017. Last accessed on 28th of May 2022.

[32] ARM Ltd., "TrustZone trustzone for cortex-a," 2022. Last accessed on 26th of May 2022.

[33] S. Pinto and N. Santos, "Demystifying arm trustzone: A comprehensive survey," *ACM computing surveys*, vol. 51, no. 6, pp. 1–36, 2019.

[34] M. Lentz, R. Sen, P. Druschel, and B. Bhattacharjee, "Secloak: Arm trustzone-based mobile peripheral control," in *MobiSys 2018 - Proceedings of the 16th ACM International Conference on Mobile Systems, Applications, and Services*, MobiSys '18, pp. 1–13, ACM, 2018.

[35] S. Eskandarian, J. Cogan, S. Birnbaum, P. C. W. Brandon, D. Franke, F. Fraser, G. GarciaJr, E. Gong, H. T. Nguyen, T. K. Sethi, V. Subbiah, M. Backes, G. Pellegrino, and D. Boneh, "Fidelius: Protecting user secrets from compromised browsers," 2018.

[36] R. Chang, L. Jiang, W. Chen, Y. Xiang, Y. Cheng, and A. Alelaiwi, "Mipe: a practical memory integrity protection method in a trusted execution environment," *Cluster computing*, vol. 20, no. 2, pp. 1075–1087, 2017.

[37] A. Ferraiuolo, A. Baumann, C. Hawblitzel, and B. Parno, "Komodo: Using verification to disentangle secure-enclave hardware from software," in *Proceedings of the 26th Symposium on operating systems principles*, SOSP '17, pp. 287–305, ACM, 2017.

[38] J. Amacher and V. Schiavoni, "On the performance of arm trustzone: (practical experience report)," in *Lecture Notes in Computer Science*, vol. 11534 of *Distributed Applications and Interoperable Systems*, pp. 133–151, Springer International Publishing, 2019.

[39] Z. Hua, Y. Yu, J. Gu, Y. Xia, H. Chen, and B. Zang, "Tz-container: protecting container from untrusted os with arm trustzone," *Science China. Information sciences*, vol. 64, no. 9, 2021.

[40] Google, "Android keystore system," 2022. Last accessed on 26th of May 2022.

[41] PINE64, "PinePhone software releases," 2022. Last accessed on 26th of May 2022.

[42] STMicroelectronics, "run time sequences in op-tee," 2022. Last accessed on 28th of May 2022.

[43] GlobalPlatform, "Tee internal core api specification," 2016. Last accessed on 28th of May 2022.

[44] K. M. Rajasekharaiah, C. S. Dule, and E. Sudarshan, "Cyber security challenges and its emerging trends on latest technologies," *IOP Conference Series: Materials Science and Engineering*, vol. 981, no. 2, p. 22062, 2020.

[45] Y. Lakhdhar, S. Rekhis, and N. Boudriga, "Proactive security for safety and sustainability of mission critical systems," *IEEE transactions on sustainable computing*, vol. 6, no. 2, pp. 257–273, 2021.

[46] A. R. Javed, M. O. Beg, M. Asim, T. Baker, and A. H. Al-Bayatti, "Alphalogger: detecting motion-based side-channel attack using smartphone keystrokes," *Journal of ambient intelligence and humanized computing*, 2020.

[47] W. Song, M. Jiang, H. Yan, Y. Xiang, Y. Chen, Y. Luo, K. He, and G. Peng, "Android data-clone attack via operating system customization," *IEEE access*, vol. 8, pp. 199733–199746, 2020.

[48] R. Setyawan, A. A. Rahayu, K. F. Nur Annisa, and A. Amiruddin, "A brief review of attacks and mitigations on smartphone infrastructure," *IOP Conference Series: Materials Science and Engineering*, vol. 852, no. 1, p. 12141, 2020.

[49] J. GRAVELLIER, "Remote hardware attacks on connected devices," 2021.

[50] S. Kumar, L. J. Kittur, and A. R. Pais, "Attacks on android-based smartphones and impact of vendor customization on android os security," in *Information Systems Security*, vol. 12553 of *Lecture Notes in Computer Science*, pp. 241–252, Cham: Springer International Publishing, 2020.

[51] mquaresma, "Setting up OP-TEE and Linux for the Pine A64," 2020. Last accessed on 30th of May 2022.

[52] Linux community, "Mobian sunxi64-linux," 2022. Last accessed on 29th of May 2022.

[53] OP-TEE, "OP-TEE optee_os, optee_client," 2022. Last accessed on 29th of May 2022.

