CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Verified boot in embedded systems with hard boot time constraints

Master's thesis in Computer Systems and Networks

Christos Profentzas
Mirac Günes

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2017

Master's thesis 2017

Verified boot in embedded systems with hard boot time constraints

Mirac Günes
Christos Profentzas

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2017

© Mirac Günes, Christos Profentzas 2017.

Supervisor: Yiannis Nikolakopoulos, Postdoctoral Researcher
Examiner: Assoc. Prof. Marina Papatriantafilou

Master's Thesis 2017
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2017

Abstract

The use of embedded systems in devices, machines, and vehicles which we interact with every day is increasing progressively. In the vehicular industry, we can see such embedded systems in the form of Electronic Control Units (ECUs) which have specific tasks in different areas of the vehicle. In order to coordinate the various embedded systems, usually, a central ECU plays the role of a hub and is equipped with an operating system. Moreover, the central ECU has expanded nowadays to have internet connectivity which raises certain security issues.

In this setting, we need to assure the integrity of the operating system against any malicious modifications. According to our threat analysis, this can be done only during the boot-up process; unfortunately, the verification process is time-consuming. Therefore, the verification process poses a serious performance issue since the ECU has certain real-time constraints.

In this thesis, we investigate most of the serious threats regarding the Operating System integrity and the boot process as well. Furthermore, we evaluate the state of the art techniques for a verified boot process for a Linux Kernel system. The experiment setup includes general purpose embedded devices with real-time constraints in mind.

After the evaluation, we conclude that we cannot implement an adequately secure solution on an inherently non-secure hardware platform; a compromise on security is necessary to meet the real-time constraints. Therefore, we propose that we should consider the security aspect during the design phase of an embedded platform.

Keywords: Verified boot, Secure boot, Embedded Linux systems, Computer Security, Operating systems, Real-time systems.

This page intentionally left black

Acknowledgements

I would like to thank my thesis partner Christos Profentzas, our academic supervisor Postdoctoral researcher Ioannis Nikolakopoulos, our examiner Assoc. Prof. Marina Papatriantafilou, our industrial supervisors from Volvo GTT OBT, Henrik Wärmlind and Hedvig Jonsson and our project manager from Volvo GTT OBT, Rolf Nilsson.

Finally, I have to express my very profound gratitude to my parents for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without them. Thank you.

Mirac Günes, Gothenburg, June 2017

I would like to thank my thesis partner Mirac Günes for the continuous collaboration during the project. Regarding our academic supervision, I would like to thank our supervisor Postdoctoral researcher Ioannis Nikolakopoulos and our examiner Assoc. Prof. Marina Papatriantafilou for the helpful comments and remarks. Regarding our industrial collaboration, I would like to thank our mentors in Volvo GTT OBT, Henrik Wärmlind, and Hedvig Jonsson, for the guidance and ideas that they provide into the thesis. Also, I would like to thank our project manager in Volvo GTT OBT, Rolf Nilsson for the support and permanent willingness to help.

Finally, I wish to express my gratitude to all computer scientists that have made this area so excited, and they continue to aspire us to extend the human knowledge even further.

Christos Profentzas, Gothenburg, June 2017

This page intentionally left black

Contents

List of Figures

List of Tables

1
Introduction

This thesis is an industrial Master's thesis performed in collaboration between Volvo Group Trucks Technology - Vehicle Engineering - On-Board Telematics department and Chalmers University of Technology.

1.1 Embedded Systems in Vehicular Systems

Embedded systems are making life easier for the human being by adding functionality, precision, and security to devices and vehicles. Hence, the integration of such systems in devices which we interact with every day is increasing rapidly. A specific type of these systems in the vehicular industry is called the Electronic Control Unit (ECU).

As these devices add functionality and precision/optimization to the vehicle, they also start to carry more responsibility. To coordinate the various ECUs in a vehicle, normally a central ECU operates as a hub and is equipped with an operating system. Moreover, the central ECU has expanded nowadays to have internet connectivity which raises certain security issues.

Since ECUs (which used to work in isolation) started to connect to the Internet over the recent years, we need to make a new threat analysis. Furthermore, the security of ECUs increases in importance as they carry vital responsibility as a safety-critical system. In this way, security researchers just started to analyze and categorize the possible threats that ECUs and similar systems can be affected with.

However, security is not the only open research area in this context. This is due to the fact that the vehicle has to be fully operational within a short period of time; as soon as the key is turned on - with a working actuator in the brake system, optimal air-fuel mixture in the engine and further essential operational functionality. This time limitation requires real-time aspects to be taken into consideration. Hence, an ECU is considered as a real-time and safety-critical system.

1.2 Problem Statement

Even though researchers have extensively worked in the security and real-time areas for the boot process of embedded systems, they have mostly worked separately and evaluated their techniques in isolation. So, there is an open scientific problem

on investigating time overheads caused by security measures in the boot process. Further, if the time overhead is violating the real-time constraint, there is a need to offset the security measures. In other words, we need to adjust the optimal balance between security and time constraints, for the boot process in embedded systems.

There are a few more aspects of the problem which needs to be considered by the research community; one aspect is the need of analyzing the possible threats to the boot process, before deciding on the necessary security measures.

There exists a wide variety of techniques where we implement security to the boot process. So, another aspect is to analyze suitable techniques which prevent/detects possible threats from affecting the boot process.

The final aspect is to investigate optimization possibilities for the system and suitable threat prevention/detection techniques.

1.3 Thesis Contributions

The thesis manages to bring together the aforementioned aspects by providing certain level of security measures in the boot process. To preserve a generic scope, we have not specified a certain boot-up time constraint but instead optimized and evaluated the time overhead impact on the boot-up time.

We start our thesis by trying to identify the main threats regarding the integrity of the operating system. Our analysis shows that we cannot detect most of the modification techniques after the boot-up of the Operating System. Thus, our analysis justifies the need for security measures during the boot process. In this way, the thesis can serve as a reference to the most common malicious modification in the Operating System. The threat model and analysis that we present can also serve as a security guideline for any embedded system that increases its functionality with an operating system.

Further, into the thesis, we define the concept of verified boot and evaluate the state of the art techniques used to verify the different stages of the boot process. Even though different platforms present most of the techniques with different methods, we can draw a common generic conclusion which helps the process of designing a Verified boot feature to an arbitrary system.

To satisfy the final aspect mentioned in Section 1.2, we evaluate different optimization techniques in terms of hardware- and software optimization for different parts of the system. The applied methods appear to have an uttermost significance to the boot-up time, which shows the importance of optimizing systems.

We do also contribute with two proofs of concept of selected hardware- and software based techniques to reveal the impacts of the Verified boot implementations, which

can have an influence in future security and real-time research. The two implemented proofs of concept techniques we selected are *Verified U-Boot* and *Verified Boot with Trusted Platform Module (TPM)*. We described the techniques in Chapter 5, and in sub-chapter 7.2 we justify the reasons for selecting them.

1.4 Structure of the Thesis

Our thesis has the following structure: In Chapter 2, we provide the basic knowledge to understand malicious modifications in the Operating System. Chapters 3 & 4 are focusing on the threat model and the range of malicious modifications. The thesis continues with Chapter 5, where we define the generic terminology of the Verified boot. We explain various Verified boot concepts, and we present various state of the art security techniques. In Chapter 6, we give an overview of boot-time optimization techniques. In Chapter 7, we present the evaluation of selected techniques. It includes the specification of the hardware, the results of the experiments and a discussion of the results. In Chapter 8, we mention some related work in this area. Finally, Chapter 9 contains the conclusion and some ideas for future work.

2
Background

Most embedded systems are using Linux as the Operating System. Due to the fact that it is an open source project which everyone can use and extend. Moreover, Linux is quite popular with a large community that supports it. Also, most of the embedded systems designed with an ARM processor supported mainly by the Linux community. In our thesis, we are focusing on securing the boot process to maintain a trusted and authenticated Operating System. In this chapter, we introduce some basic knowledge about Linux structures.

2.1 Linux kernel functionality

The main reason not to trust the Operating System is malicious modifications of the Kernel, which is the core of the Operating System. In order to understand the details of the threat, we introduce some preliminary knowledge about Kernel's functionality.

2.1.1 Introduction to Loadable Kernel Module(LKM)

There are (roughly) two types of Operating Systems (OS). One is monolithic [6], where we load the whole code as a single program (one memory space). The other is micro-kernel [6], where we divide the operating system into smaller independent programs (separate memory spaces). We can categorize Linux[1] as monolithic OS, and more details are provided here [6].

Nevertheless, new modules can be loaded up during run-time. One mechanism that Linux provides is the insmod program [7]. This program simply appends the given object file to already up-running kernel code.

Typical modules which usually are added:

- device drivers
- file-system drivers
- system calls

[1]Over the past years, Linux has become an ambiguous term. Many vendors use the term Linux for their own distributions, but what they provide is their own extension modules(LKMs). They extend LKMs on top of the original Linux-kernel wrapped into a single package. In the following text, we use Linux to mean the Linux-kernel, which is a single program, currently in version 4.9.8.

Those modules are considered part of the kernel, after being loaded. Thus, LKM is a common place for attacks. Likewise, the attacker will try to add malicious code either by using/replacing existing LKM or adding new ones.

2.1.2 Introduction to System Calls

For protection and security reasons the program execution of the system is divided into privilege and normal mode. In Intel x86 architecture this is supported by multiple levels called rings [8]; the figure 2.1 depicts the different levels of execution.

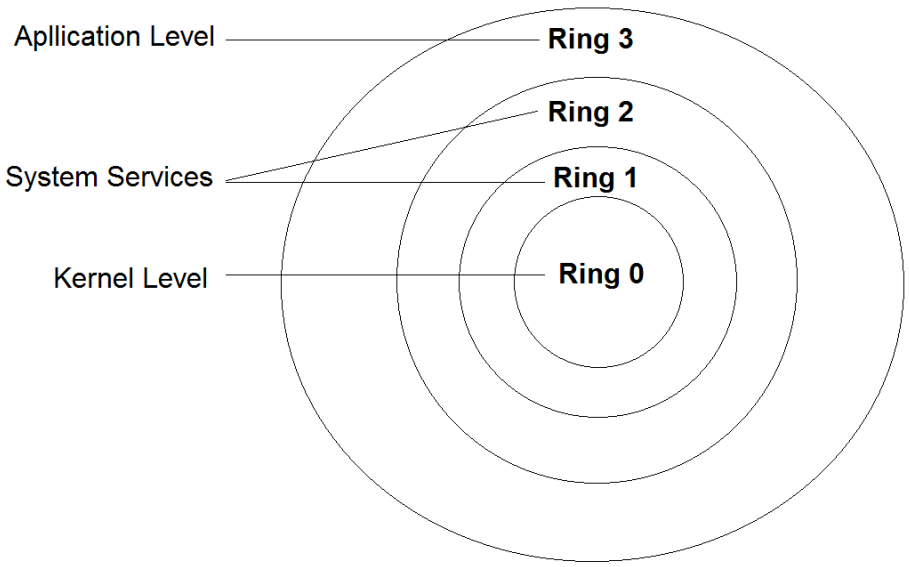

Figure 2.1: Ring levels in x86 architecture

However, Linux is using only ring 0 and ring 3 for user-mode and kernel-mode respectively. On the other hand, in ARM architecture (v4 and above) [9] there are seven operating modes which are presented in table 2.1.

Table 2.1: Operating Modes in ARM architecture

Mode	Mode identifier	
Undefined	und	Privileged modes
Fast interrupt	fiq	
Interrupt	irq	
Supervisor	svc	
Abort	abt	
System	sys	
User	user	Unprivileged mode

Regular programs will run in the user mode. To switch into the kernel mode a software interrupt should be initiated. This can be done by using the relevant *swi* command. The details are architecture specific and more details can be found in the reference manual [9].

In this way, when a user program wants to use protected resources out of its memory space (such as a network card) a system call to the kernel is being made. A system call [10] usually invokes the corresponding Kernel module code which has been explained above.

The common case is that system programmers do not directly invoke system calls (which would have been quite tedious), instead they call the standardized library wrapper function [10], which hide the details as depicted in figure 2.2.

Thus, a system call is essentially a software (synchronized) interrupt. In order to call the right handler the operating system maintains a table called **syscall**. Consequently, the syscall table and the handler subroutine are possible targets for attack. The different techniques that an attacker can exploit are described in a latter chapter.

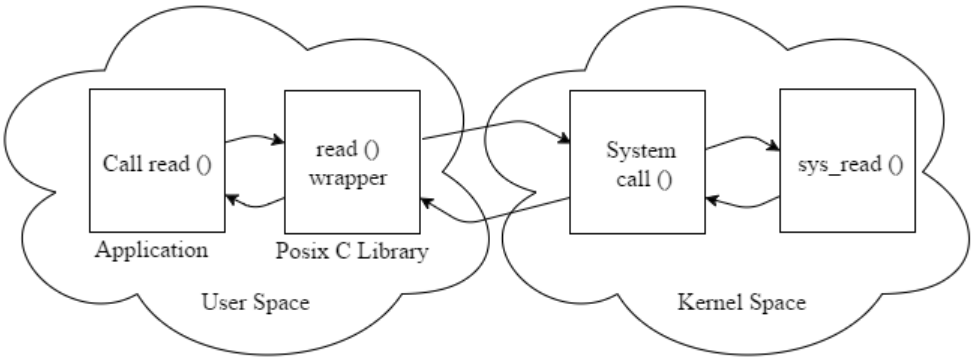

Figure 2.2: System call architecture in Linux

2.1.3 Introduction to Interrupts

Besides software system calls the operating system can be invoked to handle hardware interrupts [11]. Hardware interrupts are triggered (asynchronously) by hardware devices to notify the CPU of a current event. The way the CPU is handling the interrupts is to invoke the appropriate code of the kernel to handle the corresponding event. Figure 2.3 demonstrates an example of hardware interrupt.

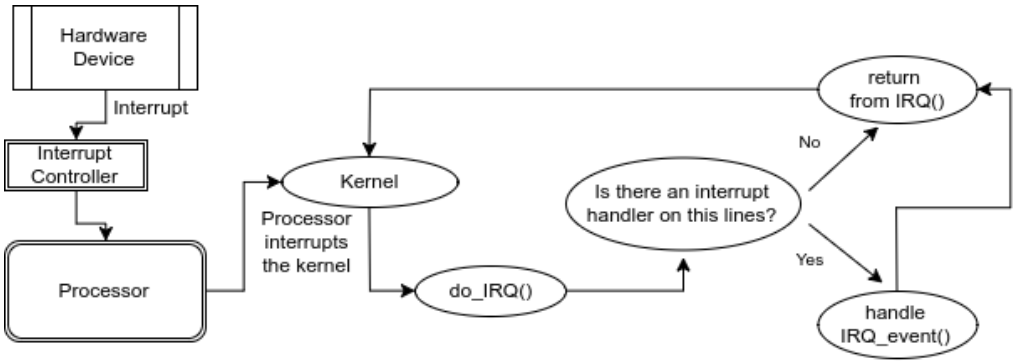

Figure 2.3: An example of hardware interrupt

In the x86 architecture, the kernel maintains a table called Interrupt Descriptor Table [8], which is created during the start-up of the system. On the other hand, in ARM architecture the table is called Exception Vector Table [9].

The above tables are set-up in a well-known and predefined memory address [9] [8]. Thus, it is easy to find the address of tables, and since the address is writable[2] and executable, thereby a common target for the attackers.

[2]It could be possible to flag this area as not writable, the prerequisite is to equip the processor with a Memory Management Unit (MMU)

3
Threat Model

In order to decide the type of security measures, which will be able to maintain a trusted and genuine Operating System, we need to make a threat analysis regarding the malicious modifications of the Kernel. In this chapter, we identify the main threat and argue why it poses a serious risk for the integrity of the Kernel.

3.1 Introduction to Rootkits

we can define Rootkit as a set of tools and coding techniques which mainly focus to maintain administrative privileges (root access) to a compromised system [12]. We provide information about how an attacker can get root access in Appendix A. From now on we will assume that an attacker has used an exploitation and successfully gained root access to the system.

Some researchers divide the Rootkits into two categories: Kernel-level and User-level [12]. In this thesis, we limited the scope to only kernel level Rootkits as stated by the above assumption.

Rootkits try to insert malicious code into the existing modules [12] of the Kernel or try to subvert part of the existing functionality [12]. Moreover, the attacker will try to hide the existence of the Rootkit by the use of various techniques which we will explain later. That feature of rootkits poses a serious risk for the system since modern anti-malware tools will fail to detect and remove this type malicious software [12].

3.2 Rootkit deployment

A broad view of malware deployment process can be described by the Cyber Kill Chain model [13]. The military has introduced originally the Kill Chain model to describe the various phases to launch an attack. Lockheed Martin [14] has adopted the concept to describe the phases of a cyber attack. Figure 3.1 depicts the 7 phases of an attack, according to this model.

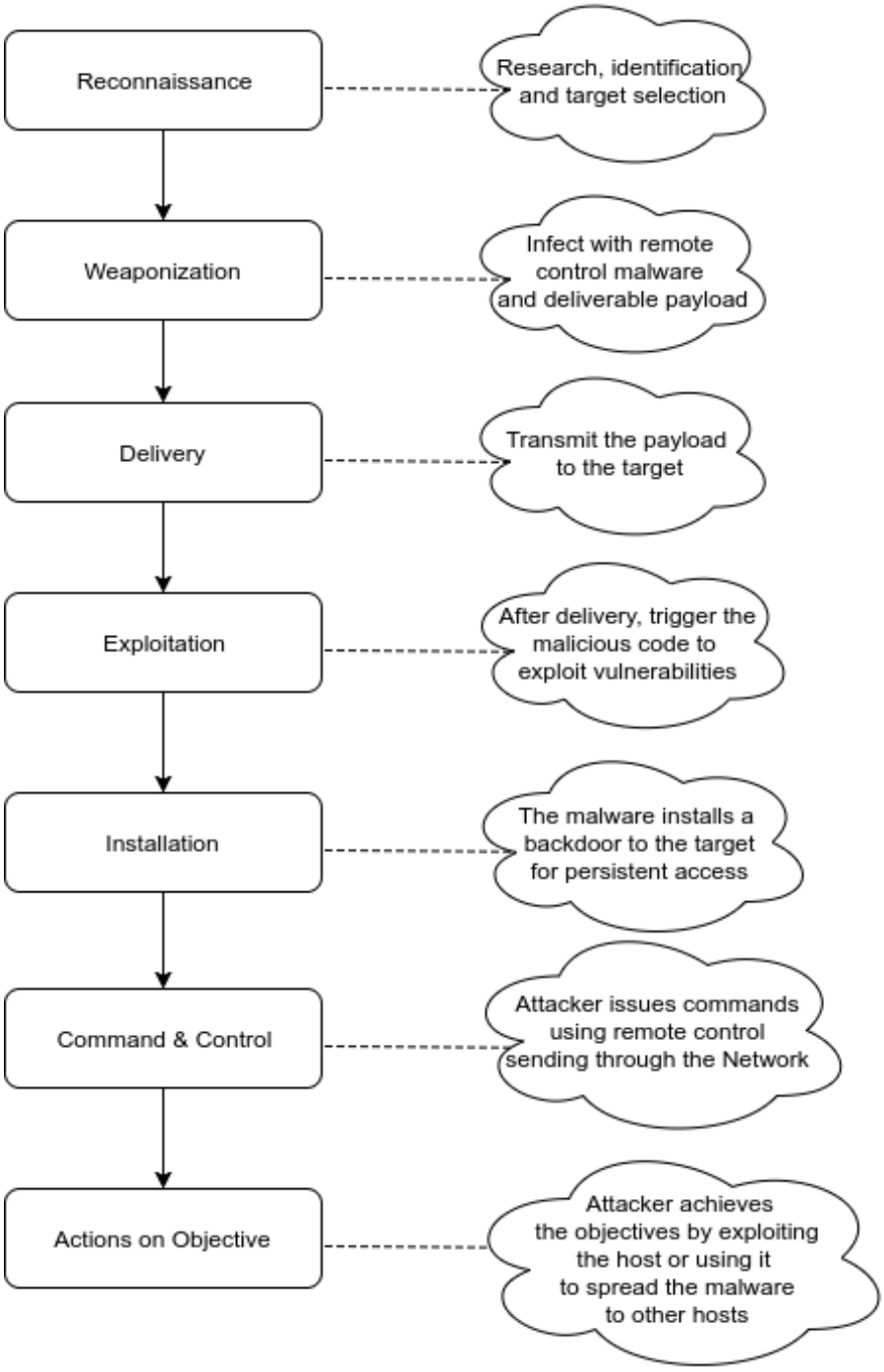

Figure 3.1: Attack phases according to the Cyber Kill Chain model

Each stage of the Kill Chain is an open research area, and more details can be found here [13].

In the case of a Rootkit, the malware analysis is focusing on the installation phase. An attacker will install a Rootkit after a successful exploitation phase and may postpone the latter phases for a future time, as the figure 3.2 demonstrates.

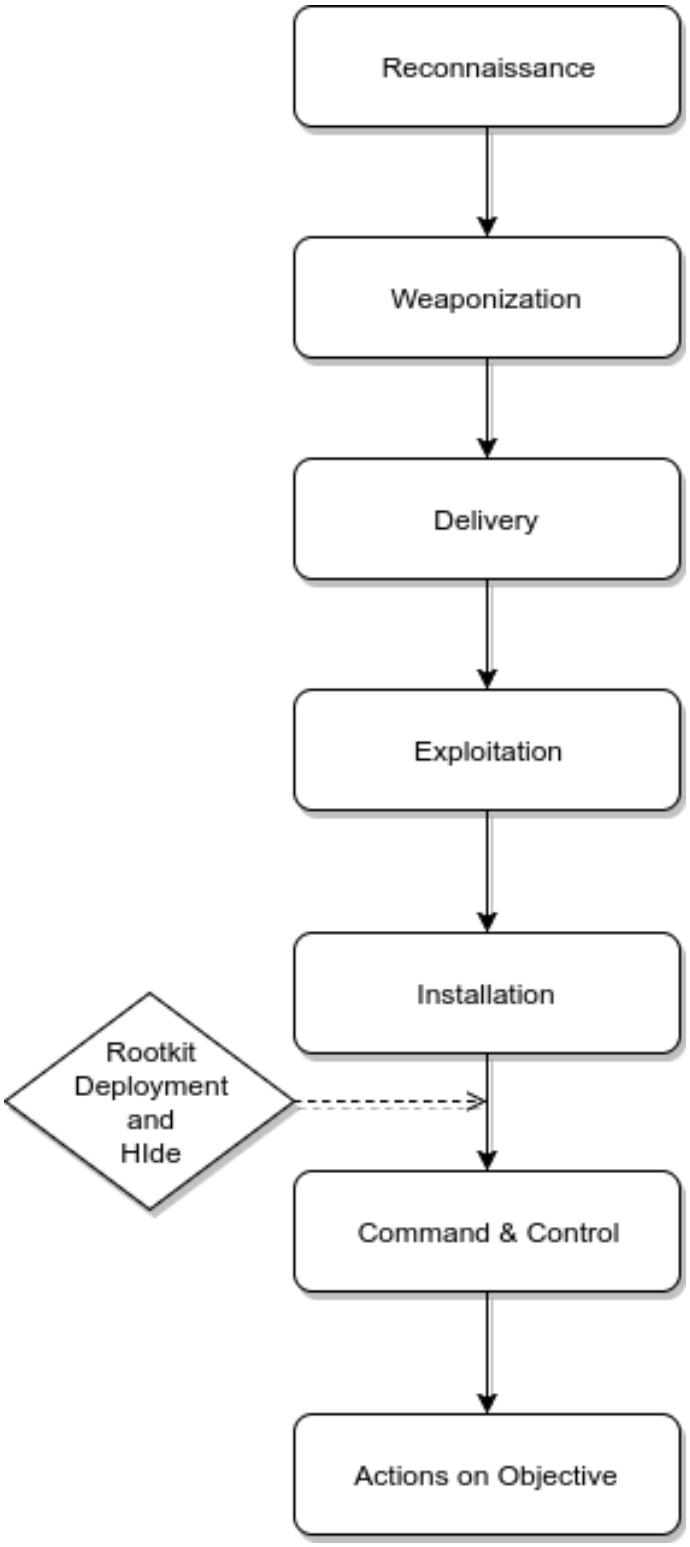

Figure 3.2: The Rootkit positioning in the Cyber Kill Chain model

3.3 Phases of Rootkit attack

A universal model for Kernel-level Rootkits has been introduced by X. Li, Y. Zhang, and Y. Tang [15]. Figure 3.3 demonstrates the model regarding the Kernel and User-space. In this section, we briefly introduce the phases being used by a typical Kernel-level Rootkit.

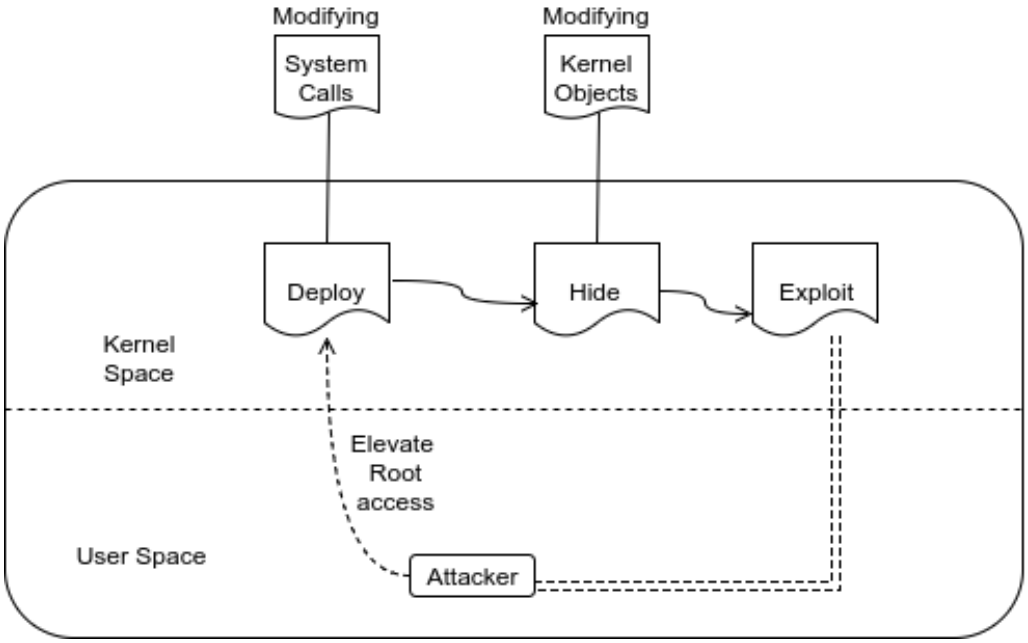

Figure 3.3: An universal Rootkit model

a) **Rootkit injection**. As described above in this phase the attacker has already root access, and he will try to modify essential parts of the kernel. Common targets are the Linux Kernel Modules (LKM), the System Call mechanism and virtual memory space (dev/kmem). We provide more details in upcoming chapters.

b) **Hiding phase**. As we will make clear later on, the purpose of using Rootkits is not immediate gains but rather future exploitation in a repeating process. To achieve the above goal the attacker needs to hide any traces and modification(s) that made to the Kernel.

c) **Working phase**. This phase is the actual exploitation and is used by that attacker to achieve the initial goal. A common use of Rootkits is keystrokes (keyloggers) to sniff passwords, open sockets to send and receive undetected traffic, spam relays, and botnets.

3.4 Advanced persistent threat

Advanced Persistent Threats (APT) [16] can be defined as a quiet, secretive and repeat procedure for compromising a computer system, which an organized group

of individuals coordinate and focusing on a single particular organization.

In this way, we can classify Rootkits also as APT, because the attacker is not focusing on gains immediately but rather for future and long term exploitation. Thus, Rootkits can pose a future undefined and unpredictable threat for the exposed organization.

Furthermore, the attacker may follow a strategic and organized plan to compromise and expose the targeted organization. It is not unrealistic nowadays that organized and professional security groups can focus and deliberately produce and propagate malicious code.

A widely well-known example of professional malware deployment is the BMG Entertainment rootkit [17]. Sony has released audio CDs with a software package extension called Extended Copy Protection (XCP) [18]. The intention was to prevent from the illegal copy of audio CDs. A cloaking technique [18] which makes all the process with sys prefix invisible hides the existence of XCP. This feature exposes the system to any attacker that was aware of this detail. Finally, Sony publicly announced the existence of the malware and issued a removal tool to fix the exposed client's computers.

Wikileaks has revealed a more recent report about organized malware deployment [19]. Wikileaks encapsulates the whole project under the name "Vault 7: CIA Hacking Tools Revealed". This report describes conversations of a security group inside the CIA about installing Rootkits and similar malware into embedded devices such as smart-TVs and smartphones. In addition, on October 2014 Wikileaks [19] already revealed discussions of the same group about investigating techniques of a possible insertion of malware into Vehicle Systems.

The fact that intelligence agencies are working in modern and sophisticated malware is not the main concern. The alarming issue is that intelligence agencies have leaked this kind of software in the past [20]. Furthermore, reverse engineering might be possible; therefore, attackers can be in possession of advanced and powerful malware without needing to develop one.

In conclusion, the modern malware has become more sophisticated and constantly evolving. Therefore, the traditional reactive approach, where updating the components after we discover an exploitation, could not be enough [21]. Consequently, we should consider a more proactive approach. The latter should try to detect and prevent the attempt of the attacker to alter the system before the deployment of the malware. Therefore, we should introduce verification techniques during the boot process to assure that the Kernel is not affected by such malicious software. In the next chapter, we provide the details and the anatomy of Rootkits.

4
Anatomy of Rootkits

In this chapter we describe the different methods that Rootkits are using to modify the Kernel. The purpose of this analysis is to investigate how different malicious modifications of the Kernel works and if we need to detect Rootkits during the boot-up process.

4.1 Control flow Hijack

The main target of the attacker is to redirect the regular execution flow of the Kernel. The main technique to achieve control flow hijack is hooking. Hooking is a technique used to modify, interrupt or shift system calls.

System developers have initially used hooking to create a dynamic behavior of APIs (Application Programming Interface). On the other hand, attackers could try to abuse this feature for malicious purposes. In the following chapters, we demonstrate how an attacker can abuse certain hooking techniques.

4.1.1 Wrapping system calls

In modern operating systems, there are two ways to link a program with a library function [22]. On the one hand, it can be done statically in the compiler-linker phase, which will produce a self-contained executable program. On the other hand, libraries can be linked dynamically in the execution phase. The latter is most commonly used because it reduces the memory requirements of programs by sharing the library code.

As explained in Chapter 2, the *libc* system calls are just a wrapper function. The manual page of Linux provides the details of the dynamic linking [7]. Figure 4.1 presents the normal flow of a program using a dynamically linked shared library.

The dynamic linker of Linux has introduced a feature called "preloading" by using the environment variables such as *LD_PRELOAD* or *LD_LIBRARY_PATH*. The intention was to allow programmers to load user-defined libraries dynamically and link into the executed program, before any other shared library [22].

An attacker might try to abuse the previous feature by using the environment variable *LD_PRELOAD*. The attacker will try to upload his own wrapper libraries that

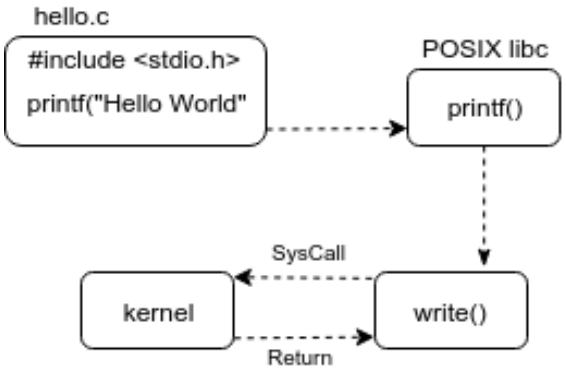

Figure 4.1: System-call of dynamically linked shared library.

change the execution flow. It is worth to mention that the linker will ignore the preload if the ruid is different than the euid; therefore, this attack will work only for the currently exposed user into the operating system. We could classify this attack as user-mode Rootkit, but as we will explain later, an attacker with root access to the operating system can elevate it to a kernel-mode Rootkit. Figure 4.2 demonstrates a possible violation of the execution flow when the attacker inserts his own shared library.

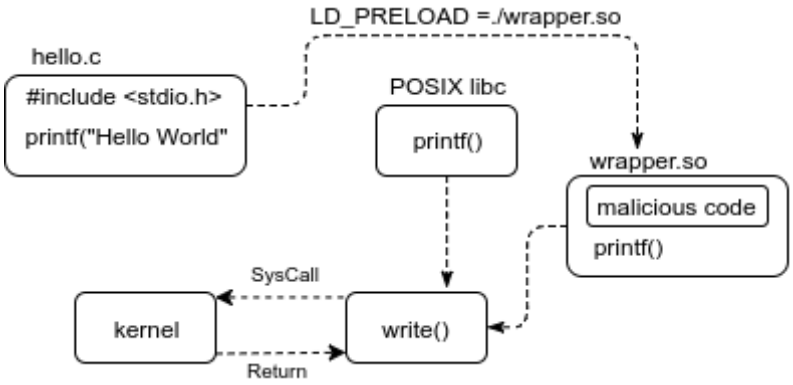

Figure 4.2: System-call using the LD_PRELOAD environment variable.

The previous limitation does not exist when the attacker modifies the */etc/ls.so.preload* file, which needs root access to modify. The loader will preload all of the libraries listed in the mentioned file, and this applies to every user in the operating system. Hence, the attacker by using the same mechanism can also hook system calls.

4.1.2 Modifying the system call table

The goal of this technique is to directly modify the syscall table which we have explained in Chapter 2.2.2. The intention of the attacker is to hijack the execution flow as figure 4.3 presents. Consequently, the kernel will invoke the malicious code instead of the handler function.

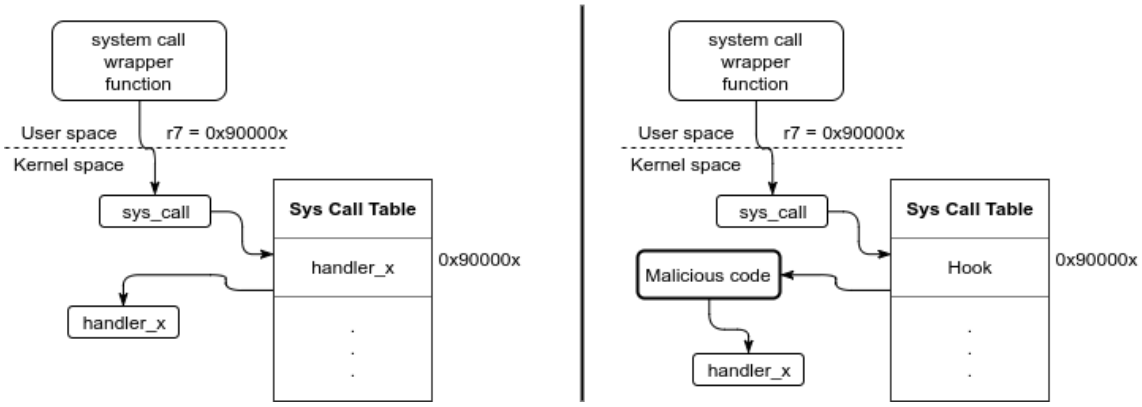

Figure 4.3: Before and after hooking the system table

In older versions of the kernel, Rootkits tried to modify the "sys_call_table" symbol to change and hook the system call table. After version 2.6 of Linux-kernel, the use of a symbolic name for the system table has been disabled. For this reason, nowadays it is harder to find the correct address of the table, but it is still possible.

As we explained in the Section 2.2.2, the function *sys_call* has direct access to the syscall table. Therefore, by analyzing the assembly code, it is possible to get the address of the table.

To get the *sys_call* function address, the attacker will try to read the Interrupt Descriptor Table (in x86) or the Exception Vector Table (in ARM). The mentioned tables are created at the start-up time of the system in a well-known address. The CPU also has a specific register with the address of the table. Therefore, the following steps could be used by the attacker:

- Get the value of IDT Register
- Find the address of syscall
- Analyze the syscall function to find the address of the syscall table

Figure 4.4 demonstrates a broad view of all the components involved during the handle of a system call.

4.1.3 Modifying the system call handlers

Another technique to subvert the execution flow of a system call is to modify the first instructions of the system call handler. The attacker will introduce a jump into his malicious code, execute part of Rootkit and then returning to the original execution flow. In this way, the system call becomes part of the malware. Figure 4.5 depicts the mentioned technique.

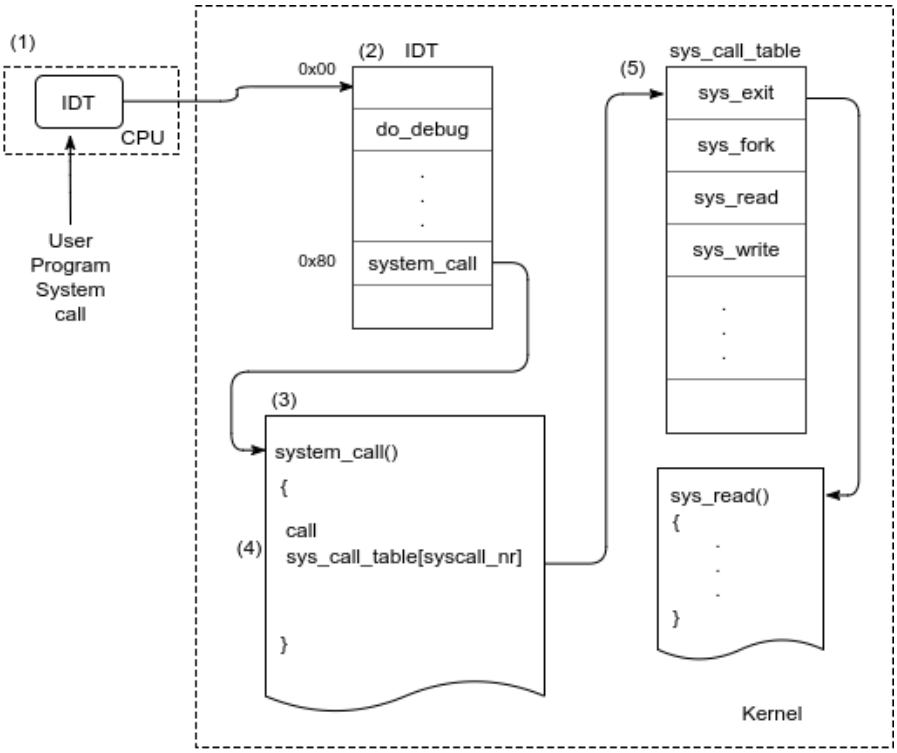

Figure 4.4: System-call overview

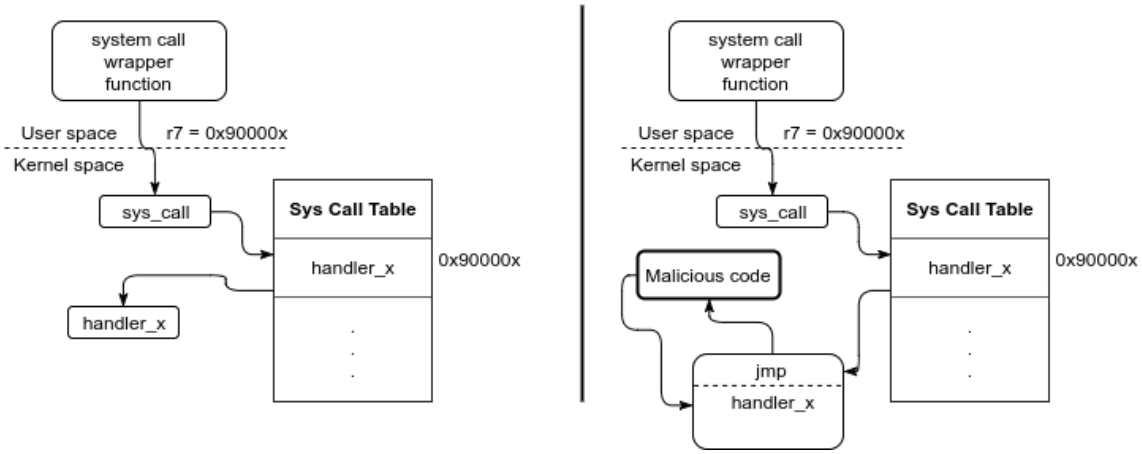

Figure 4.5: System-call handler modification

4.1.4 Modifying the system call function

Another way to hook the system call table is to modify the syscall function [10]. As explained in previous chapters, in every system call the kernel invokes the syscall function and directs to the right handler. If the attacker successfully modifies the syscall function, she can achieve to execute her malicious code in every system call. Figure 4.6 depicts the mentioned modification.

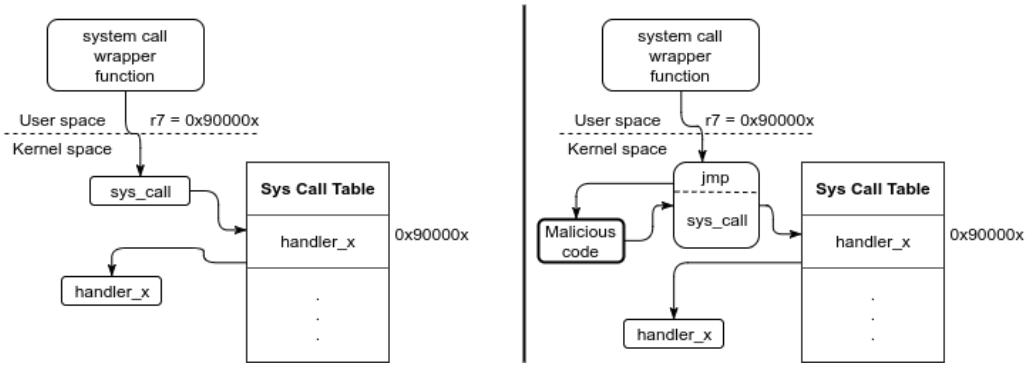

Figure 4.6: Modifying the System call function

4.1.5 Modifying the Interrupt & Exception vector table

The idea is the same as the modification of system call table. As we explained in previous chapters, the IDT & Vector table is initialized in the startup process in well-known memory addresses. The idea is to redirect the execution flow to the attacker's interrupt handlers.

4.1.6 Modifying the interrupt handler

In the same way as the system call handler attack, the attacker will try to alter the first instructions of the interrupt handler. Therefore, the handler will jump into the malicious code and later return again to the handler and continue the normal flow of execution.

4.2 Dynamic techniques

Most of the above techniques modify the kernel code which usually resides on the hard disk or the flash memory. Except for these techniques, an attacker could try to use or modify dynamic structures of the kernel into the main memory created in execution time. The latter is more stealthy techniques since they do not violate the code integrity of the kernel. In this way, it is harder to detect or prevent such an attack.

4.2.1 Direct memory access

In Chapter 2, we saw one feature that Linux used to dynamically augment its functionality, which is LKM(s). Similar functionality can be achieved by using the devices */dev/mem* & */dev/kmem* [7], which are used as virtual memory for the kernel. These devices are directly connected with the main memory and can be accessed as regular character files [7]. Hence, developers could dynamically patch the kernel using those devices without a system reboot.

On the other hand, an attacker with root access could try to access or modify the virtual memory of the kernel to alter the behavior of the Kernel. The mentioned attack

is quite stealthy since there is no integrity violation of the kernel code. Nevertheless, after a system reboot, the attacker needs to re-operate the attack phase.

4.2.2 Return Oriented Programming (ROP)

This attack has emerged as a sophisticated version of the buffer overflow attack. In general, most of the function calls are using the stack data structure to save their local variables as Figure 4.7 depicts. Due to a poor boundary checker, an attacker could try to write adjacent parts of the stack. With a successful overriding of the return address, the attacker will take control of the execution flow of the program [23]. Figure 4.8 depicts the Buffer-overflow attack.

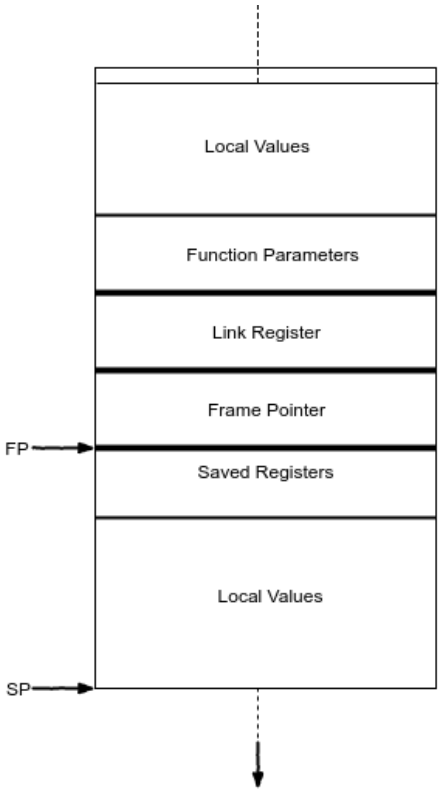

Figure 4.7: Function Call Stuck in ARM.

The previous attack description is well-known, and the first public report was in 1972 [24]. Over the years, the research community has come up with various protection mechanisms such as Non-Executable Buffers [23], StackGuard(canaries) [23], and Address Space Layout Randomization(ASLR) [25]. Thus, attackers start focusing on other ways of subverting the execution flow of the program.

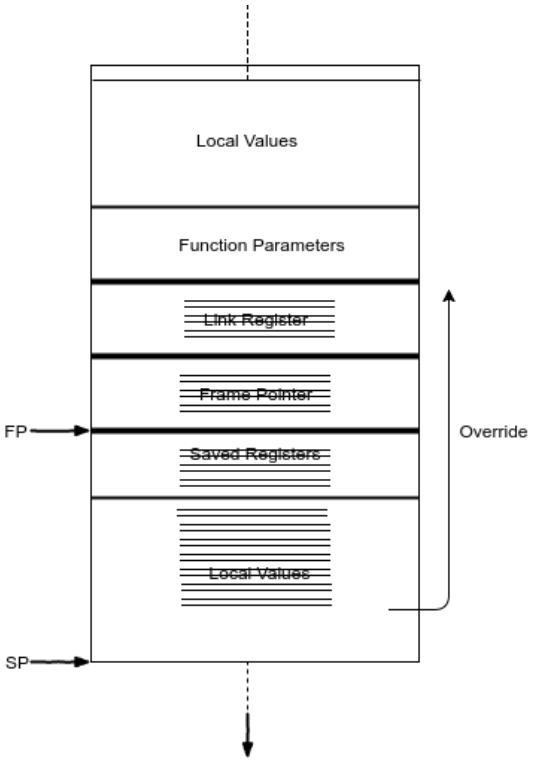

Figure 4.8: Buffer-overflow

Consequently, attackers manage to find ways to override the return address such as indirect pointer overwrite [26] and format string vulnerabilities [27]. As an example, the attacker will try to modify the return address into well-known libraries such as libc (return-into-libc [28]). The previous exploit has the limitation of using only pre-existing functions.

In the x86 CPU architecture, instructions can have variable length size. Consequently, we can modify the return address to point into the middle of the address of the next instruction; therefore, it is possible to be translated into a legitimate executable "new" instruction [29]. Figure 4.9 depicts this transformation.

Not all combinations are producing legitimate instruction, and also the attacker would need more than one instruction to deploy an exploitation. Nevertheless, it has been shown that such an attack exists, introducing a new way of programming called Return Oriented Programming (ROP) [29]. In addition, in [29] it has been shown that with certain conditions, the ROP can be Turing complete. The attacker analyzes common libraries for code snippets that produce legitimate instructions ending with a ret command. The previous are called gadgets [29] and the exploit is performed by chaining different gadgets together as figure 4.10 demonstrates.

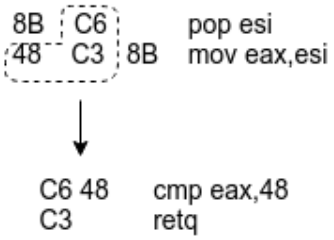

Figure 4.9: A possible instruction transformation

Furthermore, ROP is not limited to the x86 architecture. In the paper of Z. S. Huang and I. G. HarrisIn [30], they show that ROP is possible in ARM architecture as well. The branch and pop instructions are being used to achieve the gadget chained.

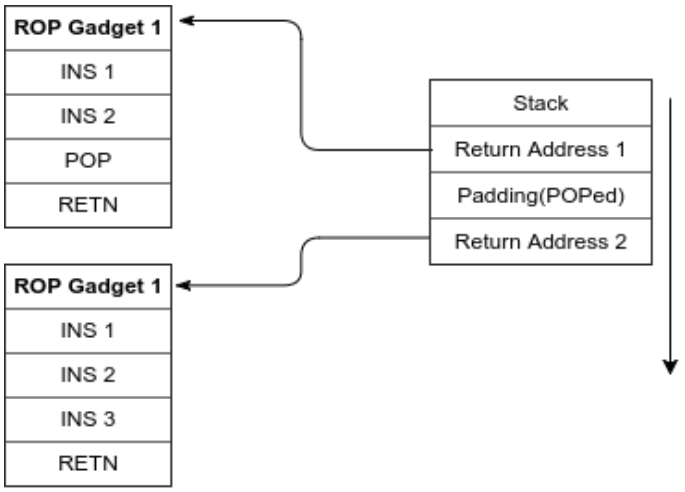

Figure 4.10: A ROP gadgets example

4.2.3 Non-persistent hardware supported Rootkits

This is another category of stealthy Rootkits that do not modify any data of the kernel code. In this way detection techniques based on integrity or similar violation fails to discover the existence of such malware.