[54] DENX Software Engineering, "Pine64 u-boot," 2022. Last accessed on 29th of May 2022.

[55] ARM-Software, "ARM Software arm trusted firmware," 2022. Last accessed on 29th of May 2022.

[56] OP-TEE, "OP-TEE qemu v8," 2022. Last accessed on 29th of May 2022.

[57] M. Gross, N. Jacob, A. Zankl, and G. Sigl, "Breaking trustzone memory isolation and secure boot through malicious hardware on a modern fpga-soc," *Journal of cryptographic engineering*, 2021.

[58] K. Ryan, "Hardware-backed heist: Extracting ecdsa keys from qualcomm's trustzone," in *Proceedings of the 2019 ACM SIGSAC Conference on computer and communications security*, CCS '19, pp. 181–194, ACM, 2019.

[59] S. K. Bukasa, R. Lashermes, H. Le Bouder, J.-L. Lanet, and A. Legay, "How trustzone could be bypassed: Side-channel attacks on a modern system-on-chip," in *Information Security Theory and Practice*, vol. 10741 of *Lecture Notes in Computer Science*, (Cham), pp. 93–109, Springer International Publishing, 2018.

[60] A. S. Salman and W. K. Du, "Securing mobile systems gps and camera functions using trustzone framework," in *Intelligent Computing*, vol. 285 of *Lecture Notes in Networks and Systems*, pp. 868–884, Cham: Springer International Publishing, 2021.

[61] L. Guan, P. Liu, X. Xing, X. Ge, S. Zhang, M. Yu, and T. Jaeger, "Trustshadow: Secure execution of unmodified applications with arm trustzone," in *MobiSys 2017 - Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '17, pp. 488–501, ACM, 2017.

[62] D. Zhang and S. You, "iflask: Isolate flask security system from dangerous execution environment by using arm trustzone," *Future generation computer systems*, vol. 109, pp. 531–537, 2020.

[63] T. Kawada, S. Honda, Y. Matsubara, and H. Takada, "Tzmcfi: Rtos-aware control-flow integrity using trustzone for armv8-m," *International journal of parallel programming*, vol. 49, no. 2, pp. 216–236, 2020.

[64] R. Ben Yehuda and N. J. Zaidenberg, "Protection against reverse engineering in arm," *International journal of information security*, vol. 19, no. 1, pp. 39–51, 2019.

[65] D. Durst, "The Endless Conundrum of creating a secure PinePhone," 2021. Last accessed on 30th of May 2022.

[66] cirosantilli, "common_userland.h," 2022. Last accessed on 16th of May 2022.

[67] W. Li, M. Ma, J. Han, Y. Xia, B. Zang, C.-K. Chu, and T. Li, "Building trusted path on untrusted device drivers for mobile devices," in *Proceedings of 5th Asia-Pacific Workshop on Systems, APSYS 2014*, APSys '14, pp. 1–7, ACM, 2014.

[68] O. Safaryan, L. Cherckesova, N. Lyashenko, P. Razumov, V. Chumakov, B. Akishin, and A. Lobodenko, "Modern hash collision cyberattacks and methods of their detection and neutralization," *Journal of Physics: Conference Series*, vol. 2131, no. 2, p. 22099, 2021.

[69] S. Zhao, Q. Zhang, Y. Qin, W. Feng, and D. Feng, "Sectee: A software-based approach to secure enclave architecture using tee," in *Proceedings of the 2019 ACM SIGSAC Conference on computer and communications security*, CCS '19, pp. 1723–1740, ACM, 2019.

[70] M. Huber, B. Taubmann, S. Wessel, H. P. Reiser, and G. Sigl, "A flexible framework for mobile device forensics based on cold boot attacks," *EURASIP Journal on Multimedia and Information Security*, vol. 2016, no. 1, pp. 1–13, 2016.

[71] ncc group, "Tpm genie: Interposer attacks against the trusted platform module serial bus," 2018. Last accessed on 25th of May 2022.

[72] C. Qiang, W. Liu, L. Wang, and R. Yu, "Controlled channel attack detection based on hardware virtualization," in *Algorithms and Architectures for Parallel Processing*, vol. 11334 of *Lecture Notes in Computer Science*, pp. 406–420, Cham: Springer International Publishing, 2018.

[73] Z. Wang, Y. Zhuang, and Z. Yan, "Tz-mras: A remote attestation scheme for the mobile terminal based on arm trustzone," *Security and communication networks*, vol. 2020, 2020.

[74] D. Harbin, "Trusted Firmware OP TEE v3.17.0 release," 2022. Last accessed on 16th of May 2022.