These types of Rootkits are targeting architecture specific vulnerabilities, which help Rootkits to hide and co-exist with the other components of the system. Since they do not modify the kernel, they need to provide their own services and be self-contained. In addition, they need to use the resources efficiently and stay hidden.

An example of this type of a Rootkit type is Cloaker, and the details can be found here [31]. Cloaker is targeting ARM platforms with certain specifications. Cloaker exploits the interrupt vector which is a feature of ARM architecture. More specifically it is possible to locate the interrupt vector to two different address by changing

a bit of a configuration register. The goal is to install a second interrupt vector in the new address and leave untouched the original one; the OS will still use the original vector table. In this way, integrity check will fail to detect the alteration in execution flow.

The other feature that Cloaker exploit is the translation lookaside buffer (TLB). ARM processor allows for certain entries of the table to be locked down. Consequently, the attacker is able to use physical memory addresses without modifying any page table of the virtual memory.

Finally, the payload includes a micro-OS which provides various facilities. As stated above, this type of malware does not depend on OS services. Therefore, there are three components to provide system services. First, there is memory management for dynamic allocation of memory. Then, an interrupt manager provides the interaction with the hardware. Lastly, a network API for sending and receiving IP packets.

In conclusion, this type of malware is quite stealthy and hard to detect. The attacker could establish a mini-OS to perform certain malicious activities. Nevertheless, this method does not survive after rebooting the system.

4.3 Hiding techniques

The kernel is creating and maintaining data structures such as process list and kernel module linked list. The attacker will try to modify those structures in order to hide specific files, processes and kernel modules. The common term used to categorize this technique is Direct Kernel Object Manipulation (DKOM) [32]. The mentioned techniques apply to the hiding phase of the Rootkit as described in a previous chapter.

A straightforward way to hide a process directly modifies the administrative tools that report active processes. The attacker will try to modify utilities such as *ps*, *top* and *pstree*. Nevertheless, this attack is easy to detect by simply checking the code integrity of the administrative tools.

A more elegant way to hide a process is to modify the */proc* file system directly. To report the status of the process, most of the administrative tools examine the files in the */proc* file system. This modification is a dynamic technique since there is no need to modify any program or kernel code. On the other hand, the attacker needs to modify the */proc* file system after each reboot of the system.

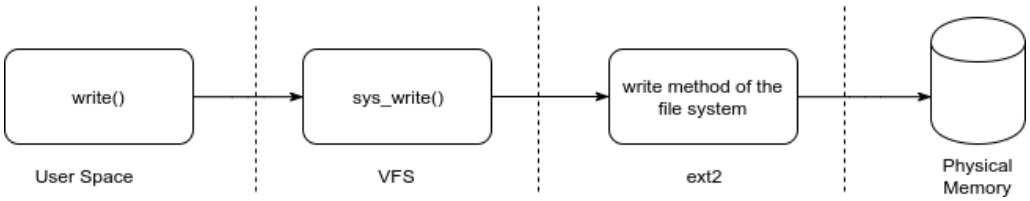

Figure 4.11: The Virtual File System interface in Linux systems

To avoid modifying the */proc* file system each time, attackers manage to find other ways. Virtual File System (VFS) is an abstraction layer (API) between the user program and the file system (i.e. ext2). Therefore, the user's programs can make system calls to handle files regardless of the underlying file system of the kernel. VFS is a common target for attackers because it handles the calls related to the special file system */proc* [10]. As mentioned above most of the administrator tools retrieve their information from the */proc* file system, the attacker will try to intercept those calls to hide the existence of certain calls to.

The previous attacks work well for specific tools, but other tools that are using different ways to report running process will reveal the hidden process. More advanced techniques are looking to modify the kernel's structures related to process handling. The kernel stores each process using a linked list data structure [10]. Each element has a next and previous attribute linked to the corresponding process. By modifying the links of the mentioned structure, an attacker can hide the existence of a process as figure 4.13 demonstrates.

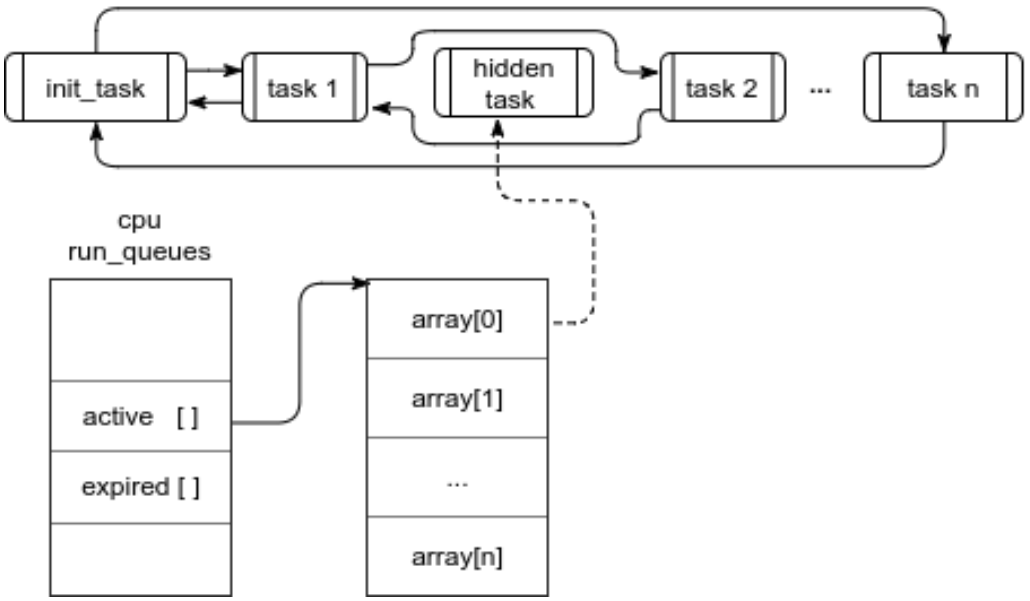

Figure 4.12: Process link list inside the Linux kernel

Another kernel structure that attackers targeted is the Linux scheduler. Every process in Linux systems is assigned a value called "goodness" [10]. The value is a combination of the static and dynamic properties of each process. The Linux

scheduler [10] use the mentioned value to choose and assign a ready process to the processor. For this reason, the scheduler is keeping two running queues. One active to keep track of processes with remaining CPU time and one expired to keep track of processes that the time slice has expired. An attacker will try to modify the running queue; thereby, the malicious process will never expire. Therefore, there is no need to keep a process id to reallocate the process time. The result is a hidden process into the system [33].

4.4 Bootkits

With the advance of various Rootkit techniques the research community has responded with different detection mechanisms such as kernel integrity monitors [34], virtual machine monitors [34] and bus snooping [35].

In response, the attackers try to establish deeper structures to hide their malicious code. Consequently, the boot process and the firmware of the hardware become the new target for the attackers. In a successful modification of either boot or firmware code, the attacker will be able to run and prevail his code before any other detection mechanism in the system. Those new tools focusing in the start-up of the system are covered with the term Bootkit [36].

Bootkit is a set of boot and start-up tools with the intention of permanent exploitation with a highly undetectable manner. One way is to modify and extend boot- or system files in order to insert malicious behavior during the boot-time. The other way is to run malicious code in the start-up phase before any other boot-code of the system.

In the next sections, we describe the different techniques used in personal computers and the relevant appearance of those attacks into embedded devices such as smartphones.

4.4.1 Bootkits in Personal Computers

In personal computers, the first type of Bootkits appeared in 1987 (Stoned Bootkit) [37] which affects the boot-section of the hard drive. For a long period of time, this type of malware wasn't widespread, and the attackers had barely focused on it for exploitation. In modern personal computers, after certain successful attacks, a significant amount of malware appeared.

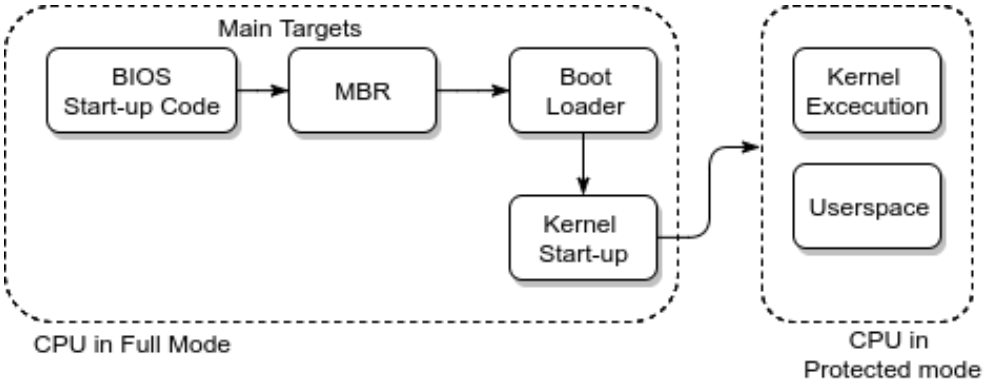

Figure 4.13: Main Targets of Bootkit

In [38] the evolution of modern Bootkits in personal computers is presented. Figure 4.13 demonstrates the main targets of Bootkits. The BIOS start-up is the first code to be executed and for this reason modern bootkis are trying to compromise it. Modern manufacturers introduce more sophisticated firmware such as Unified Extensible Firmware Interface (UEFI) [39] to replace the BIOS; nevertheless, a successful compromise of UEFI has been reported [39].

The Master Boot Sector (MBR) is the first part of the hard disk and keeps vital information about the disk partitions, namely the Operating System and File System. As described in [38] there is a significant amount of Bootkits trying to modify the MBR.

The bootloader is responsible for preparing the kernel execution, load the kernel and pass the control from CPU to the kernel. The code of the bootloader has to be small and efficient. Thus, few protection mechanisms exist at this stage of the system. Consequently, the bootloader is a possible target for an attacker.

Finally, the appearance of operating systems in embedded systems is a recent phenomenon and mostly through the success of smartphones. There is no proof of concept for compromised systems with Bootkits, but reports [38] refer to the possible appearance of Bootkits in Android devices.

4.5 Conclusion of threat analysis

Our threat analysis has presented different malicious techniques which an attacker could use to infect the Kernel and the boot process. We can not detect those modifications after the boot up of the Kernel. Since an infected Kernel with a Rootkit will provide false information to any detection mechanism that is running after the start-up of the Kernel. This analysis justifies the need for security measures during the boot process.

5
Verified boot

In order to prevent and/or detect the threats mentioned in the previous chapters, we need to introduce suitable terms, concepts, and standards. The main concept of our thesis is Verified boot which is thoroughly defined in this chapter. The outline of this chapter is as following, subsection 5.1 provides a comprehensive definition of the Verified boot concept as well as the term *Verified boot*. Consequently, subsection 5.2 consists of definitions for commonly used concepts when implementing Verified boot. Subsection 5.3 explains some of the states of the art techniques (introduced with their origin) which are derived by using the concepts in 5.2. Finally, the chapter is concluded with subsection 5.4, a conclusion section which summarizes the important outcome of various verification techniques.

5.1 Definition of Verified boot

The term Verified boot has been vaguely used throughout various research and industrial contexts. This is presumably due to the fact that the first used term was the Secure boot to describe a concept which provides a secure and trustworthy computer systems. So today, when the companies slowly start to implement a secure and trustworthy boot-up process in their embedded systems, the term Secure boot and other terms are being used to describe the concept without reaching any consensus about one single term to use. In this section, we will see a general definition of Verified boot, based on research papers and industrial implementations.

We can commonly see the concepts such as Secure boot, Trusted Boot, Measured Boot and Authenticated boot being used while explaining in principle similar functionality as Verified boot. The above mentioned techniques have been defined and implemented in various environment specific forms [40][41][42] [43]. Especially, the Secure boot has been among the pioneering concepts to describe a boot process with the essential duty of integrity assurance. Tygar and Yee [44] are presenting the first secure boot process using a secure co-processor in their model. Secure boot was initially aimed at PCs but has lately been in the scope of the embedded system providers. Secure boot aims to recover or terminate the modified system, whereas Verified boot mainly provides a report (verification) about the genuineness of the boot-up code or terminates/recovers the system if the modifications are too severe. The difference in names of the concepts for PC and embedded systems is due to the fact that embedded systems have a lower performance capability compared to

PC systems and in most of the cases the embedded systems also have real-time constraints which makes it more difficult to freely implement time consuming cryptographic security measures.

Thus, Verified boot is used to describe a lighter version of Secure boot adapted to embedded systems, which is a booting technique essentially with respect to the integrity assurance of the boot process code but also awareness of potential threats (e.g. in form of Rootkits or general malware) which might take action at a later time. Finally, it is also providing the flexibility for manufacturers to update and control the software. However, most Verified boot designers pay attention to authorization (e.g. to ensure that the correct device has the necessary right to run the code) and freshness (e.g. to prevent replay attacks) as well.

5.1.1 Definition by hardware and software providers

Below we can see the definition of the Verified boot by *Das U-Boot*, a bootloader provided by the DENX Software Engineering community [45][46], *Chromium OS*, an open source project that aims to build an operating system pioneered by Google [47] and *Android*, a mobile operating system developed by Google [48]. Chromium OS and Android provides a definition which concentrates on their specific devices. Observe that these devices are including environment specific functionality in their design, thereby expects a user to interact with the device (in contrast to autonomous real-time embedded devices). A simple example is that a mobile phone or PC user can be prompted to have the option to choose whether or not continue the boot process of the device, meanwhile an autonomous real-time safety-critical embedded device in a vehicle has to make a decision by itself and take action immediately (by terminating or restarting the boot process).

I. Verified boot in Das U-Boot

The main goal of a Verified boot process is claimed to have a device which runs the expected code at each bootup instance. This is maintained by preventing malware infections, providing safe updates and allowing the manufacturer to control the software.

U-Boot uses two essential concepts to implement Verified boot; cryptographic hashing and public key cryptography. Software images (e.g. a Kernel image or another U-boot bootloader) which are executed by U-boot, can be verified on the target system by;

1. Creating an asymmetric key pair (with e.g. RSA) on a host device.
2. Running the software image (e.g. kernel image) through a hash function (e.g. SHA-1 or SHA256) on a host device.
3. Signing the hash value of the image on a host device which keeps the private key secure and embeds the public key into the U-boot binary.
4. Finally, distribute the U-boot binary (which contains the public key) to the target system (which will run U-boot) for it to verify the signed image.

U-Boot claims that the concept of Verified boot comes from Chromium OS and is an easy way to provide security for a system with minimal code size and boot time overhead [46].

II. Verified boot in Chromium OS

Chromium OS describes the Verified boot functionality as a solution to ensure the security of the users when they log in to their Chromium OS device. The verification of the boot-up process starts in the read-only portion of a firmware and ranges until the kernel is executed. The Trusted Chain model (which is explained as *Chain of Trust* in Section 5.2) is followed here which results in an execution of the next stage only if the current stage has verified it. The goal is to ensure that all of the boot process code comes from the Chromium OS source tree (tree hierarchy consisting of directories), instead of an attacker or corruption.

The user should be able to reset the device to a known previous reliable state if the Verified boot functionality detects a problem. Alternatively, the user can proceed the boot-up process since the detection might be due to an update in the system [47].

III. Verified boot in Android

Android software designers define Verified boot as *"an implementation which guarantees the integrity of the device software starting from a hardware root of trust up to the system partition."* [48]. Here, *Root of Trust* implies the absolute start point of the boot process. The strategy used to guarantee the integrity of each stage is fulfilled by, starting from the system hardware level, each stage verifies the integrity and authenticity of the next stage before allowing it to be executed. This concept is called the *Chain of Trust*. The comprehensive definitions of the Root of Trust and Chain of Trust will be explained with necessary details in the subsections 5.2.

They further explain the aim of Verified boot as, to ensure the users that their device is in the same state as it was when last used. In this case, either the user is notified considering a state change in the system (which might give the user an option to continue with the next stage, depending on the severity of the warning), or the system boots up without any complications.

5.1.2 General definition

In order to make a new and general definition of the Verified boot by using various already existing definitions, we have to base it on the definition of the word *Verification*, and specifically its meaning in the technical concept. It is defined as;

"Verification. The evaluation of whether or not a product, service, or system complies with a regulation, requirement, specification, or imposed condition. It is often

an internal process. Contrast with validation." [49]

The relevant part of our subject is that the system shall comply with a requirement, specification or imposed condition. So our system shall act as the designer/owner desires even at the next boot-up or any arbitrary consecutive boot-up. We can reach a consensus regarding that verification is an evaluation process which requires the system/authority to approve if the boot-up process code was unaltered and warn/take action if the process code was altered.

A common aspect which is clearly noticeable among the previously mentioned systems which define Verified boot is that they guarantee protection against potential threats which might alter the entire or part of the boot-up process code. In other words, they implement security. To implement the verification, various techniques can be used (presented in Section 5.3). When implementing these techniques, which cause an overhead in the system regarding the area and boot-up time, it is desirable to benefit from the security features these techniques provide. So this is exactly what most of the Verified boot designers do.

We have tried to express a definition of the Verified boot which is as self-explanatory as possible. Thereby, we maintain an extensive definition which is;

Definition 1. *Verified boot is a booting process which provides a certain level of security at each boot-up instance by aiming to guarantee detection of unwanted modification in the boot system critical code, ranging from the Root of Trust to the final stage. The security and detection measures are in the proportional extent to the hardware and real-time constraints. Thereby, if the altered code jeopardizes the genuine control ability of the system boot-up process until reaching the last stage, the system enters a recovery mode or terminates. However, if the boot process is allowed to reach the last stage, and the code is altered, the desired action is taken according to the system functionality. Finally, if the system genuineness is verified, the system proceeds functioning.*

The general term *security* in this context aims to describe the required level of integrity protection and authenticity in a system boot up process, as well as other necessary concepts which might be relevant for the specific system (e.g. freshness).

Some Verified boot designs can include the ability to detect replaced hardware components but this functionality results in additional complexity and thereby additional overhead. The case might be that the detection of hardware replacement is out of the question depending on the design. An example is a case when the system is using an exclusive hardware component for the verification process (e.g. a *TMP*, explained in 5.2), since the system has a single point of failure, if the attacker manages to replace this device, the verification process might become unreliable. Hence, a distributed verification design or remote attestation is one of the options to detect hardware replacements (see *Sealed Storage* and *Remote attestation*, Section 5.2)

By saying *"the last stage"* in our definition, we mean the final stage which is set by the system designer. However, this stage can be considered as the stage before any application level process starts according to our research about various Verified boot techniques. Generally, this stage is the kernel execution stage but can freely be defined by the system designer.

The security and detection measures are in the proportional extent to the hardware and real-time constraints (which is estimated by the system designer). Thus, the security and detection expectations are increasing with a proportion to the increasing hardware performance and real-time flexibility. Hardware performance and real-time flexibility are in turn provided with respect to the security necessity of the system, cost of the hardware, heat/area limitations and the system purpose.

A concept frequently mentioned in the same context as verified boot is trusted computing. The concept is defined by the *Trusted Computing Group* (TCG). We will see that many techniques are using the standards provided by TCG under the *Trusted Computing* concept. Verified boot, however, can be considered as a generic concept which can be implemented without the restriction of any standard or hardware.

5.2 Key concepts and standards used in Verified boot

In order to have an understanding of the various Verified boot techniques, a number of key concepts and popular standards must be known. Some of these concepts are specific for the application level (which is out of the scope of this thesis) but are yet mentioned in short to provide a comprehensive understanding. Most of these concepts can be combined in order to design a Verified boot system. The section outline is as follows; The fundamental standard and concepts which are used in most of the techniques are explained first (5.2.1-3) and more specific concepts are described consequently.

5.2.1 Chain of Trust and Root of Trust

In a *Chain of Trust* (CoT), each stage in an execution hierarchy (e.g. boot-up process) is verified by its predecessor and consequently executed. This can either be done with the aid of a monitoring dedicated device (hardware CoT) or solely by the stages themselves (software CoT). The first stage is referred to as the Root of Trust (RoT) by the Trusted Computing Group (TCG) or as Trust Anchor in general contexts. The RoT concept (in most cases) consists of a trusted hardware component which is assumed to be fully reliable (i.e. ROM device in an embedded system). TCG has defined the absolute first component to be trusted as the Core Root of Trust for Measurement (CRTM) which is considered to be immutable. Different techniques can be used to achieve this concept but one of the most common ones is by doing an integrity check on each stage by getting a verification from a dedicated trusted

device, starting from the RoT and only executing next step if it passes the integrity check and thereby constructing a CoT. [41][50]

5.2.2 Trusted Computing by TCPA (TCG)

Trusted Computing is an industrially developed guideline, which provides security to diverse computer systems by using the RoT concept. It was proposed by the Trusted Computing Platform Alliance (TCPA) in 2003 [51]. TCPA was founded by IBM, Intel, Microsoft, HP, and Compaq in 1998 to form a specification for a secure chip used in PCs due to the increased importance of computer security in industrial and commercial environments. The popularity grew quickly and over one hundred member companies joined TCPA within a short period of time. TCPA was however considered to have a closed nature which limited the research and contribution potential on the specifications.

In 2005 the Trusted Computing Group (TCG) was formed (with a board consisting of IBM, Microsoft, HP, Infineon, Lenovo, Fujitsu, Wave Systems, Sun, Intel, and AMD), where everyone can join and contribute to the specifications as long as they pay their dues [52]. The Trusted Computing concept was initially developed for PC systems including a dedicated hardware device with cryptographic capabilities (i.e. TMP, see 5.2.3), but has been extended and widely used in smaller devices such as hard disk drives, mobile phones or general embedded systems. Some of its context-relevant benefits are listed below.

- Protects code and data against unauthorized changes (integrity protection)
- Authentication and protection for certificates, keys, and passwords
- Secure and flexible remote access between devices for authentication (see 5.2.7 Remote Attestation)

[53]

5.2.3 Trusted Platform Module (TPM)

Trusted Platform Module (TPM) is an international standard implemented in dedicated hardware devices which can be seen as a secure co-processor to fulfill most of the security operations in a system [51]. A TPM is used to authenticate the system in question by storing passwords, certificates or keys [54]. Another possible task of the TMP is to store crucial trusted measurements of the system so that the system can prove its trustworthiness to the TPM or to store untrusted measurements of the system which are sent to a remote party for comparison (see Remote attestation 5.2.7). The TPM can also aid the main system by functioning as a RoT. Where RoT is intended to be the initial basic block in the trusted computing system used to validate the first boot stage [55]. TPM was proposed by the TCG in 2009 and has been developed since then [56].

In Figure 5.1 we can see the different components which are necessary for a secure chip to be considered as a TPM according to the latest TPM specifications (v. 1.2).

We are going to shortly describe the task of each component and thereby mention the basic functionality expected of a TPM.

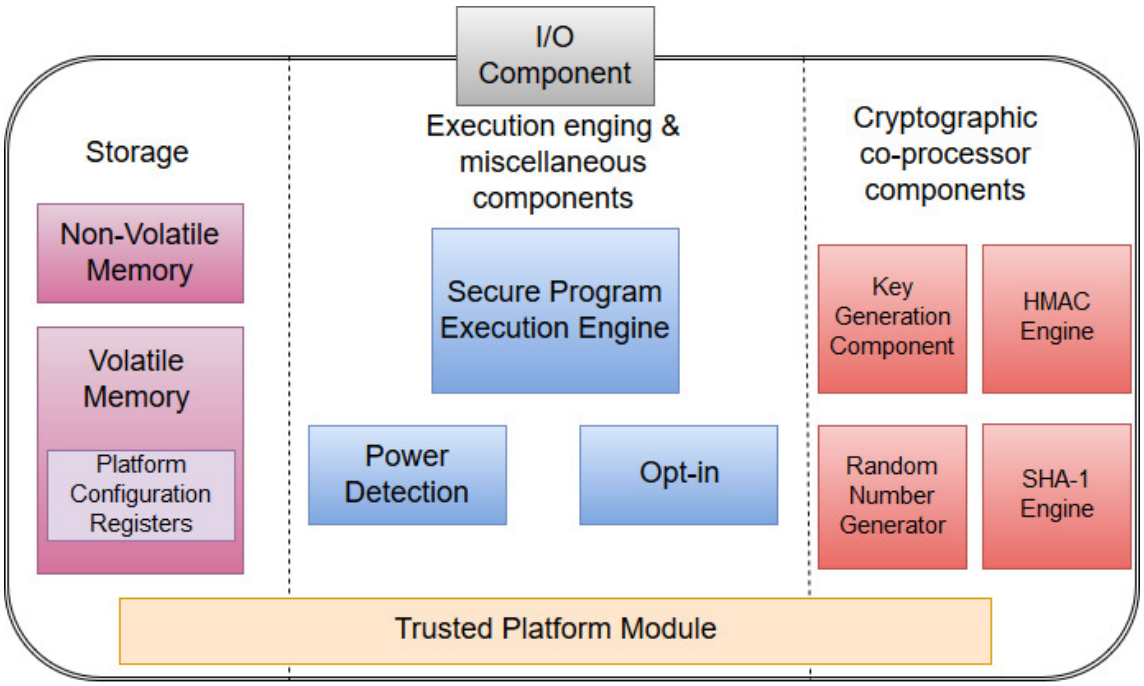

Figure 5.1: TPM block diagram

5.2.3.1 TPM components

I/O component is handling the information exchange on the communication bus as well as encoding and decoding the data according to a specific protocol.

Cryptographic co-processor is handling the cryptographic operations such as asymmetric key generation (RSA), asymmetric encryption/decryption (RSA), hashing (SHA-1) and random number generation. The latest version of TPM must support the cryptographic algorithms, RSA, SHA-1, and HMAC. However, TPM can implement additional algorithms as desired.

Key Generation component is creating RSA key pairs and symmetric keys.

HMAC Engine is providing the TPM with two pieces of information. One is that a request that arrives to reach an object within the TPM shows the knowledge of the AuthData (a 160 bit shared secret which is implemented within secret objects). Second is that the request which arrives is authorized.

Random Number Generator provides random values for nonces, key generation and randomness in signatures to the TPM.

SHA-1 Engine supports measurement collection during platform boot phases yet primarily used by the TPM.

Power Detection handles the power state changes in the TPM in affiliation to the main system power states.

Opt-In is providing the TPM with functionalities such as being turned on/off, enabled/disabled or activated/deactivated in a protected fashion.

Execution Engine is executing program code provided by the I/O port to run TPM commands.

Non-Volatile Memory is storing persistent data such as identity and state of the TPM. It can also be used on behalf the TPM owner to store data belonging to authorized entities.

Volatile Memory is used to store temporary data which is reloaded on each power up or updated more frequently (e.g. Platform Configuration Registers, PCRs).

5.2.3.2 TPM Key types and features (v.1.2)

Keys	Functions
Attestation Identity Key (AIK)	Used to sign data only within the TPM itself (e.g. PCR values).
Endorsement Key (EK)	Is the only key commonly shipped with the TPM hardware, used to decrypt AIK certificates and initial setup.
Storage Keys	Used to store other keys outside of the TPM memory space.
Signing Keys	Used as general purpose keys by the TPM.
Legacy Keys	Can be used as signing or storage keys, normally to provide compatibility.
Binding Keys	Used to store small amounts of data.
Storage Root Key (SRK)	Used to store data outside the TPM memory space which can only be decrypted by the TPM itself.
Certified Migratable Keys (CMKs)	Keys that can be migrated (freely transferred out of the TPM, in contrast to non-migratable keys) to specific Migration Authorities selected at key creation phase.

Table 5.1: TPM keys and features

Platform Configuration Registers (PCRs) are 160-bit storage locations in the VRAM used by the TPM to store integrity measurements. There shall be minimum 16 registers but the vendors can extend the number to a maximum number offered by the TPM. The PCR can hold an unlimited number of measurement in its registers by using a cryptographic hash to concatenate and hash all (*Extend*) new values in one PCR. The procedure is presented in the pseudo code below:

» $PCR_i\_New = HASH\ (PCR_i\_OldValue\ //\ value\ to\ add)$

DIR bytes are 20 bytes of NVRAM used by the TPM to have a copy of the golden boot measurements which are used to interrupt the booting of a system if the PCRs are in an unstable state.

Taking Ownership is the process when the owner initially saves a shared secret (password) in a shielded TPM location. This is done once and will be required if an operation in the TPM requires authentication.

Maintenance capability is providing a backup functionality of the SRK in case of a motherboard failure, so that the SRK can be transferred to another system.

Direct Anonymous Attestation (DAA) Zero knowledge proof of that a specific AIK is produced by a specific TPM.

Timestamping a tick counter as an alternative to a real-time clock since it might be too expensive.

Monotonic Counter is a global counter that gets incremented by 1 each time but cannot get decremented.

Locality is a signal line used by the TPM to gather information about the system state when a command is sent to the TPM.

Audit is supplying the TPM with information about how it was used.

Caching is a technique used to cache the internal TPM memory to the hard disk by using symmetric encryption (quicker compared to asymmetric key operations).

The definitions have been concluded from [57] and [58] to potentially preserve a comprehensive understanding of the further content of this thesis report where various Verified boot techniques are explained.

5.2.4 Mobile Trusted Module (MTM)

The Mobile Trusted Module specification is a mobile version of TPM which can also be used in automated embedded systems. It has a smaller implementation footprint compared to a TPM implementation. This is due to the fact that an MTM includes a minimal set of the TPM functionality. Even though that MTMs can be implemented in dedicated hardware devices (hardware MTM) similar to a TPM device, the MTM is often suggested as a software module (software MTM) implemented in the environment which has the necessary hardware isolation. The MTM is designed with secure boot in mind (aborts the boot-up sequence if the measurements are not approved). The MTM standard is (as well as the TPM) flexible and can be implemented by different platforms in form of different designs. Hence, there are abstraction layers to provide compatibility which is proposed [41].

The intention of the MTM specification is to be flexible considering the hardware environment where it is implemented. It was first introduced to the the ARM TrustZone architecture (2008) and has since then shown intention to be a secure model specification deployable in various hardware architectures.

During the boot-up process, the device implementing MTM passes two standard defined states, *reset* and *root of trust initialization*. The reset state has a duty of checking that no software is running before the MTM setup whereas the root of trust initialization phase is defined by 5 root of trusts as described below;

Root of Trust for Enforcement (RTE) is a logic that halts the boot process on verification error.

Root of Trust for Measurement (RTM) is measuring the systems trust state to include it into the MTM.

Root of Trust for Verification (RTV) Validates the future execution environment of the MTM before initiating it.

Root of Trust for Storage (RTS) is maintaining a safe storage (confidentiality and integrity) in the underlying platform for the MTMs persistent state, assuming that the MTM is not considered as a physical device.

Root of Trust for Recording (RTR) Signs measurements for attestation purposes.

The security requirements proposed by the MTM specification are minimal for each of these stages but shall at least use 160-bit SHA-1 hashing. The reason for the weak security requirement for each of them is because they form the security of the end product as a set. The underlying hardware must be suitable to fulfill all 5 of the above-mentioned RoT requirements to implement MTM (on the other hand, on a basic TPM only the RTM needs to be taken into consideration due to the TPMs hardware capabilities in terms of integrity and isolation). We have to note that MTM is relying on the non-MTM security of the underlying platform.

The initialization process of a device using MTM will only be considered successful if it with respect to the RTV leads to a successful state. It will stay there as long as the system state updates match an obvious known state. If the match fails, the MTM process is moved to a failed state by the RTE. It will either reboot or other actions are taken to preserve the security and integrity. A question is how the system will know whether or not the obviously known state which is used as a reference is valid. The MTM standard makes sure to assure this validity. The system measurements are stored in PCRs to verify the state's [59].

5.2.5 Trusted Software Stack (TSS)

The TPM and MTM standards are in our context concentrating on the lower levels of the system design (boot-up process), therefore the TCG introduced a specification named the TCG Software Stack (TSS). The main purpose of TSS according to the TCG is to provide a set of APIs for the Application vendors to reach the TPM or MTM [60]. There are three methods in TSS that an application can use to reach the TPM. One is by using a cryptographic provider (e.g. *PKCS 11*), another is through the *TSS Service Provider* (TSP) and the third is through the *TSS Core Services* (TCS). Applications using PKCS 11 are for example Cisco's VPN or Firefox. TSP is providing all TPMs features to the programmers. The TCS layer is used to allow remote machines to communicate directly with the TPM. There are however other interfaces which can be preferred as other options rather than TSS in order to communicate with the TPM [61]. Even though that TSS is out of the scope of this thesis since it is an Application layer concept, it is necessary to have a knowledge of its existence and a basic understanding of the concept.

5.2.6 Sealed Storage

Sealed storage is the concept of securing private data by binding it to the system configuration, commonly used with the TPM concept. It can be used to make sure that only a certain hardware can be used in combination with specific software. This means that, if a hardware (e.g. hard disk) is moved to another system, the content will not be accessible anymore [62]. Related to the context, since hardware replacement can cause crucial complications for the system security and is commonly not in the scope of the papers proposing secure boot solutions, a sealed storage could be a useful technique to prevent the system from functioning if an essential Root of Trust hardware is replaced involuntarily [63]. TPM provides the *seal* and *unseal* commands where seal uses a key protected by the TPM and specifies certain PCR setting in order for the data to be accessible with unsealing. A drawback with the Sealed storage concept is that, if the PCR configurations change too often, the sealed data could become inaccessible if something went as unexpected [62]. Since PCRs are a volatile type of memories and loaded at each boot-up instance, by measuring and extending (as explained in sub-chapter 5.2.3.2), an update in the system boot-up process code before unsealing all sealed data would cause the TPM to measure and extend new hash values to the PCRs and thereby change the PCR settings. The consequence of this would be inaccessible sealed data.

5.2.7 Remote attestation

The Remote attestation concept, also referred to as *Phenotyping*, is providing integrity and authenticity verification on the data or code from the host device by sending the relevant data to a third authorized party [64]. The design of such a concept can be based on the TCG standards. TCG provides guidelines to implement the protocol used between the host device and the remote device as well as a remote attestation design based on a TPM and its PCR values. [65] [66][67]

5.2.8 Memory curtaining

Memory curtaining is a hardware enforced memory isolation technique which provides confidentiality and integrity protection against unauthorized programs which tries to access a memory it is not allowed to read or write to. This is a good solution for the application level of the system as spy-wares running in parallel with authorized programs cannot reach these program's curtained individual memories. The operating system does not have access to curtained memories either, which prevents virus infected operating systems from affecting the individual memory space of a running program [68].

5.3 Verified boot techniques

The term Verified boot is recently being used in the context of autonomous embedded systems and is decently new compared to the first appearance of the CoT concept. Since Secure boot was the first term used to take up the discussion about a secure boot-up process for PC systems back in 1991 [44] the term is still widely used in the state of the art research papers instead of the Verified boot. Google has been using the term Verified boot to describe a secure booting process within the Android and Chromium OS systems as we saw earlier. This section outline is as follows; we start by presenting the pioneering concept (AEGIS) for the Chain of Trust, which has been used by the majority of the techniques that we present in the remaining part of the section. The consecutive techniques are state of the art, with the aim of maintaining a functionality similar to the Verified boot in various system types. Most of the techniques rely on the TPM concept which was explained in subsection 5.2.3.

5.3.1 AEGIS, the first Chain of Trust technique

In 1997 the researchers W. Arbaugh, D. Farber and J. Smith proposed a new bootstrap integrity assurance method which was going to be the pioneer for many projects in the future. It was named "AEGIS", a guaranteed secure boot-up process which ensures the computer system to recover and start-up even if integrity failures occurs. It has the aim of securing various areas such as Internet commerce, security systems and "active networks" [69].

The authors note that a system cannot be secure without integrity, "*thus, any system is only considered as secure as the foundation upon which it is built*" [69]. The authors divide the boot-up process into several abstraction levels which are the boundaries of the different system layers. They claim that the integrity of a level during a bootstrap process can only be guaranteed if and only if two rules are followed;

1. *"The Integrity of lower layers are checked" [69]*
2. *"Transitions to higher layers occur only after integrity checks on them are complete" [69]*

These two rules came to define the integrity chain (Chain of Trust) which as a whole guarantees the system security, beginning at the power-on stage and continues until that the final boot-up layer hands over the control to the OS.

The authors name the Chain of Trust concept as “chain of integrity checks”, which is represented in the following way,

$$I_0 = True,$$
$$I_{i+1} = I_i \,\&\, V_i(L_{i+1})$$

Where I_i represents the integrity of the level i (assuming that i = 0, 1, 2, ..., which is a sequential execution up to a certain level depending on the system design). V_i is the verification function which takes the argument L_i (which represents the level to verify) and returns a boolean value. The verification function performs a cryptographic hash operation on the level code and compares the output with the stored value for the respective level, thereby maintains its output in form of a boolean value. In case of failures, the system starts booting in a recovery kernel which is in a ROM card. The recovery kernel contacts a remote trusted server through a secure protocol which provides the verified copy of the failed layer. After the repair or replacement of the altered/corrupt component, the system restarts [69].

The recovery methodology can be seen as a remote attestation technique, but instead of only a verification, the system is recovered.

5.3.2 Verified Boot using U-Boot

This method is a software-based solution which is a feature included in Das U-boot. The authors to this subsection is, however, the authors of this thesis report, where the information is gathered during the implementation phase of the technique on specific hardware platforms.

The bootloader, U-Boot, aims to verify that the integrity of the kernel has not be tampered. In order to implement this method, several assumptions should be made. Firstly, the pre-boot environment of U-Boot has to be trusted and considered as what is mentioned as the Root of Trust in the definition of the Verified boot. This means that the security of the boot stages prior to U-Boot is not included within the scope of this method. Secondly, the U-Boot has to be loaded into a read-only memory since there is no prior verification of the booting process. Lastly, there must be an appropriate policy to store and manage the cryptographic keys which are produced through the method. The system designer has the responsibility of the security for the last mentioned policy.

Some other key points include the software update process. The kernel and other signed (by the private key) firmware can be flexibly updated. On the other hand, to update the bootloader (U-Boot), it is necessary to reconfigure the updated version with the public key. Another point is, when changing the security level in terms of different size on the key pairs, this can cause a chain of reconfiguration to the whole U-boot stage.

A simple overview of a general process for the implementation of verified U-Boot can be seen in figure 5.2.

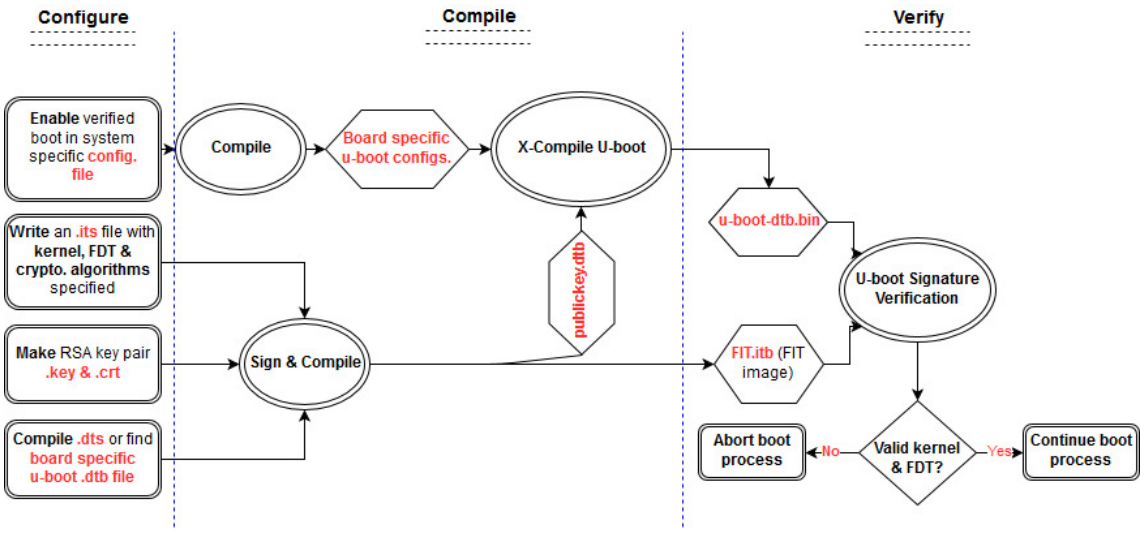

Figure 5.2: Implementation process of verified U-Boot

Figure 8.1 is divided into three phases; *Configure*, *Compile* and *Verify*. The configure phase shows the files which are configured and keys created which is a phase to prepare the necessary files needed for the compile phase. In the compile phase, it is possible to see that the main goal is to manage to compile U-Boot with the verification feature enabled. The maintained U-Boot binary file will be specific for the specific hardware device it has been compiled for, so only a general (universal) view is presented in this image. There will, however, be more specific descriptions for the used hardware in upcoming chapters.

The last section presents the verification process. A U-Boot binary which contains the public key is maintained as well as a FIT image which contains the configurations of how to verify the kernel image and a hardware specific device tree blob (dtb) file. The FIT image has been compiled to contain these two files (kernel image & dtb file) which mean that these files are no longer necessary in the boot partition separately and can thereby be removed.

During the compilation process of the FIT (Flattened Image Tree) image, there is an option (-r) for the image creation tool *mkimage* to enable the feature of requiring U-Boot to verify the images in order to continue the process of booting up. Since this feature is enabled in our case, a faulty verification process would lead to

the termination of the boot-up process.

5.3.3 TPM based techniques

This subsection, as the title implies is presenting some of the state of the art techniques available using the TPM standard to achieve a Verified boot sequence.

5.3.3.1 Hardware Security Module based on TPM

A TPM based system design is proposed in [70], the approach is based on a two-processor design where one of the processor's functions as an application processor and the second one as a secure co-processor. The secure processor is a Hardware Security Module (HSM) which implements the TPM standard. The trusted boot as the authors calls it, relies on the Root of Trust concept where the Root of Trust for Measurement (RTM) is a boot ROM located on the application processor.

The authors differ Secure boot from Trusted boot as follows, Secure boot relies on certificates from certificate authorities while trusted boot measures the hash value and provides flexibility to the system designer to chose how the system shall act (or user of the device in case non-autonomous embedded systems). By referencing to cases where systems with Secure boot has been exploited by malware which has successfully used signatures of stolen/misused certificates, the authors present their design as a minimal functionality of Trusted boot for embedded systems. The implementation is said to concentrate on software-based attacks with only a minimal measure against physical, hardware-based attacks.

A common model of Chain of Trust is applied during the boot process which verifies the integrity of the different layers step by step using a hash value. The procedure consist of a **Measure** -> **Extend** -> **Execute** process. The hash value of a boot-up stage image is **measured** in the TPM and consequently **extended** (hashed together with the current content in the PCR) and the code for the layer is **executed** if the extended value is successfully verified to a trusted reference value.

The HSM which has implemented the TPM specifications provides cryptographic operations for the application level. The TPM implementation is at a minimal level, to keep the system optimized. It is direct as a bare metal approach without an operating system. Hence, the applications with keys to reaching the TPM functionality can request specific operation. The validity of the key is confirmed or denied by the HSM since it stores and generates these keys. If the validity is denied for a specific application, the requested cryptographic operation will not be provided (see Sealing in subsection 5.2.6).

Since the implementation is a minimal functionality of trusted boot as mentioned earlier, the authors have not included remote attestation in the design, they do however mention that this functionality can be added without major complications since the HSM component supports the necessary functionality.

The boot-up process in this design is presented in Figure 5.3 and described step by step below.

Boot process:

1. The HSM processor and the Application processor (which contains the RTM) are started in parallel
2. HSM sends a ready signal to the Application processor when the initialization stage is completed. The HSM will wait for a response before taking the next step.
3. The Application processor can consist of several cores, in any case, the first core starts the Chain of Trust process by verifying each layer in the boot-up process while the other cores are staying idle.
4. The idle cores are activated at kernel start-up.

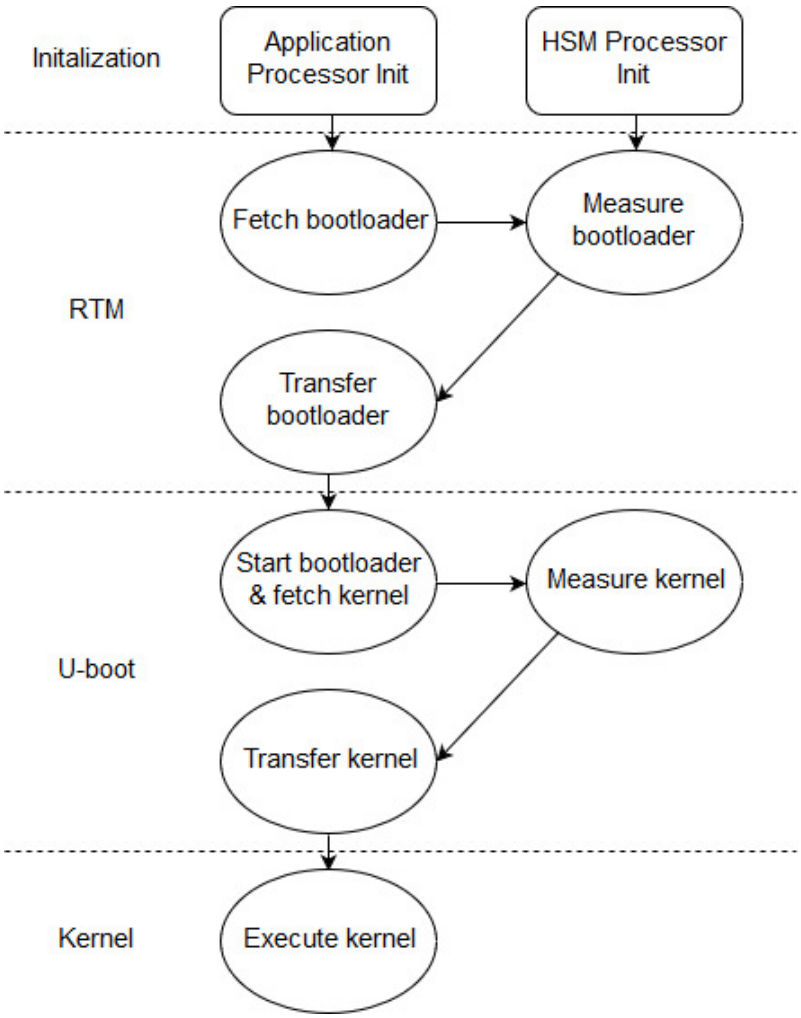

Figure 5.3: Bootup process with a HSM processor

This design is according to the authors not providing any protection against physical attacks or side channel attacks. It is rather only a remote attack security measure

as mentioned earlier.

5.3.3.2 Portable TPM (USB Key)

An extraordinary method of providing a secure boot (for a PC) system has been proposed in [68] where a USB key is used with the Unified Extensible Firmware Interface (UEFI) system. The paper is in turn based on an older paper which is the pioneer in the portable TPM concept [71] which is implemented with the older versions of the TCG specifications. The USB key is also implementing a portable TPM functionality and is designed as a secure chip. This technique can potentially be adapted to a main embedded system in the automotive industry by minor design modifications.

The main idea is to store the EFI System Partition (ESP) content of the UEFI design which consists of the bootloader program, driver files and other essential code used in the booting process, into the USB memory. The boot process is visualized step by step in Figure 5.4, where Security (SEC) and Pre EFI Initialization (PEI) are considered as the RTM of the system (the initial codes executed in the system and are considered to be non-modifiable).

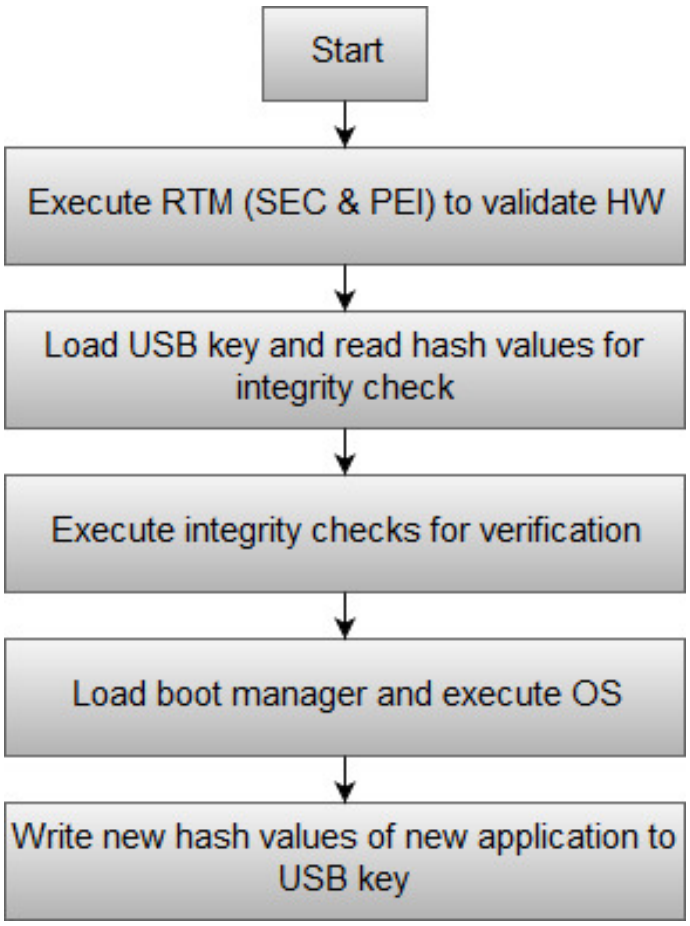

Figure 5.4: Bootup process with USB key

As a secure chip, the role of the USB key is to provide cryptographic encryption and decryption features. The memory curtaining concept is used in order to provide integrity and confidentiality to the RAM in the main system where data from the USB key is being transferred to. The Memory curtaining concept (see subsection 5.2.8) is used in the USB key memory space as well. This is due to the fact that the ESP is approximately 100-200MB in size whereas the memory capacity in a USB normally starts with alternatives of 2GB and keeps growing further. Hence, partitioning is necessary and the isolation between the partitions are provided by memory curtaining.

Finally, in order to not allow any unauthenticated USB key to be able to boot the specific system, the Sealed storage concept (see subsection 5.2.6) is used where cryptographic keys are used for authentication between the USB key and the system. These keys are generated at every boot-up instance and are based on the specific USB key and the respective system which the USB key is intended for.

5.3.3.3 ARM TrustZone based TPM

The ARM TrustZone technology is a brand specific concept which is the reason for why it has not been included in the general concepts section (Section 5.2). The basic idea is to provide a System on Chip (SoC) with Secure and Non-Secure (Normal) worlds. The switch between these worlds is provided by a software named a Secure Monitor. The concept is however not only restricted to the processor, but rather extends the memory, software, peripherals, bus transactions, and interrupts [72]. These and all other hardware and software components are partitioned into either of the worlds. All components which require high security are included in the Secure world while all remaining components are in the Normal world. The method used to isolate these two worlds from each other (hardware-wise) is implemented in the hardware logic of ARMs TrustZone enabled devices. ARM solves the partitioning issue on some of their processor cores within one single physical processor with a time-sliced method, which benefits the system by lowering the area usage and power consumption while keeping the performance at a high-level [73]. There are other brands who have developed techniques to maintain similar security goals as the TrustZone, examples are AMD Secure Execution Environment and Intel Trusted Execution Technology.

The main goal of ARM TrustZone is to provide a Trusted Execution Environment (TEE). This hardware environment is supposed to provide confidentiality and integrity protection to the system.

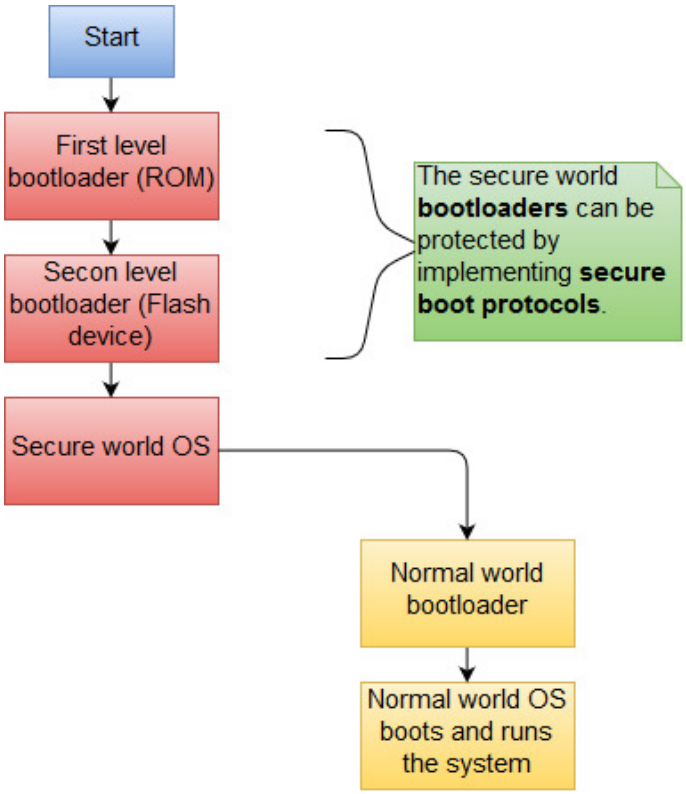

Figure 5.5: Bootup process in an ARM TrustZone processor

Some papers relate to ARM TrustZone in their research for proposing security between the Normal and Secure applications by virtually running different operating systems (e.g. [74]). The main purpose of this section is, however, to provide security during the boot-up process by using the ARM TrustZone.

As seen in Figure 5.5 the boot process is initiated in the Secure world and thereby prevents the Normal world (untrusted) software from interacting with the boot-up process. The Verified boot process is based on the Trusted Computing (TC) concept with an RTM placed in a ROM which is the first level bootloader in this case. The procedure is following a CoT model from there on in order to reach a fully functional system. The mentioned "secure boot protocol" in Figure 5.5 could be seen as a model such as the CoT supported by a TPM. As seen in this case, the TrustZone technology mainly concentrates on the application level security but also provides additional hardware and software security to a Verified boot process supported by a TPM.

Some proposed methods to prevent a certain type of attacks such as shack attacks or man in the middle attacks (when data/code is copied between memories) are presented in [73]. A method to prevent shack attack is to keep the execution of secure world resources in the SoC memory locations since it is a lot more difficult for an attacker to obtain data from these locations compare to an open Printed Circuit Board (PCB).

The designer has to be careful to not provide open windows for the attacker when the code is copied from one location to another. So the cryptographically hashed images in the SoC storage shall be authenticated after being copied to the secure location. If not, the man in the middle attacker will have a short window opportunity to modify the data after being authenticated and while being copied.

As a conclusion, we can observe that the ARM TrustZone concept is rather an additional security measure for specific ARM hardware than a general Verified boot technique.

5.3.4 RIM certificates with software MTM

The technique presented in [41] is based on secure boot which aborts execution as soon as the integrity check fails. The authors mention the existence of a hardware based MTM which is a smart card connected via a serial interface. It implements a minimal version of the MTM and TPM standards and is programmed in Java. The explained software based MTM, on the other hand, is an open source MTM emulator provided by Nokia [41].

The analyzed system relies on a ROM which gradually measures the images and checks the integrity of the next step by using its RIM certificate embedded in its image. To report the collected measurement values, the bootloader needs to access the MTM. The MTM can either be a software based MTM integrated into the bootloader or a hardware based MTM. The paper proposes a low-level API layer to provide the access to the MTM.

An MTM can be designed in several ways since TCG only sets requirements and feature for the MTM but not how to implement it. The paper proposes a high-level API layer for the applications (which starts after the boot process is completed) to reach the services provided by the trusted computing implementation.

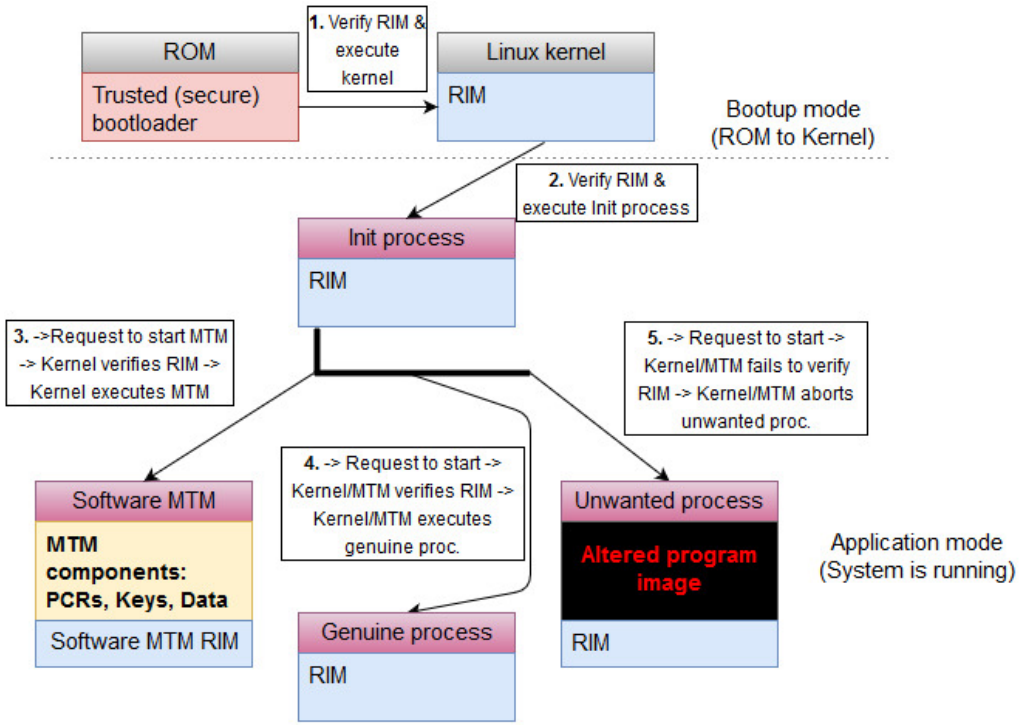

Figure 5.6: Secure boot process with RIM & software MTM. Stage 4 and 5 can either be verified and executed by the kernel or the MTM.

As seen in Figure 5.6 the software based MTM is proposed as an user-space process which starts as the first child process of the init process, thereby running under the OS kernel. As mentioned earlier, the MTM is used to (among other duties) store measurement values. However, as mentioned in Figure 5.6 the MTM could be used by the kernel to verify RIM certificates.

The software image RIM certificates could be stored in a central storage, which would cause a single point of failure. To solve this problem, the paper [41] proposes a decentralized approach where each RIM certificate is embedded into respective software images they belong to. This is achieved with the mechanism for embedding certificate meta-data into binary images, the ELF binary format, as seen in Figure 5.7.

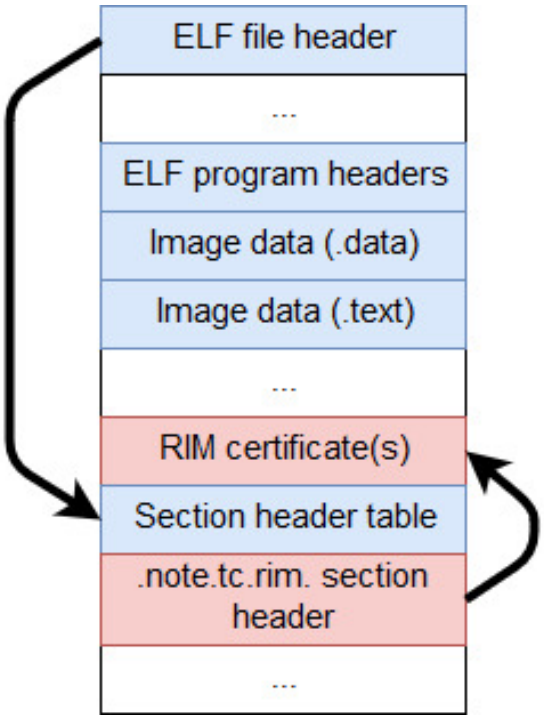

Figure 5.7: RIM certificate(s) embedded into ELF formatted software image.

When measuring the software images in the process of verification, the RIM certificate is ignored and the remaining ELF image is measured. The "notes" section in Figure 5.7 is set to binary zeros before measurement, however, in the final form of the image, this section contains the RIM certificate. The paper also proposes to include the RIM verification keys into the software images. Thereby, the verification key needed for next stage can be extracted from the current stages binary executable file image. To load the verification keys into the RIM certificates, the executable image has to be verified. As a consequence, if the RIM certificate for a "parent" stage becomes invalid, all of the verification keys inside that executable image becomes unavailable for the boot process, this can be seen as a CoT. In the proposed software MTM approach, the MTM starts as the first child process of the init process. So the secure boot process is described in the following way:

Boot process:

1. The CPU directly jumps into the secure bootloaders entry point.
2. The bootloader verifies the Linux kernel. The verification key is bound to the bootloaders image.
3. The Linux kernel verifies and possibly boots up the init process in a same fashion as the bootloader verified the kernel.
4. The Init process can now start the software MTM. The MTM can now be used for the remaining security functionality and collective verification of the previous stages as seen in Figure 5.8. Each stage that was verified and loaded during the boot process has to hand over their measurements to the MTM and be verified. If any error occurs during this process, the boot process gets

aborted.

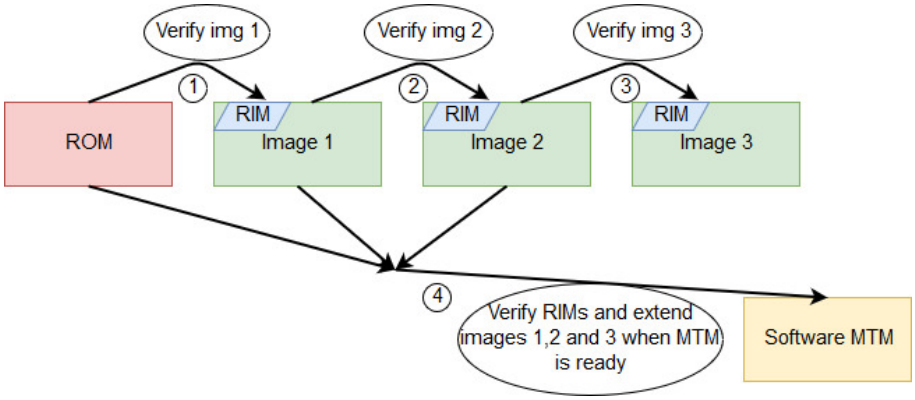

Figure 5.8: Secure boot process with a software MTM.

Hence, the paper presents a software based MTM solution as a verification method to maintain secure booting. This is achieved by using a trusted verification method during boot-up and a reverse verification after that the software MTM is executed.

5.3.5 Verified boot in a FPGA system

The FPGA Integrated Circuits (ICs) have been considered to have lower speed/performance, higher complexity, and higher volume compared to ASIC ICs. The performance of FPGA ICs however, have been increasing lately [75]. The authors in [42] propose a security chain starting at the bitstream up to kernel booting process. Thereby, the integrity of the boot-up system will be preserved and also include countermeasures for replay attacks.

The bitstream is another way of expressing the update-data in this context. The security of the bitstream over an untrusted network is provided by implementing a specific protocol between the system designer (SD) and the target host FPGA system (see Figure 5.9). The method is intended to prevent replay attacks but can also be used to provide a secure remote attestation functionality.

As seen in Figure 5.9, the FPGA side of the communication consist of three parts, the *static logic*, *user logic* and *user flash*.

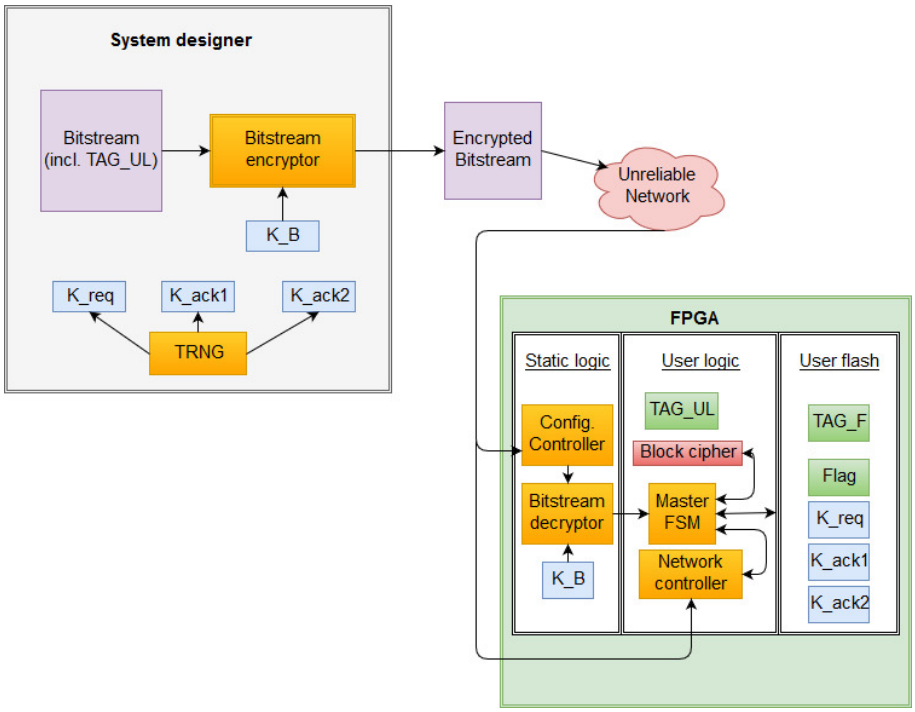

Figure 5.9: Bitstream protection and replay attack prevention over untrusted network.

The static logic is, as seen by its name, hardwired and cannot be changed. Its main purpose is to provide confidentiality and integrity protection for the bitstream using the symmetric key K_B, which is shared between the SD and the FPGA system.

The user logic has the duty of verifying the bitstream version (updated version). This part can be configured by the SD in contrast to the static logic part and is designed as a *Finite State Machine*. The different components which the user logic consist of are; TAG_UL which indicates the current version of the bitstream, block cipher to authenticate the SD and FPGA to each other, network controller which can enable the remote update functionality and NVM controller to store various essential data for the entire protection process.

The user flash (NVM) contains three keys for encrypting tags and the TAG_F which indicate the version of the received (new update/) bitstream, it can only be changed by the SD and is compared to TAG_UL. The keys are unique for each FPGA which means that the SD has three keys for each FPGA it manages. Only the user logic can read and write from the user flash.

Bitstream verification process:

1. An update command with incremented TAG_F is sent from the SD to the FPGA encrypted with K_req.
2. FPGA decrypts and compares it to TAG_UL. If the received value is differ-

ent, FPGA awaits new commands, if same, the FPGA increments TAG_F, encrypts new tag with K_ack1 and sends it to the SD.

3. The SD sends ciphered bitstream to the FPGA which is validated with design specific methods in order to protect the FPGA from unintended (malicious) bitstreams. The FPGA acknowledges this with a new tag sent back encrypted with K_ack2 with the new bitstream design installed. Installed bitstream now contains the incremented TAG_UL which should be same as the TAG_F.
4. The FPGA compares TAG_UL and TAG_F on each start-up. If they differ from each other, the system sets off an alarm for detecting a replay attack and the SD is notified to take desired design specific actions.

The next step is to execute the bootloader and Linux kernel in a verified fashion. The bootloader code is located in a block RAM whose execution is initiated by the bitstream (see Figure 5.10).

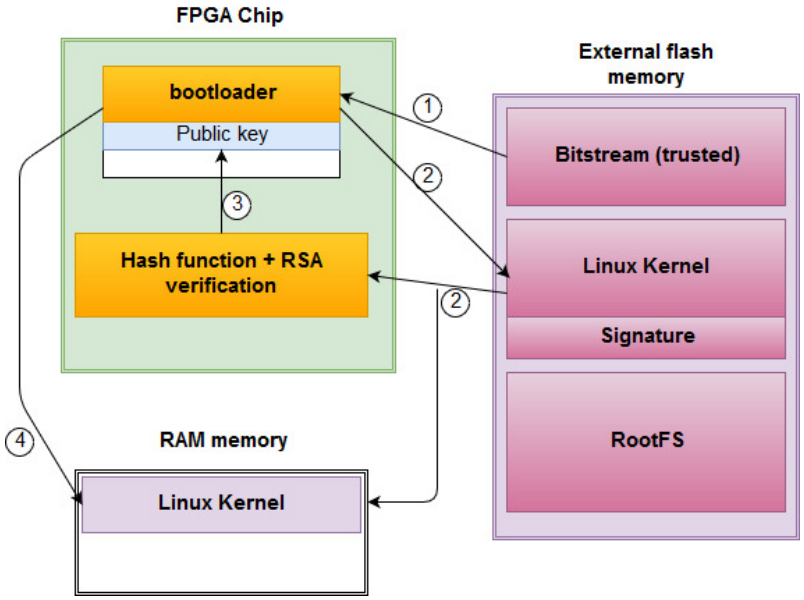

Figure 5.10: Bootup with integrity verification on Xilinx FPGAs

The proposed method is claimed to be flexible due to the fact that the kernel code can be changed in the external memory without having to download new bitstream or reprogram the FPGA. The kernel provider has the private key to sign any new versions of the kernel and the bootloader has the public key of the asymmetric key pair to verify any new versions of the kernel. Thereby, each time the kernel is updated, the bootloader code can stay the same since the same key pairs are used, in contrast to not using asymmetric keys but hash values instead, then the reference hash value would need an update in the bootloader which would mean that the bootloader had to be updated each time the kernel was updated. The steps using asymmetric key cryptography for verification are explained below.

Boot process:

1. The bootloader is loaded into the block RAM of the FPGA chip (from the bitstream in external flash memory) and contains the public key for the verification of the kernel code.
2. The kernel provider creates a hash of the kernel and signs it with the private key and places it in the flash memory after the kernel code.
3. The Hash core (on the FPGA chip) generates a hash value of the received kernel code and the bootloader verifies this value with the signature by using the public key stored in the block RAM.
4. The bootloader starts executing the kernel code after verification.

As previously mentioned since asymmetric cryptography is used, any update of the kernel code will not affect the bitstream or the data in the block RAM, as the keys for verification remains the same. This technique is however vulnerable to replay attacks. A proposed method of preventing replay attacks is to make use of the *Merkle tree* technique which creates a hash value of the entire memory hierarchy in a trusted chain fashion. The non-volatile top hash value of the memory hierarchy can be stored in the FPGA chip to validate that the transferred data is genuine.

Another boot-up process for an FPGA chip is proposed in [70]. The paper presents the same design proposed in subsection 5.3.2.1 but adapted to the FPGA architecture (see Figure 5.11).

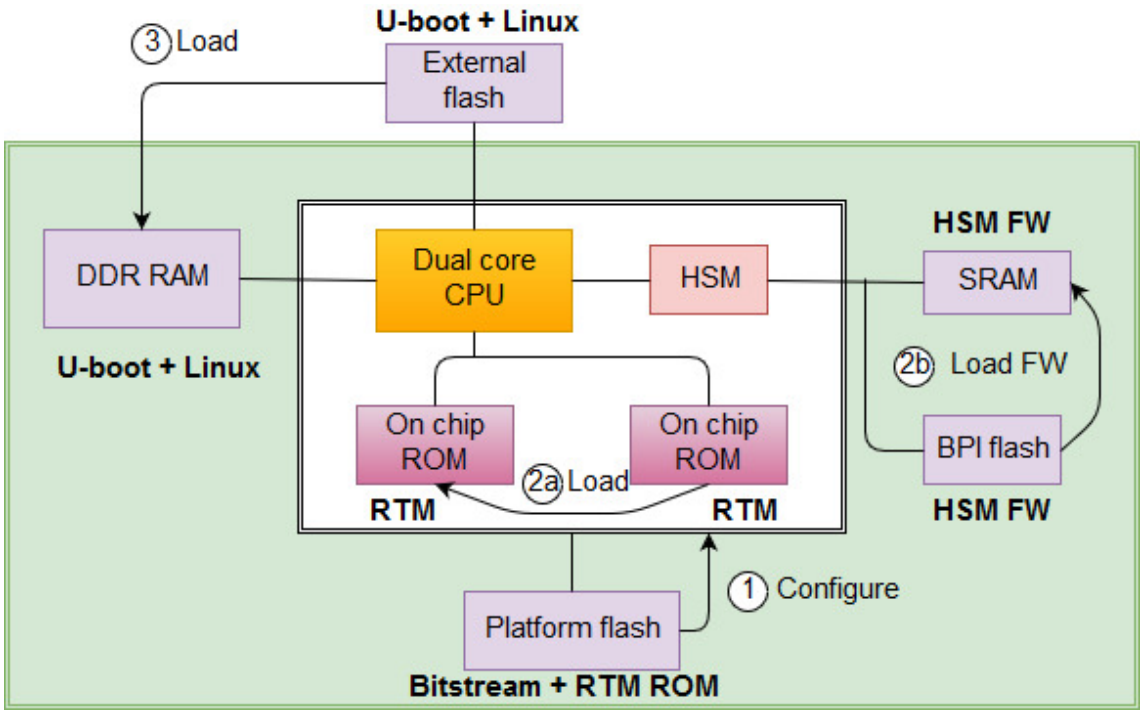

Figure 5.11: The FPGA components and steps taken during the boot-up process

Boot process:

1. Initially the FPGA loads in the hardware configuration from the platform flash memory. The configuration also includes the RTM code.
2. The Hardware Security Module (HSM) executes its firmware code in the SRAM due to a capacity limitation in the block RAM, which it fetches from the Byte Peripheral Interface (BPI) flash.
3. The HSM can now initialize the TPM specification by verifying different TPM specific functionality.
4. HSM signals the Application core that it is ready to serve.
5. The Application core fetches the RTM code from the block RAM to immediately execute it relying on its trustworthiness.
6. RTM fetches the bootloader binary (U-Boot in this case) from an external flash to store it in the DRAM.
7. The RTM performs the measure, extend and execute the process (see 5.3.2.1 for more details about this process) on the bootloader binary by piping it from the DRAM to the HSM.
8. The HSM performs an SHA-1 measurement on the bootloader binary and extends the retrieved hash.
9. If the hash value is approved, the HSM allows the execution of U-Boot.
10. The same procedure as in step 6-9 is performed by the Linux kernel, but now, instead of the RTM, the bootloader is controlling the process.
11. The kernel is booted if the verification was successful.

5.4 Conclusion of Verified boot

As seen in the technique descriptions, all of the techniques, whether it is a software based or hardware based technique, are built on top of the RoT and CoT concepts. We can see that the hardware provider plays a vital role in the design phase of a system with Verified boot.

Most of the techniques assume a boot ROM as the RoT which in most cases are pre-programmed by the hardware providers. A first stage bootloader is normally considered as a minimal and simple code. Thereby, a verification feature for this stage is mostly unfeasible to add if it already was not designed by the hardware manufacturers.

We can conclude that designers who design similar techniques have to follow one general definition of a secure boot-up solution (Verified boot) in order to not cause confusion, and at the same time ease the process of implementing such techniques to their systems.

Since the complexity of the boot process stages is increasing as they reach closer to the final stage and since many techniques are designed in an ad-hoc fashion on-top of inherently non-secure systems, we can see that the opportunities for disabling features or optimizing these stages with various techniques can be an option. In the next section, we will evaluate optimization techniques to lower the boot-up time.

6
Boot time optimization techniques

Most systems where user interaction is present often aims to boot up the system to a state where graphical and/or audible feedback is quickly presented. The aim is to make basic functions available while the rest (non-crucial part) of the system is initializing in the background. In the case of embedded systems, another system is normally using the embedded system in contrast to a user interacted system. In this case, the tasks which are most relevant for the interacting system have to start running as soon as possible to reflect a fast boot time of the embedded system. It is up to the system designer to prioritize the tasks to be available as soon as the system is powered on.

The design phase is however where the optimization starts, the second aspect is to chose powerful hardware. The hardware is directly related to extra expenses and area/heat limitations, so it is normally already chosen as in the case of our thesis project. Our area of focus is getting limited to optimizing a system with already chosen hardware and designed system. The only options remaining are to find techniques to disable non-essential functionality, possibly slight modifications in different software areas and compromise between security and boot time to maintain a faster boot time.

6.1 System optimization

The software optimization alternatives below are presented for Linux based systems since Linux is the main alternative for embedded systems [76] and U-boot. The hardware based optimization techniques are mentioned thoroughly since they are dependent on an available budget to spend on the system and varies with different hardware properties.

6.1.1 Software based techniques

I. Reduce the kernel size

The generic Linux kernel has a vast collection of default configuration settings for a variety of features, all of these are not necessary for most systems, especially not embedded systems. To spend time as a designer on marking out and disabling the unnecessary features can save a lot of time in the boot-up process (up to 3 seconds or more) [76]. Less critical but necessary features can be converted to loadable kernel modules to later on (after the kernel boot process) be loaded. Examples of

such features may be, IPv6, support for various file-systems or device drivers (which requires time for initialization) for devices which are not used [76].

II. Uncompressed kernel image

The decompression time of the kernel image can increase the boot time by several seconds in some systems [76]. This is however in balance with the type of the non-volatile memory in the system. The reason is that having an uncompressed kernel image will take a longer time to read from the non-volatile memory, however, to have a compressed kernel image will require time to decompress [77]. Depending on the type of the non-volatile memory in the system and the kernel size, to have an uncompressed image can yield a shorter boot time. The compression algorithm has also a big impact on the boot time. Test results in [77] shows a significant boot-up time difference, nearly 870 ms, between two different compression algorithms.

III. Final stage bootloader (U-boot)

When optimizing the final stage bootloader, which is U-boot in our case, the relevant parts to concentrate on are bootloader binary size (can be reduced before cross-compiling), environmental variables and scripts/configuration files (which are used to automatically load the necessary files).

The essential approach for optimization is to remove features which are unnecessary for your system's purpose. U-boot provides configuration files specific for hardware platforms which enable different features. All of these might not be useful and can be disabled before cross-compiling and thereby both reduce the size of the U-boot binary and potentially also shorten the U-boot initialization time.

Another way of optimizing is to disable features in the U-boot environment variables. These can be displayed after booting up U-boot by typing the "*printenv*" command and changed with the "*setenv*" command. Changes done in the environment variables are only temporarily available. In order to keep these permanent, all the new environment variables can be saved to a file by using the "*saveenv*" command.

A third option for optimization in U-boot is to choose the best automation option (in terms of loading the next stage). U-boot has by default a feature which can read files ending with "*boot.scr*" or "*boot.scr.uimg*". These script files are read to modify the environment variables as specified in the script file. Another option is to type the desired configurations in a "*uEnv.txt*" file, this option is however not supported by all boards. A third option is to manually change the environment variables by using the above-mentioned commands to create a "*uboot.env* file which is protected by a CRC32 checksum and by default read by U-boot when getting initialized.

IV. Remove initrd

Initrd is essential for larger generic systems but might be eliminated when considering embedded systems where the amount of device drivers are limited and the

function of the system is deterministic and simple. The functionality of initrd can be hard-coded into the kernel image. Removing initrd would as well result in decrease of the size of the kernel image and thereby lower boot time [76].

V. Selecting filesystem

To remove initrd might be effective considering the boot time as previously mentioned, however, to have an initial filesystem (i.e. CRAMFS) which makes it possible to start critical tasks might be necessary to decrease the boot time. The approach to maintain this efficiency is to have a different filesystem for less critical tasks which will be mounted after the boot-up [76]. Some statistics on different file systems are presented in [77] where no significant difference is noticed. The authors of [77] do however mention that they have only used one configuration tool while gathering the data.

VI. Timer loop-calibration

Timer loop-calibration is a process executed during the boot-up by the kernel which counts the amount of loops that can be iterated during a certain amount of time called jiffy. This argument is however constant for each processor, which means that it can simply be transmitted to next boot occurrence statically. Thus, this calibration process can be skipped by using the command lpj as a boot argument [77].

6.1.2 Hardware based techniques

Hardware based techniques are mainly related to the choice of memory, processor and cryptographic co-processor (if one is used). The choice of a better processor and the maximum limit it can be clocked to is an obvious factor which will have an impact on the boot time. However, embedded systems are normally limited by size as well as energy consumption which is the reason to why the discussion about alternative memory hardware and cryptographic co-processors becomes more interesting.

I. Choose of memory

A research has been made over the choice of fast non-volatile memories and its impact on the boot-up time for real-time systems in [78]. A noticeable outcome is the fact that flash memories using the NOR logic (NOR flashes) are significantly faster than NAND flash memories. The cost of NOR flashes is however relatively higher. In [78] we can see a comparison with respect to price and performance of the above-mentioned memory types in comparison to a volatile DDR2 memory. The results present approximately 100 times faster random access read and 150 times faster write performance for NOR flashes compared to NAND flashes, the price of NOR flashes are however nearly 15 times higher. These two non-volatile memories are however significantly slower performance-wise compared to a volatile DDR2 memory.

II. Cryptographic co-processor overheads (HSMs/TPMs)

When considering the overhead of a cryptographic co-processor, or so-called a Hardware Security Module (HSM), the technique mentioned in subsection 5.3.2 is a good example. The major part of the secure boot time overhead (17.93 sec) is caused by the HSM firmware initialization which cannot be parallelized with the application processor since the Chain of Trust layers is dependent on each other. Hence, a design restriction prevents most possibilities of optimizing the boot time related to the additional HSM component. An option, in this case, can be to chose a more modest design on the HSM to minimize the overhead as much as possible. [OnImp].

6.2 Conclusion of Boot time optimization techniques

There are various methods and options to optimize a system as seen in previous subsections, ranging from software to hardware methods. The key point here is to know the purpose of the system, only necessary functionality shall be enabled in order to keep small size on files and short boot-up times.

In the next chapter, we will see an evaluation of experiments (proof of concepts) and how an optimization technique can be applied to have a radical impact on the boot-up time of an embedded system.

7
Experimental evaluation

An experiment is a crucial part of a scientific research method within many areas of computer science. The reason is that solely theoretical research would not be enough by itself in such complex hardware systems. After concluding in Chapter 4 that the security measures need to cover the boot process to prevent Rootkits & Bootkits, we evaluated different secure techniques in Chapter 5. The next step is to present proof of concepts, one as a software- and one as a hardware-based technique to reflect the impact of a verification overhead in a clear manner.

The specific methods which are used in sub-chapter 7.2 are not only chosen with respect to their hardware and software property but also with respect to the resource and time limitations. With the feasible experimental availability, the scope was set to maintain security in part of the boot process in the experimental setups.

With our experiments, we are looking for an acceptable trade-off between different security measures (in terms of cryptographic algorithms and a secure co-processor) and the time impact to the boot process. In sub-chapter 7.3 suitable optimization techniques were used to minimize the boot-up time as the compensation for the added security time overhead.

7.1 Experiment setup

In this sub-chapter, we describe the hardware and their properties that we used to perform our experiments. We choose generic embedded platforms that are easily available, namely a Raspberry Pi and a BeagleBone. Moreover, those platforms are commonly used for prototyping by thousands of projects in the area of Internet of Things (IoT), which also reflects the expansion of the embedded systems connected to the Internet. Finally, those platforms are considered for prototyping also by engineers in the automotive industry and we believe the hardware capabilities look alike of an ECU.

7.1.1 Raspberry Pi 3 Model B

Raspberry Pi was a project started in the UK to support young people in their quest for increasing knowledge within the field of computer science. The usefulness in the industry, research areas as well as private projects kept increasing and several

version of Raspberry Pi's are now produced in quantities reaching millions [79].

The device is a low power consuming small computer which can either be used as an embedded device or a self-operating independent system. Raspberry Pi 3 Model B is the third generation Raspberry Pi which is as of 2017 the latest model available [1]. It is well suited for simple experiments and researchers, however for more complicated researchers, the documentation support is not as broad as expected which leads to more time spent understanding the low-level concepts as well as writing own modules depending on the project it is used in.

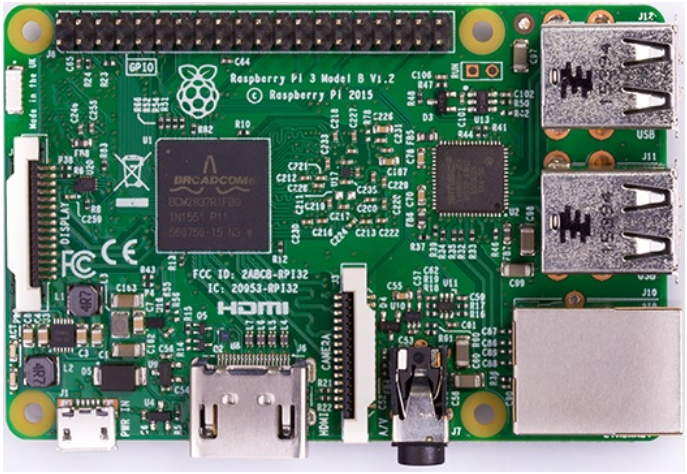

Figure 7.1: Raspberry Pi 3 Model B [1]

Table 7.1: Raspberry Pi 3 Model B specifications [1][4][5]

Processor & memory	**Connectivity**	**OS**
-1.2GHz 64-bit quad-core ARM Cortex A53 (ARMv8) -VideoCore IV 3D graphics core -Broadcom processor BCM2837 -1 GB RAM LPDDR2	-4 USB ports -40 GPIO pins -Full HDMI port -Ethernet port / -802.11n Wireless LAN -Bluetooth 4.1 / -Bluetooth low energy -MicroSD card slot	-Raspbian (Debian Jessie with Linux kernel v 4.4) -Ubuntu -Windows 10 IoT Core -RISC OS

7.1.1.1 Boot process in Raspberry Pi 3

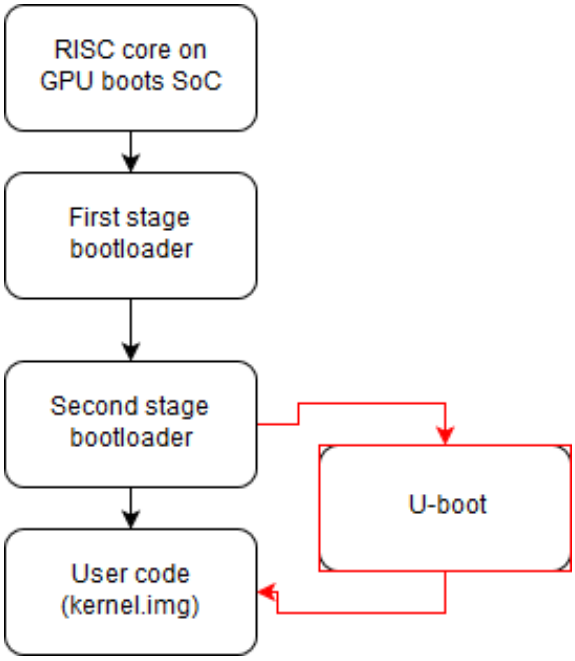

Figure 7.2: Raspberry Pi 3 Model B boot sequence [1]

The Raspberry Pi 3 starts its boot process with a RISC core on its GPU. The CPU is offline at this stage and will remain offline for the most part of the boot process. The default boot process does not contain a U-boot bootloader. Since we will be using U-boot in order to verify the kernel image, we have added it as a component of the boot process. The boot process can be explained step by step as following;

1. First stage bootloader (located in the SoC ROM) mounts the FAT32 boot partition to a MicroSD and loads the 2nd stage bootloader to L2 cache in the SoC. The first stage bootloader is what we refer to as the Root of Trust in Chapter 5.
2. Second stage bootloader then retrieves the GPU firmware from the MicroSD card and initiates the GPU. This bootloader can be found in the MicroSD card boot partition named as "*bootcode.bin*".
3. As soon as its firmware is loaded the GPU starts up the CPU and fix the SDRAM partition to function between the GPU and the CPU [80].
4. The CPU executes the U-boot binary or the kernel.img file.
5. If the U-boot binary was executed, as in the case for Verified U-boot (see subsection 5.3.2), the FIT image (which contains the kernel image) has to be loaded from the MicroSD boot partition into the RAM memory of the Raspberry Pi. A load address is specified and after the load and the verification process, U-boot can be prompted to execute the kernel image.

In tables 7.2 and 7.3 we presented the performance results of base case setup boot time using the Raspberry Pi 3. For each different configuration we run five test to measure the average time of the boot process.

The first table (Table 7.2) presents a base case configuration of U-boot which boots the kernel binary image file directly. In the second base case which will be the base case for the Verified U-boot tests (Table 7.3), we present a U-boot configuration with Verified boot enabled and a test configuration with the kernel binary image and flattened device tree file embedded into a FIT image (.itb).

The details are as follows:

- kernel image: **.img** (generic Linux kernel binary image)
- kernel version: 4.4
- U-boot version: 2017.05

Table 7.2: Boot time performance of U-boot and kernel on Raspberry Pi 3

test#	test1	test2	test3	test4	test5	Average
U-boot	0,881	0,883	0,882	0,882	0,882	0,882
kernel	1,781	2,244	2,826	3,316	3,380	2,709
Total	2,662	3,127	3,708	4,198	4,262	3,591

The kernel time variation is noticeably significant between **test1** to **test3**. The reason for this could not be analyzed due to lack of time. However, a reason can be a backup routine being initialized since the system was shut down by disabling the power source rather than properly shutting down the system from the terminal or graphical userspace. The suspicion for this reason is since the first boot-up time was always the lowest one when the MicroSD card was freshly inserted into the Raspberry Pi 3.

- kernel image: Embedded in **.itb** (Flattened Image Tree file)
- kernel version: 4.4
- U-boot version: 2017.05

Table 7.3: Boot time after using FIT Image without hashing on Raspberry Pi 3

Test#	Test1	Test2	Test3	Test4	Test5	Average
U-Boot	0,923	0,922	0,922	0,922	0,921	0,922
kernel	2,786	2,723	2,285	2,698	2,781	2,655
Total	3,709	3,645	3,207	3,620	3,702	3,577

7.1.2 BeagleBone Black

BeagleBone Black is an open hardware general-purpose embedded device and has been designing by Texas Instruments [2]. The main purpose of the device is to provide a low-cost & low-power board for educational purposes. It is targeting

Universities and any individual interested in Internet of Things (IoT). In table 7.1 we list the hardware specifications of the board.

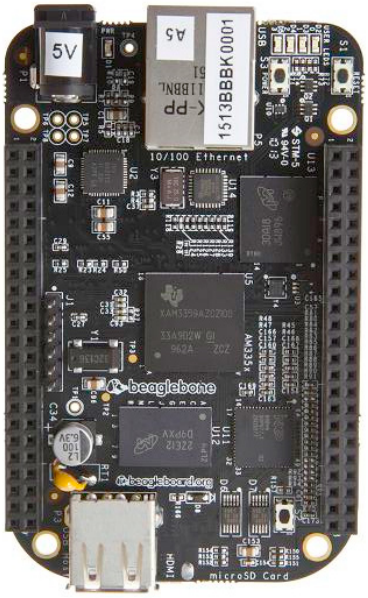

Figure 7.3: BeagleBone Black [2]

Table 7.4: Beglebone Black Specifications

Processor	**Connectivity**	**OS**
AM335x 1GHz ARM® Cortex-A8 512MB DDR3 RAM 4GB 8-bit eMMC on-board flash storage 3D graphics accelerator NEON floating-point accelerator 2x PRU 32-bit microcontrollers	USB host Ethernet 2x 46 pin headers HDMI	Debian Android Ubuntu

7.1.2.1 Boot process in BeagleBone

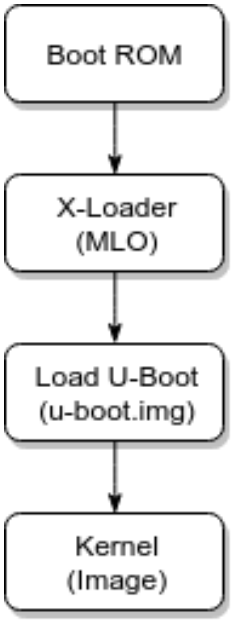

Figure 7.4: BeagleBone Black boot sequence.

Figure 7.4 presents the boot sequence of the board. The first stage of the boot process is to execute the code in Boot ROM. This code is predefined and will run automatically after powering on the device. Moreover, the manufacturer has hard-coded the code to the ROM, and it cannot change. According to the Texas Instruments reference manual [81], the Boot ROM performs the follow functions:

- Configure the CPU
- Initialize the essential peripherals

Therefore, the device has been prepared to execute the Secondary Program Loader (SPL). Next, the Boot ROM will try to search, find and load the SPL into the RAM. Finally, the control is passing to the SPL, and the CPU starts to execute commands from the SPL program.

The second phase of the boot sequence is controlled entirely by the SPL, frequently referred as Mmc LOader(MLO). The main purpose of the code is to prepare and initialize the device to execute the 3rd stage boot-loader (U-boot); there is very limited configuration to this stage by the user. Finally, the SPL will try to find the U-boot image, load it into the RAM and pass the control of CPU to U-boot.

The third phase starts with the execution of the U-boot. This program is the operating system bootloader and provides the ability to configure the booting options using a command prompt. After setting the configuration variables, which defines the boot option of the kernel, the U-boot will load and pass the execution flow to

the kernel.

The final stage is the execution of the kernel image. The kernel image can be a plain binary file or zipped. An example of kernel image is uImage, where prior to the kernel there is also an information header to give details about the kernel. The information usually describe for which architecture the kernel has been compiled, the size of the image and the address where the image will be loaded.

In table 7.5 we present the performance results of boot time of the BeagleBone Black. For each different configuration, we run five test to measure the average time of the boot process.

The details of the configuration are as following:

- kernel image: uImage
- kernel version: 3.8.13
- U-boot version: 2017.05

Table 7.5: Boot time performance of BeagleBone Black

test#	test1	test2	test3	test4	test5	Average
U-boot	0.698	0.698	0.698	0.698	0.699	0.698
kernel+INIT	1.985	2.049	2.146	2.012	2.006	2.040
Total	2.683	2.747	2.844	2.710	2.705	2.738

In our experiments, we apply Verified U-Boot, and since uImage does not support any hashing process, we need to use another type of image. Thus, we use the FIT type as kernel image; furthermore, to fit the memory address space of the Beable-Bone we need to compress it. We use the lzop algorithm to compress the kernel before formatting it into a FIT image. In table 7.6 is presented the performance after using FIT image, and from now on it will be considered as the base case for following time measurements.

Table 7.6: Boot time after using FIT Image - Base Case

Test#	Test1	Test2	Test3	Test4	Test5	Average
U-Boot	0.985	0.956	0.957	0.957	0.957	0.962
kernel+INIT	2.971	2.971	3.070	3.023	3.020	3.011
Total	3.956	3.927	4.027	3.980	3.977	3.973

7.1.3 CryptoCape for BeagleBone Black

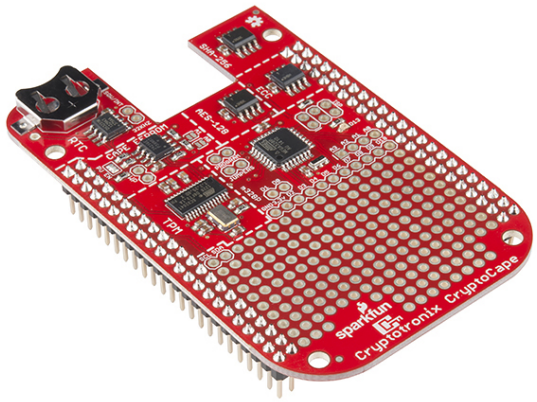

Figure 7.5: CryptoCape [3]

The CryptoCape is a plug-in board to extend the capabilities of the BeagleBone and is developed in collaboration by Cryptotronix [3] and Sparkfun [82]. More specifically it provides cryptographic functions using dedicated hardware. In table 7.7 we list the cryptographic-hardware of CryptoCape.

Table 7.7: CyptoCape Properties

Hardware	Crypto-Function	Manufacturer
ATAES132	AES-128	Atmel®
AT97SC3204T	TPM	Atmel®
ATSHA204	SHA-256	Atmel®
ATECC108	ECC	Atmel®

For the scope of the thesis, we use only the AT97SC3204T TPM from CryptoCape. We have explained the details of the functionality of the TPM in Chapter 5.

7.1.4 Micro-SD card used by both of the embedded devices

We store the boot partition (parts of it which are not located in ROM memories on the embedded devices) on a Micro-SD card. Since the type of memory can have an impact on the boot time as mentioned in Chapter 6.1.2.I, it is relevant to mention the hardware specification for the Micro-SD card used (see Table 7.8).

Table 7.8: Micro-SD card specifications

Model (size)	**Read speed (MB/s)**	**Write speed (MB/s)**	**Manufacturer**
Pro microSDHC UHS-I class 1 (16GB)	>90	>80	Samsung

7.2 Evaluation of verification techniques

In this sub-chapter, we present the results of our proof of concepts. We have tested two of the most available (in terms of resources) and suitable (in terms of popularity) techniques that were mentioned in Chapter 5. We selected one software- and one hardware technique to see how much impact security can have in terms of time overhead. The software technique is *Verified U-Boot* and the hardware technique is *Verified boot with TPM*.

The choice of Verified U-Boot is based on the popularity of this bootloader (U-boot). Although the popularity brings advantages, it is only up to a certain extent, such as guidelines and necessary low level hardware specific code/files to enable the Verified boot feature. The additional content that we had to add/program was fortunately also achieved within the limited time period of this thesis.

The choice of Verified Boot with TPM is due to the reason of testing one of the most important standards available in the Trusted Computing concept and also since we managed to find a suitable hardware (see 7.1.3) suiting one of the experimental setups we are using. The details of how to configure the Verified U-boot technique on both of the experimental setups are provided in Appendix B.

7.2.1 Performance Evaluation of Verified U-Boot (Software based technique)

Here we present the performance results of Verified U-Boot applying different cryptographic algorithms and using the experimental setup which we have explained in previous sub-chapter. These results are showing how different degrees of security (the cryptographic algorithms) can have an impact on the boot-up time. For comparison purposes, we present the results in column graphs (Figures 7.6 - 7.12).

7.2.1.1 Impact on performance after hash integrity check

The size of the kernel is quite big for singing and verifying its value. For this reason, a common method is to use a cryptographic hash function [83]. The main purpose of a cryptographic hash function is to map an arbitrarily long data to a small and fixed number of the bit string. Cryptographic functions need to fulfill certain cryptographic properties to be considered secure, for more details see [83].

U-Boot supports three cryptographic hash functions, namely MD5 [84], SHA-1 & SHA-256 [85]. Table 7.9 present the properties of the functions regarding the security perspective.

Table 7.9: Hashing algorithms properties

Algorithm	Message Digest Size (bits)	Block Size (bits)	Rounds
MD5	128	512	64
SHA-1	160	512	80
SHA-2	256	512	64

The important factor of a cryptographic hash function is the message digest size; the message digest or simply digest is the output of the hash function. The bigger the digest size is, the stronger is the resistance against collision attacks. Regarding MD5 there have been several collisions reported [86] over the years and should be avoided for security reasons; nevertheless, we include in our investigation to compare trade-offs in performance. With reference to SHA-1, quite recently in February 2017, Google announced the first collision with SHA-1 [87]. As for SHA-256, there have not been any collisions reported yet. For more details about hash collision see [83]. For our purposes, the size of the digest is a crucial factor for the performance of the hash function, which we are trying to investigate. Therefore, we start our examination by using different hashing algorithms for validating the integrity of the kernel image, and we measure the performance impact on the boot time.

In our configurations, we are testing the experiment five times and calculate the average time. The tables below present the results for both of the boards, the results are taken regarding the performance impact as for the hashing algorithms MD5, SHA-1, and SHA-2.

Table 7.10: Performance impact of MD5 - BeagleBone

Test#	Test1	Test2	Test3	Test4	Test5	Average
U-Boot	1.014	1.016	1.016	1.016	1.016	1.016
verify overhead	0.124	0.123	0.123	0.123	0.123	0.123

Table 7.11: Performance impact of SHA-1 - BeagleBone

Test#	Test1	Test2	Test3	Test4	Test5	Average
U-Boot	1.051	1.052	1.055	1.052	1.051	1.052
verify overhead	0.157	0.158	0.158	0.157	0.158	0.158

Table 7.12: Performance impact of SHA-256 - BeagleBone

Test#	Test1	Test2	Test3	Test4	Test5	Average
U-Boot	1.113	1.114	1.108	1.109	1.110	1.111
verify overhead	0.213	0.213	0.213	0.213	0.213	0.213

Table 7.13: Performance impact of MD5 - Raspberry Pi

Test#	Test1	Test2	Test3	Test4	Test5	Average
U-Boot	1,020	1,020	1,021	1,021	1,022	1,021
verify overhead	0,124	0,123	0,123	0,124	0,124	0,124

Table 7.14: Performance impact of SHA-1 - Raspberry Pi

Test#	Test1	Test2	Test3	Test4	Test5	Average
U-Boot	1,05	1,051	1,05	1,05	1,051	1,050
verify overhead	0,152	0,152	0,152	0,152	0,152	0,152

Table 7.15: Performance impact of SHA-256 - Raspberry Pi

Test#	Test1	Test2	Test3	Test4	Test5	Average
U-Boot	1,114	1,114	1,114	1,114	1,115	1,114
verify overhead	0,213	0,213	0,213	0,213	0,215	0,213

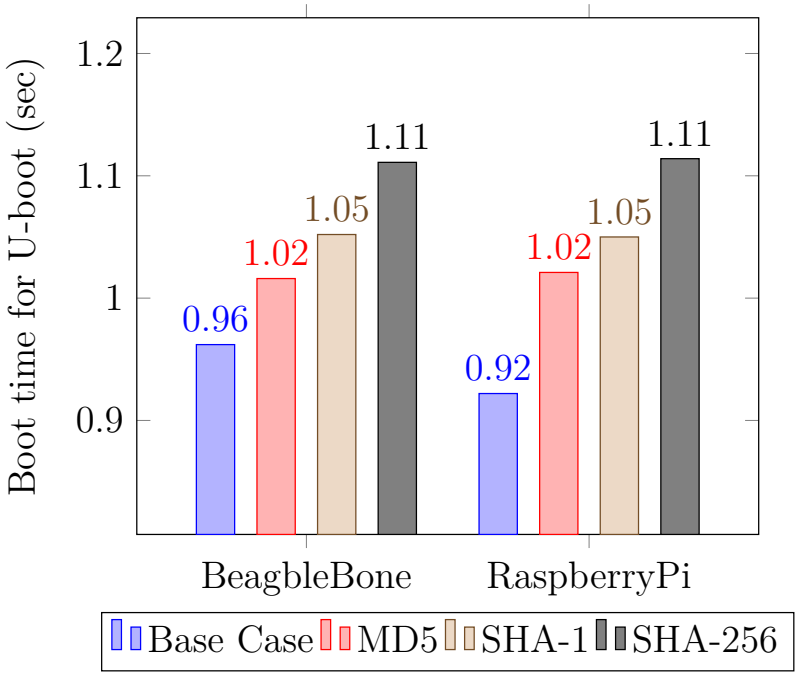

Figure 7.6: Performance impact after hash integrity check

The Figure 7.6 is showing the increase in boot time as we increase the security against collisions attacks. By introducing the hash integrity, we significantly increased the boot time. MD5 performs better than the other algorithms. SHA-1's performance is in the middle between MD5 and SHA-256. The hash algorithm SHA-256 has the worst performance but also provides better integrity assurance among the inspected algorithms. Finally, it is interesting to note that the performances are similar even though we are using different platforms to set up our experiments.

7.2.1.2 Impact on performance after verification

The integrity of the kernel is not the only one cryptographic property that we need to assure, we need to verify the authenticity of the kernel as well. To assure the authenticity of the kernel, we need to apply Public Key Cryptography (PKC), for the details about PKC see [83]. In general, the hash of the kernel image is being signed by a private key, and the verifying bootloader, U-Boot in our case, can verify the authenticity of the hash value by applying the public key. In Chapter 5, we have explained the details of the verification process in U-Boot.

The most popular [83] public key cryptographic algorithms are RSA [83] and Elliptic Curves (ECC) [83]. Unfortunately, U-Boot currently supports only RSA, even though ECC is considered to be faster [83]. The most important factor of public key algorithms is the size of the key, with bigger key sizes it is more difficult for an attacker to brake the crypto-system. On the over hand, the key size determines the time performance of the crypto-algorithm.

Nevertheless, for our purpose the speed of calculating the signature is not something that should be taken into consideration, since this is being made in the setup of the boot system (at a host device) and not in the execution time. The way to apply the verification and the files to be included play more important role regarding the trade-off between security and time performance. In this way, we demonstrate two different ways to setup the verification process. The different configurations provide different security assurances regarding the boot system.

I. Verification performance impact after using MD5

We start the examination by applying the two different key sizes of the RSA, namely 2048 bits and 4096 bits. We verify the signatures of the hash values of the kernel & device tree (fdt). We start the examination with the MD5 hashing algorithm that has shown the best performance. We consequently present the performance impact of different configurations and key sizes.

Since U-Boot does not support signatures using MD5, we re-hash the output of MD5 with SHA-1, which has a minor impact since the digest of MD5 is quite small in size (128 bits).

Table 7.16: Verify kernel & fdt with RSA 2048 using MD5 - BeagleBone

Test#	Test1	Test2	Test3	Test4	Test5	Average
U-Boot	1.234	1.259	1.258	1.239	1.233	1.245
verify overhead	0.284	0.283	0.284	0.284	0.283	0.284

Table 7.17: Verify kernel & fdt with RSA 4096 using MD5 - BeagleBone

Test#	Test1	Test2	Test3	Test4	Test5	Average
U-Boot	1.342	1.343	1.344	1.343	1.344	1.343
verify overhead	0.339	0.337	0.338	0.338	0.337	0.338

Table 7.18: Verify kernel & fdt with RSA 2048 using MD5 - Raspberry Pi

Test#	Test1	Test2	Test3	Test4	Test5	Average
U-Boot	1,256	1,254	1,256	1,256	1,257	1,256
verify overhead	0,297	0,296	0,296	0,296	0,297	0,296

Table 7.19: Verify kernel & fdt with RSA 4096 using MD5 - Raspberry Pi

Test#	Test1	Test2	Test3	Test4	Test5	Average
U-Boot	1,389	1,390	1,392	1,389	1,391	1,390
verify overhead	0,362	0,362	0,362	0,361	0,361	0,362

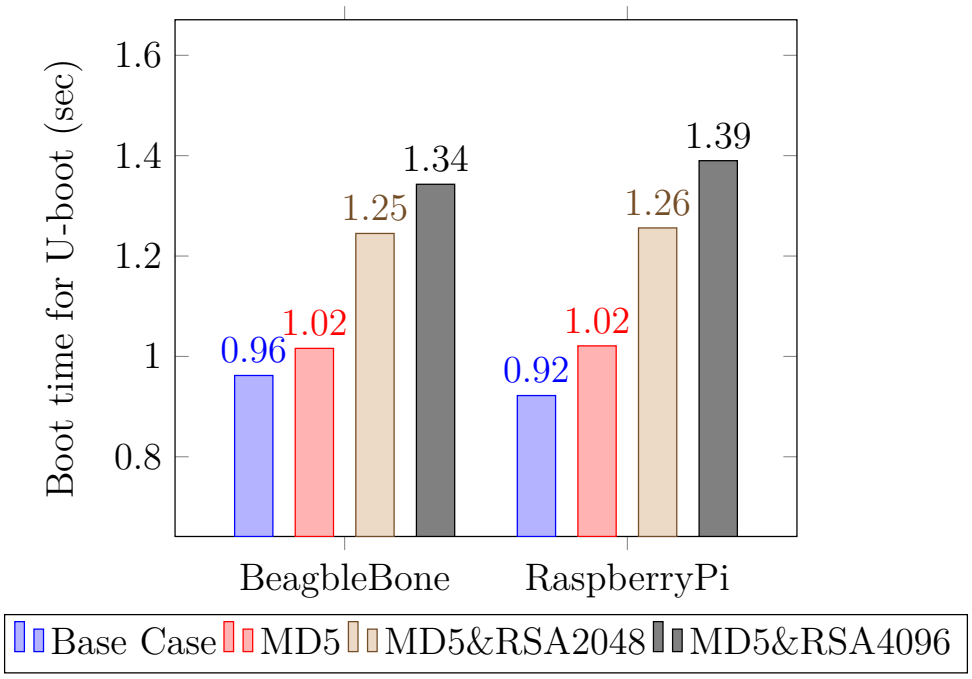

Figure 7.7: Performance impact after verifying kernel & fdt (MD5)

The Figure 7.7 demonstrates the scale of increase in the boot time when applying verification using RSA. If we look at the details, if we verify the kernel and fdt separately, it highly increases the boot time. Here, we can see this by using bigger key size (4096 bits) the security trade-off has a high impact on the performance of the boot system.

Thus, we examine the performance of the verification using a different method. In this method, we hash and verify the entire configuration which includes both the kernel and the device tree. Because in this method we calculate a single hash value and expect to improve the performance of the boot system.

Table 7.20: Verify the configuration with RSA 2048 using MD5 - BeagleBone

Test#	Test1	Test2	Test3	Test4	Test5	Average
U-Boot	1.018	1.019	1.021	1.020	1.020	1.020
verify overhead	0.123	0.123	0.123	0.123	0.123	0.123

Table 7.21: Verify the configuration with RSA 4096 using MD5 - BeagleBone

Test#	Test1	Test2	Test3	Test4	Test5	Average
U-Boot	1.045	1.046	1.046	1.046	1.030	1.043
verify overhead	0.122	0.122	0.122	0.123	0.123	0.122

Table 7.22: Verify the configuration with RSA 2048 using MD5 - Raspberry Pi

Test#	Test1	Test2	Test3	Test4	Test5	Average
U-Boot	1,030	1,029	1,029	1,029	1,028	1,029
verify overhead	0,124	0,122	0,123	0,122	0,123	0,123

Table 7.23: Verify the configuration with RSA 4096 using MD5 - Raspberry Pi

Test#	Test1	Test2	Test3	Test4	Test5	Average
U-Boot	1,053	1,053	1,054	1,053	1,053	1,053
verify overhead	0,123	0,123	0,122	0,122	0,123	0,123

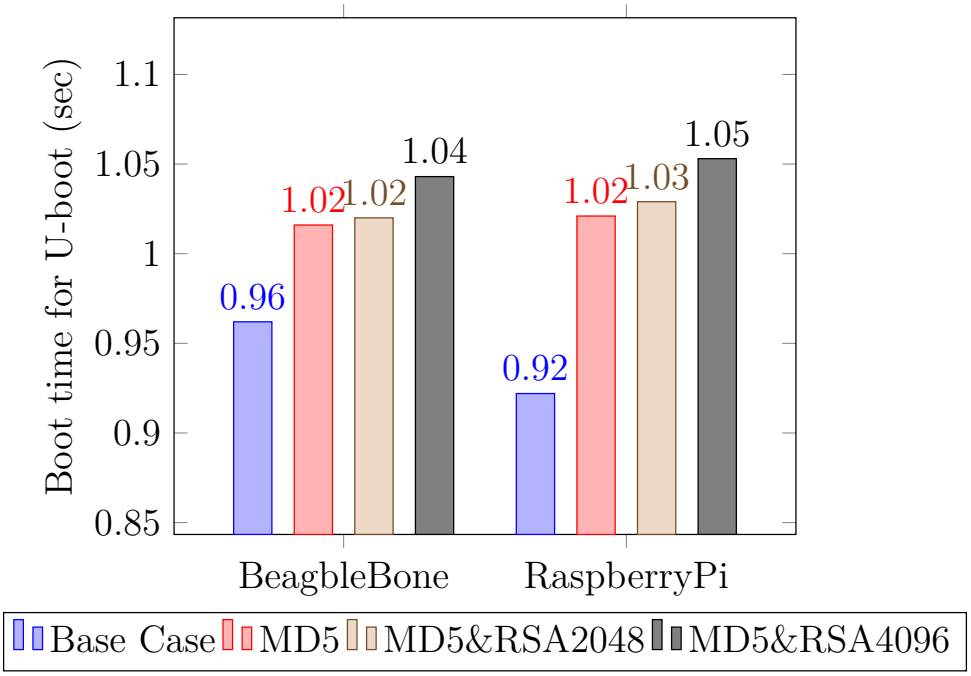

Figure 7.8: Performance impact after verifying the configuration (MD5)

In Figure 7.8, we can see that we have improved the performance by verifying the configuration rather the kernel and fdt separately. Also, the verification using a key size of 2048-bits perform slightly worse compared to bare hashing with MD5. We need to notice here that the option of hashing the configuration is available only when the verification is enabled; therefore, is not possible to test the performance of only the hash integrity of the configuration. Lastly, the choice of using greater key size which provides greater security assurance does not have a high impact on the performance of the boot system.

II. Verification performance impact after using SHA-1

We continue the examination by presenting the results of the hashing algorithm SHA-1. Yet again, we do apply the two supported key sizes of RSA (2048 bits and 4096 bits). We start by demonstrating the results when we verify the signatures of the kernel & device tree hashes separately.

Table 7.24: Verify kernel & fdt with RSA 2048 using SHA1 - BeagleBone

Test#	Test1	Test2	Test3	Test4	Test5	Average
U-Boot	1.286	1.280	1.281	1.280	1.281	1.282
verify overhead	0.318	0.319	0.317	0.317	0.319	0.318

Table 7.25: Verify kernel & fdt with RSA 4096 using SHA1 - BeagleBone

Test#	Test1	Test2	Test3	Test4	Test5	Average
U-Boot	1.413	1.391	1.377	1.380	1.379	1.388
verify overhead	0.373	0.373	0.372	0.374	0.373	0.373

Table 7.26: Verify kernel & fdt with RSA 2048 using SHA1 - Raspberry Pi

Test#	Test1	Test2	Test3	Test4	Test5	Average
U-Boot	1.312	1.310	1.311	1.313	1.311	1.311
verify overhead	0.328	0.325	0.327	0.326	0.327	0.326

Table 7.27: Verify kernel & fdt with RSA 4096 using SHA1 - Raspberry Pi

Test#	Test1	Test2	Test3	Test4	Test5	Average
U-Boot	1,420	1,418	1,419	1,420	1,421	1,420
verify overhead	0,391	0,389	0,389	0,389	0,390	0,390

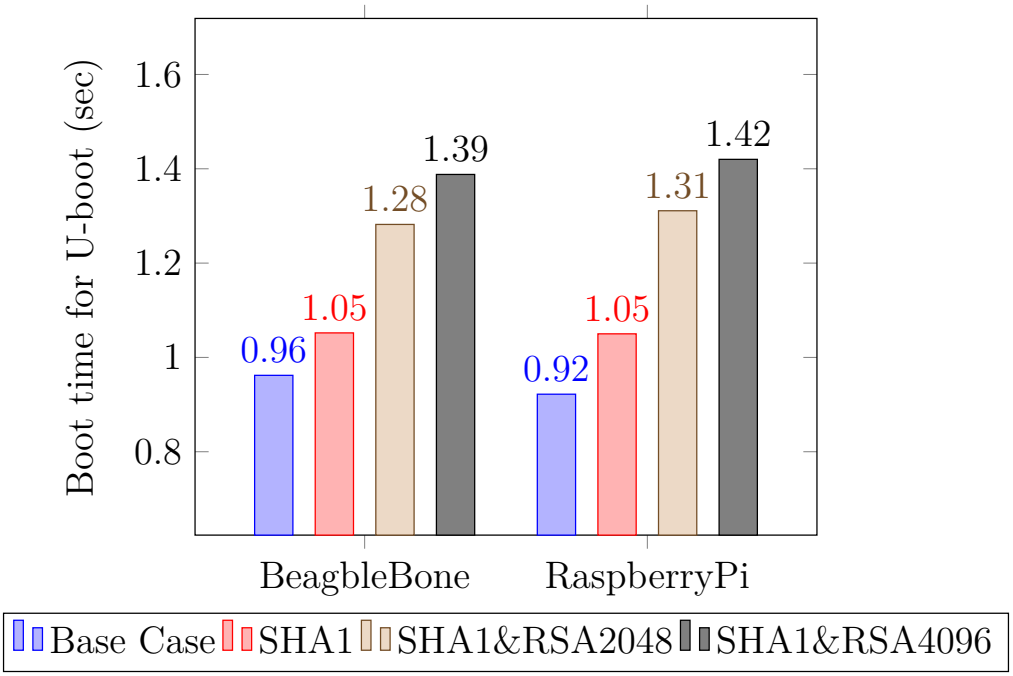

Figure 7.9: Performance impact after verifying the kernel & fdt (SHA1)

In Figure 7.9, we can see an increase in the boot time as expected. As we can see, when verifying the kernel and fdt separately, we increase the boot time. Also, the use of a larger key size (4096 bits) has a significantly increased the trade-off between security and performance.

Next, we examine the performance of the verification again, using the improved method. As being explained previously, in this method, we hash and verify the entire configuration, which includes both the kernel and the device tree.

Table 7.28: Verify the configuration with RSA 2048 using SHA1 - BeagleBone

Test#	Test1	Test2	Test3	Test4	Test5	Average
U-Boot	1.093	1.075	1.076	1.075	1.076	1.079
verify overhead	0.157	0.158	0.156	0.158	0.157	0.157

Table 7.29: Verify the configuration with RSA 4096 using SHA1 - BeagleBone

Test#	Test1	Test2	Test3	Test4	Test5	Average
U-Boot	1.089	1.084	1.075	1.075	1.074	1.079
verify overhead	0.157	0.158	0.158	0.157	0.157	0.157

Table 7.30: Verify the configuration with RSA 2048 using SHA1 - Raspberry Pi

Test#	Test1	Test2	Test3	Test4	Test5	Average
U-Boot	1.057	1.058	1.059	1.060	1.058	1.058
verify overhead	0.152	0.152	0.152	0.154	0.152	0.153

Table 7.31: Verify the configuration with RSA 4096 using SHA1 - Raspberry Pi

Test#	Test1	Test2	Test3	Test4	Test5	Average
U-Boot	1,084	1,082	1,081	1,084	1,085	1,083
verify overhead	0,152	0,152	0,152	0,152	0,154	0,152

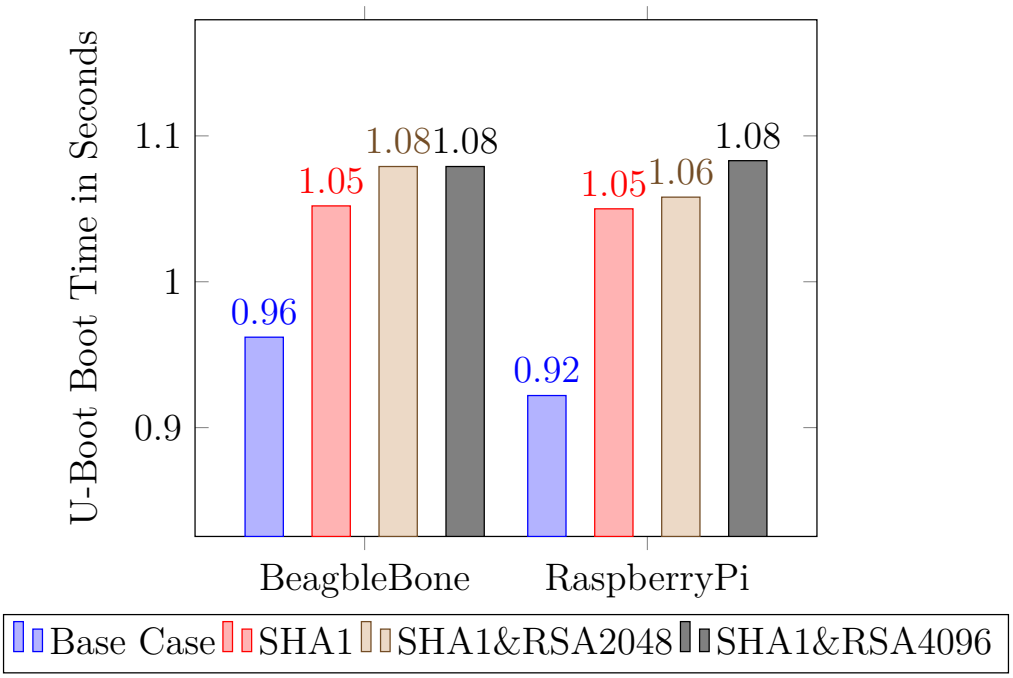

Figure 7.10: Performance impact after verifying the configuration (SHA1)

In Figure 7.10, we can see that we improved the performance as expected. Moreover, the verification of the configuration does not have any further impact on the boot time performance. Finally, using a key size of 2048 or 4096 does not make any great difference in the performance; therefore, the security trade-off is neutral here.

III. Verification performance impact after using SHA-256

Finally, we examine the performance impact using the strongest supported hashing algorithm SHA-256 (in U-Boot). Again we applying the two supported key sizes of the RSA (2048 bits and 4096 bits). We start by demonstrating the results when we verify the signatures of the kernel & device tree hashes.

Table 7.32: Verify kernel & fdt with RSA 2048 using SHA256 - BeagleBone

Test#	Test1	Test2	Test3	Test4	Test5	Average
U-Boot	1.411	1.393	1.394	1.395	1.394	1.397
verify overhead	0.427	0.427	0.427	0.428	0.427	0.427

Table 7.33: Verify kernel & fdt with RSA 4096 using SHA256 - BeagleBone

Test#	Test1	Test2	Test3	Test4	Test5	Average
U-Boot	1.526	1.507	1.501	1.503	1.503	1.508
verify overhead	0.481	0.481	0.481	0.482	0.481	0.481

Table 7.34: Verify kernel & fdt with RSA 2048 using SHA256 - Raspberry Pi

Test#	Test1	Test2	Test3	Test4	Test5	Average
U-Boot	1.448	1.447	1.448	1.448	1.449	1.448
verify overhead	0.444	0.444	0.446	0.444	0.445	0.444

Table 7.35: Verify kernel & fdt with RSA 4096 using SHA256 - Raspberry Pi

Test#	Test1	Test2	Test3	Test4	Test5	Average
U-Boot	1.620	1.619	1.619	1.622	1.623	1.620
verify overhead	0.510	0.508	0.509	0.509	0.510	0.509

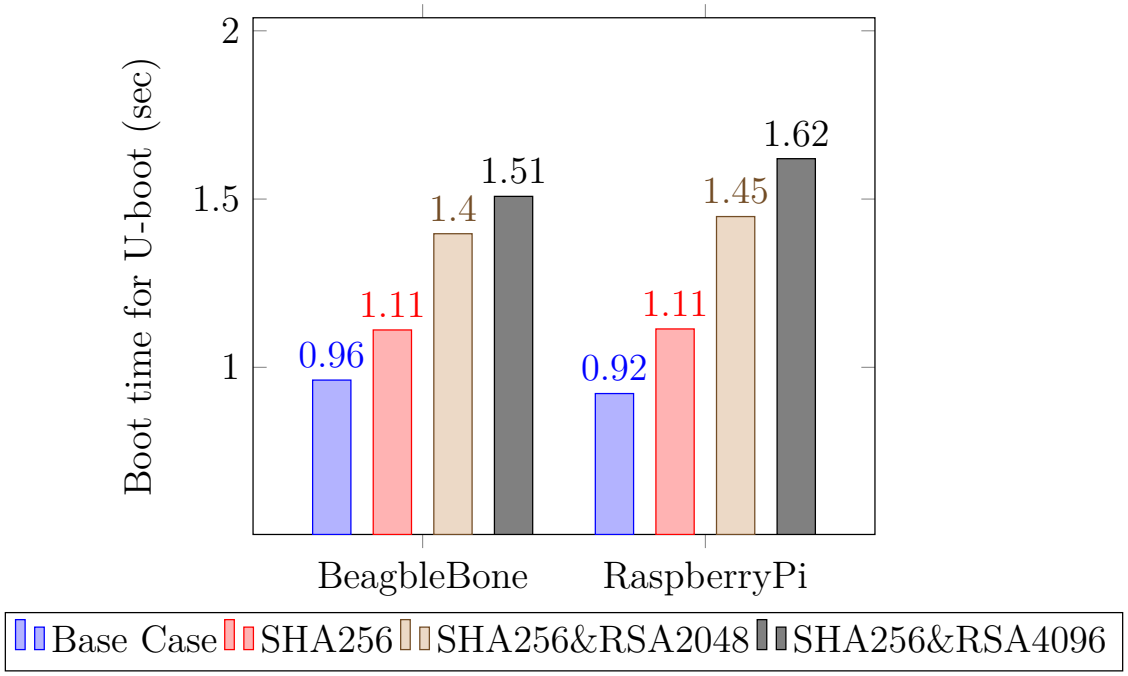

Figure 7.11: Performance impact after verifying kernel & fdt (SHA256)

In Figure 7.11, we can see the increase in the boot time as expected. As we can see, if we verify the kernel and fdt separately, it is increasing the boot time. Also, the use of a larger key size (4096 bits) has significantly increased the trade-off between security and performance.

Next, we examine the performance of the verification again, using the improved method. As being explained previously, in this method, we hash and verify the entire configuration, which includes both the kernel and the device tree.

Table 7.36: Verify the configuration with RSA 2048 using SHA256 - BeagleBone

Test#	Test1	Test2	Test3	Test4	Test5	Average
U-Boot	1.125	1.126	1.126	1.127	1.127	1.126
verify overhead	0.213	0.212	0.213	0.213	0.213	0.213

Table 7.37: Verify the configuration with RSA 4096 using SHA256 - BeagleBone

Test#	Test1	Test2	Test3	Test4	Test5	Average
U-Boot	1.158	1.152	1.153	1.134	1.135	1.146
verify overhead	0.232	0.232	0.233	0.233	0.234	0.233

Table 7.38: Verify the configuration with RSA 2048 using SHA256 - Raspberry Pi

Test#	Test1	Test2	Test3	Test4	Test5	Average
U-Boot	1,152	1,151	1,151	1,151	1,151	1,151
verify overhead	0,213	0,213	0,213	0,213	0,212	0,213

Table 7.39: Verify the configuration with RSA 4096 using SHA256 - Raspberry Pi

Test#	Test1	Test2	Test3	Test4	Test5	Average
U-Boot	1,167	1,167	1,170	1,166	1,166	1,167
verify overhead	0,212	0,213	0,214	0,213	0,213	0,213

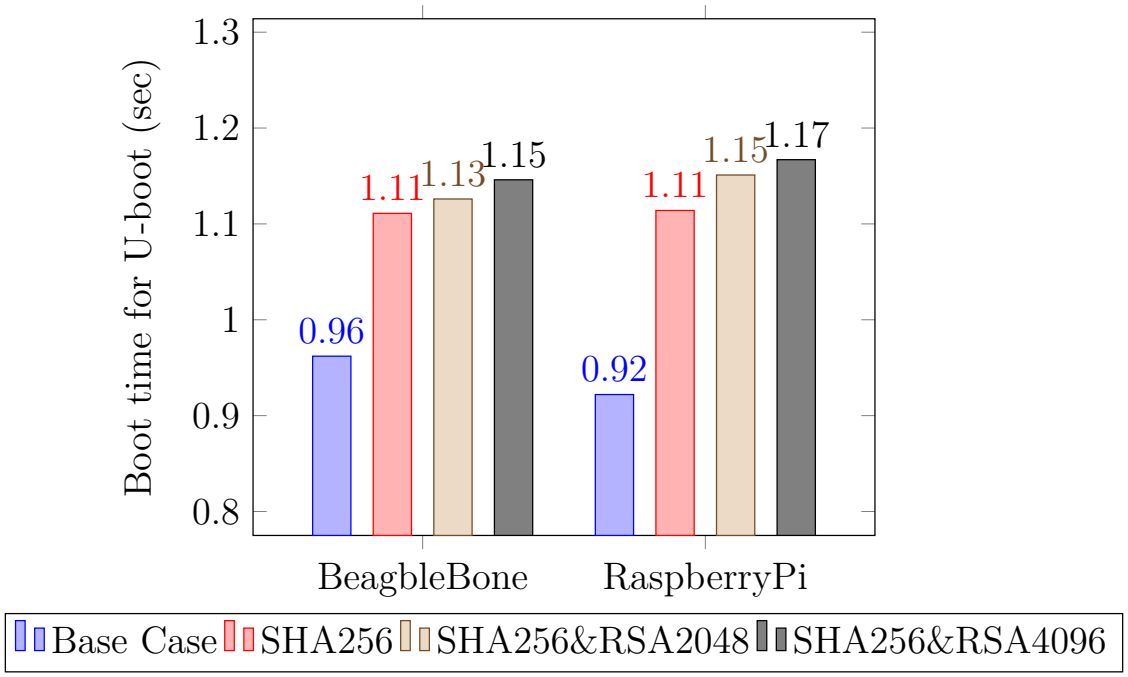

Figure 7.12: Performance impact after verifying the configuration (SHA256)

In Figure 7.12, we can see again that we improve the performance as expected. Moreover, the verification of the configuration does not have any significant impact on the boot time performance. Using the larger key size (4096) has a small impact on the boot time, but it can be considered negligible.

7.2.2 Performance evaluation of Verified Boot with TPM (Hardware-based technique)

We have described the details about Verified Boot with TPM in Chapter 5. Here we present an implementation of Verified Boot with TPM only in BeagleBone Black. The reason to why we only implemented this technique in BeagleBone Black and not in the Raspberry Pi 3 is due to the hardware availability/suitability and software availability. After a long period of investigation, we were not able to find an quickly available TPM hardware among the few that existed, and less did we find any trace of a guide, reference or necessary low-level hardware specific code/files to the application of a TPM on a Raspberry Pi 3.

The CryptoCape provides the TPM as we present in subsection 7.1.3, which is compatible only for BeagleBone. Nevertheless, we need to initialize before we can use it.

In addition, We need to take ownership and produce the endorsement keys which we have explained in Chapter 5. Also, two NVRAM memory indexes should be configured to save two sealed data (also see Chapter 5), one for the U-Boot pre-environment and one the kernel image.

Since the BealgeBone uses U-Boot by default, the first step was to obtain a version of U-Boot that supports communication with the TPM of the CryptoCape. A version of U-Boot that supports Inter-integrated Circuit (I2C) communication for AT97SC3204T (Atmel TPM) we can found it at github.com/theopolis/u-boot-sboot. Next, we need to patch the U-Boot with an implementation of Verified Boot with TPM, which we can found at github.com/theopolis/sboot.

This technique is just one possible implementation of verified/secure boot. It is being used in our project because it is the only currently available implementation for the BeagleBone Black. However, this implementation is not following thoroughly the concept that we have presented in Chapter 5. The Root of Trust does not start from the boot ROM, but there is a compromise. The measurements and thereby the Root of Trust is starting from the U-Boot pre-environment which includes:

- The U-Boot image
- Platform specifics (label and configuration)
- Environment variables and commands that U-Boot has pre-set
- Device tree files of U-Boot & kernel
- Kernel image

The method starts by performing the basic tests of the TPM (TPM_Start, TPM_SelfTest) this is necessary to ensure that the TPM is enabled and activated. Next, it measures the U-Boot pre-environment as explained above and extends for each time the PCR value (see Chapter 5). Next, it compares the PCR measurements with the sealed data in the NVRAM of TPM. In the case of success, the boot-process will continue normally, in the case of failure the boot process will halt.

A preferable implementation would have started by making the measurements immediately after the boot ROM. The main problem is that the boot process of the BeagleBone is not configurable and the first boot stage can not be modified. Nevertheless, the method regarding the initialization of the TPM, measurements of the U-Boot and the Kernel are still the same. Therefore, the performance impact does not change even though the security assurance is different.

In table 7.40, we present the boot time performance when we apply Verified Boot with TPM, and we named it for simplicity VBoot_TPM. In table 7.41, we present in details of the overhead caused by the TPM.

Table 7.40: Time performance using VBoot_TPM

test#	test1	test2	test3	test4	test5	Average
Boot Time	3.201	3.199	3.199	3.199	3.199	3.199
Verify overhead	1.925	1.922	1.923	1.922	1.922	1.923

Table 7.41: Verification overhead of using a TPM

Overhead	Average time
TPM Init	0.993
Measurements	0.138
Extend PCR values	0.791
Total	1.923

7.3 Evaluation of optimization methods in Verified U-boot

In this sub-chapter, we present the techniques used to optimize U-boot and results showing the impact of the optimization. We gather the boot time data for the basic case without a FIT image for Raspberry Pi and with a FIT image for BeagleBone. Since the U-boot configuration available for Raspberry Pi 3 is not supporting the use of a uEnv.txt file (see Chapter 6.1.1.III), we gather the results for these from the BeagleBone. However, the main point of interest is the overhead caused by the different non-optimized techniques which appear when the configurations are the default. In this case, it does not matter what type of an image that is read.

7.3.1 Optimization by disabling features in the environment variables

We detect some unnecessary features in the U-boot's default configuration of the Raspberry Pi 3. These features were not a problem in BeagleBone since U-boot by default is a boot-loader for this device and is mostly optimized. Among the unnecessary features, the one with the highest impact on the boot time was the USB support. Since the access to U-boot in Raspberry Pi 3 is maintained by a serial connection via the GPIO pins in our setup, the USB feature was totally unnecessary. The environment variable "*preboot=usb start*" was deleted and a significant optimization impact of approximately 2.6 seconds appeared (see Table 7.42).

Table 7.42: U-boot on Raspberry Pi 3 with and without USB support

test#	test1	test2	test3	test4	test5	Average
uboot.env (no USB)	0.881	0.883	0.882	0.882	0.882	0.882
uboot.env (with USB)	3.486	3.482	3.486	3.487	3.487	3.486
USB overhead	2.605	2.599	2.604	2.605	2.605	2.604

We notice that the USB overhead is included in the boot.scr technique presented in table 7.44. USB support could unfortunately not be disabled by using boot.scr since the boot.scr file is read after the USB initialization happens in U-boot. However, if we assume that some changes are made in the code and USB support is disabled in the case with boot.scr used, we can still see that uboot.env has a lower boot time overhead compared to boot.scr (see table 7.43).

Table 7.43: boot.scr without USB support compared to uboot.env

test#	test1	test2	test3	test4	test5	Average
uboot.env (no USB)	0.881	0.883	0.882	0.882	0.882	0.882
boot.scr (no USB)	1.430	1.436	1.431	1.430	1.430	1.431
USB overhead	2.605	2.599	2.604	2.605	2.605	2.604

7.3.2 Choice of automation techniques

The different techniques used to automatically boot the Linux kernel from U-boot have been tested as seen in table 7.44. The different techniques consist of using a boot.scr file, compilation of environment variables to the boot partition (uboot.env file for RPi) and to use an uEnv.txt file.

Before the optimization process, Raspberry Pi 3 was using boot.scr and BeagleBone Black used uEnv.txt.

Table 7.44: U-boot boot time with different automation techniques

test#	test1	test2	test3	test4	test5	Average
uboot.env (RPi3)	0.881	0.883	0.882	0.882	0.882	0.882
uboot.env (BBB)	0.754	0.754	0.754	0.755	0.755	0.754
uEnv.txt (BBB)	0.985	0.956	0.957	0.957	0.957	0.962
boot.scr (RPi)	4.035	4.035	4.035	4.035	4.035	4.035

By taking the overhead from uEnv.txt compared to uboot.env for BBB, we can merge the uboot.env data and use the boot time for RPi to ease the observation.

7.3.3 Comparison of U-boot automation techniques

To compare all three techniques to each other, we plot a graph which uses the uboot.env boot time as reference value (see Figure 7.13). Note that the uEnv.txt (General) data which we present in the graph is the overhead observed by comparing the BBB uboot.env to uEnv.txt added on top of the RPi uboot.env boot time;

uEnv.txt (BBB) - uboot.env (BBB) = uEnv.txt (Overhead) (1)
uboot.env (RPi) + uEnv.txt (Overhead) = uEnv.txt (General) (2)

Applying (1) and (2) to our values we maintain,

(1) 0.962 - 0.754 = 0.208 s
(2) 0.882 + 0.208 = 1.090 s

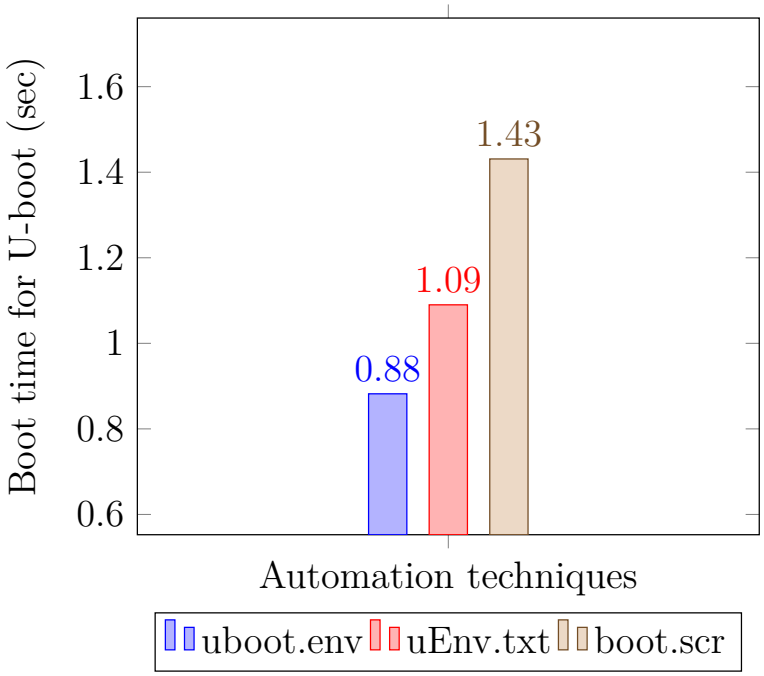

Figure 7.13: Comparison of different automation techniques

7.4 Discussion

We have examined several cryptographic algorithms and their implication in time performance. Due to the diversity of the applications and various limitations of embedded systems, there cannot be a straight line to follow regarding the security implications. In order to achieve a low time overhead, several compromises need to be made considering the security of the system.

With reference to the Verified U-Boot technique, the results indicate that there is no significant variation in the verification overhead using different hardware platforms. In our case, we used the two most popular general purposes embedded devices, BeagleBone Black and Raspberry Pi 3. We can, however, see a slight difference between the base case of the U-Boot boot-time without verified boot enabled (see 7.2 and 7.5). Even though we have been optimizing U-Boot in Raspberry Pi 3 to lower the boot time (by 3.153 seconds, see sub-section 7.3), it is a clear fact that U-Boot is not a default boot-loader for Raspberry Pi 3. This also makes it less probable for U-Boot to be ultimately optimized for Raspberry Pi 3 compared to a system which by default uses U-Boot (i.e. BeagleBone). So instead of further trying to investigate low-level code to optimize U-Boot for Raspberry Pi 3, we kept on going with our test in Verified U-Boot where the test results between the two hardware devices appeared to be insignificantly low as previously expressed.

On the other hand, the configuration of the Verified U-Boot can have a notable performance impact. More precise, the choice of the hashing algorithm is one of the crucial points. MD5 provides better time performance, but not recommended due to the quite high number of hash collisions [86]. We recommend using SHA-256 since it achieves a great balance between security and performance while having an insignificantly low (60 ms) time overhead compared to the other two hashing algorithms and the total boot time (1.11 s). Regarding the verification algorithm we evaluated only RSA; nevertheless, we have shown that the verification overhead has a minor impact on performance when smaller key size is used (see Tables 3.36-37 & 3.38-39). In this way, we recommend using RSA with the larger key size 4096-bits.

We have to note that a change in the image size which is being hashed, signed and verified will have an impact on the total boot time due to the different complexity in the hashing process. However, this change will not have an impact on the signing process since the size of the signed hash value will remain the same regardless of the image size. There has been no opportunity to evaluate the impact of different sized images during the Verified boot process, but this is a good test idea for future works.

As mentioned in the previous section, we can optimize the U-Boot by verifying both the device tree file and the kernel image together. This configuration is not only providing a time optimization but also increasing the security. The reason is, when signing the Flattened Device Tree (FDT) and kernel image together, what is signed is actually the configuration which inherently signs both of the file properties. However, when we sign the kernel image and FDT separately, the configuration is not signed. An attacker cannot change the files in this case, but what can happen is that a new FIT image with new configurations can be created. In this case, attacks such as mix and match attacks, where one valid kernel image is combined with another valid FDT is configured in the FIT image or roll back attacks, where an older version of signed files can be used in the configuration, can appear. Whether or not this will provide the attackers some benefits can be argued, it is nevertheless a security problem.

Regarding the Verified Boot using TPM, there are some very important points to notice. The use and support of cryptographic hardware are quite limited in embedded systems. Furthermore, manufacturers rarely address the Chain of Trust (see Chapter 5), which leads to ad-hoc solutions and compromises. In our implementation the Root of Trust is not starting at the Boot ROM; it is initiated at the pre-environment of the U-Boot. Compared to the software-based technique, this Root of Trust is in an earlier stage of the boot process, so we expect a larger verification time overhead. It is not recommended to set stages after the boot ROM as Root of Trust because it is possible that an attacker can modify prior boot stages without being detected. Therefore, we recommend always to start from the Boot ROM when implementing a verification method.

With reference to the boot performance of Verified boot with TPM, we have noticed a relative higher verification time overhead (approximately eight times more)

compared to the software-based technique (Verified U-boot). One of the causes of this performance is that there are quite more measurement requirements. More precisely, in this method, we verify the kernel, U-Boot and pre-U-Boot environment which includes all the variables and system configurations. In addition to the previously mentioned fact, this technique also uses a different strategy in the verification process of the measured stages as we explained in sub-chapter 7.2.2. We described an example of this strategy in sub-chapter 5.3.3.1. Therefore, the time overhead is reasonable to increase.

Another cause of the overhead difference is the initialization time of the TPM. Instead of initializing the TPM in series within the boot-up process, it can be powered on in parallel with the embedded platform it belongs to. This modification will improve the time performance of the method and in this case compared to the software-based solution the overhead of the TPM technique will be approximately four times more. Let us bear in mind that this relation is not constant so it can quickly become smaller if we add more verification stages to the software-based experiment. So it is difficult to say whether a software-based solution would be faster or slower compared to a hardware-based solution. In fact, these two tests should not be compared unless they verify the same number of stages. What is clear however is that a hardware based solution is more secure than the software based. Finally, our TPM supports only the hash function SHA-1; therefore, an improved TPM regarding security performance might provide better results.

We have to bear in mind that the TPM standard can be applied to different hardware. Hence a wide variety of TPM hardware providers exists. A better hardware module will thereby speed up the required verification process and lower the impact on the boot-up time. Since the quality and performance hardware relates to economic costs and space/heat overheads, we need to make a balance between these aspects.

An ideal implementation of Verified Boot with TPM, which starts the verification process at an earlier stage (design phase), can provide better security assurance for the boot process. Nevertheless, the verification time overhead will not be improved since even more series of measurements will take place. Therefore, there is an indication of the results that the improvement in security will confront with the time requirements of a real-time system such as ECU. So a relevant question appears, is it worth the effort, cost and time overhead to try to implement Verified boot in a system which is inherently not designed to implement it?

8

Related work

After implementing and evaluating two techniques we realize that security providing techniques which are added after-hand to systems with no security cannot secure the system to full extent. Furthermore, such ad-hoc solutions can even cause a lot of overhead.

Most of the research available today are tested on off-the-shelf embedded platforms without Boot ROM components supporting the verified boot process.

Our main contributions have been to evaluated two different proof of concepts so far. We can already see that the main focus on a Verified boot implementation shall be to consider the security during the design phase of the used system hardware. A conclusion and explanation regarding the design phase can be seen Chapter 9. The investigated papers within the area of a secure boot process tries to have a Root of Trust as early as possible in the boot process but normally uses inherently non-secure embedded platforms.

State of the art research which is related to this thesis with subjects in various threats can be seen in Section 8.1. We can in a similar way see research related to different Verified boot techniques below in Section 8.2, where the aforementioned fact regarding the design has not been considered in these papers.

8.1 Threat Model

In our Thesis, we identify the main threat before applying any security measure, which is kernel modifications by Rootkits. Various researchers worked to identify, analyze and categorize different types of Rootkits. Levine, Grizzard, and Owen [88] conduct an analysis on the behavior of Rootkits, by observing a full-phase attack of a Rookit using Honeynets. Yu, Yi, and Yong [15] worked towards the categorization of Kernel-level Rootkits, and they proposed a universal model for Rootkits.

8.2 Verified boot

Khalid et al. [70] shows that using a TPM on a HSM is not reliable with respect to certificate authorities (used by Secure boot for public key distribution) so they prefer a technique they define as Trusted boot which measures hash values (**Measure** -> **Extend** -> **Execute**). They do thereby concentrate on software based attacks

and not hardware based threats.

Even though that the authors design a system which starts at a very early stage in the boot process, they use two LEON3 highly configurable of-the-shelf processors both as the application processor and HSM. In this case, not having a specially designed HSM integrated into the embedded system violates our conclusion of a fully secure and low time overhead system.

A. S. Kushwaha [68] presents an excellent idea for PC systems to provide a secure boot technique which includes a USB key containing and verifying most of the boot process code. This technique can easily be adapted to embedded systems. The technique is, however, defining the RTM as a boot stage code which is "not easy to alter". With the present technology and the risks mentioned earlier in this report, where Bootkits are described, an RTM shall rather than being "not easy to alter" actually be impossible to alter without a physical access.

K. Dietrich and J. Winter [41] describes a software MTM technique. The explained technique is based on a software solution which starts at a later point in the boot process (after the init process). The authors propose a backward verification of the already executed stages of the boot process whenever the software MTM is active and running. This solution is however not very secure since a Bootkit which acts at an early stage can change the boot process and make it unreliable.

F. Devic et al. [42] are giving a good example of a remotely attested system where an update also can occur in a safe way over an unreliable network. The verification technique is however done in an FPGA system which is a programmable integrated circuit (IC) which can have an arguable security compared to a statically programmed/manufactured ASIC IC. Another fact is that the verification technique is a software based solution which is less secure compared to a hardware based solution according to our conclusion.

Similar arguments are valid for the FPGA technique by O. Khlaid et al. [70] The difference, in this case, is that an HSM is used. An argument for the security of an FPGA system compared to an ASIC system is not present in the paper provided by the authors.

9
Conclusion

In our thesis, we have shown that including the security perspective on top of a finished design of a safety-critical embedded system can be a challenging process. Moreover, a compromise between the level of security, system requirements and performance needs to be achieved.

By concluding, we can say that an adequately secure solution shall start from the Boot ROM until the final stage of the boot process (which is defined by the system designer). This will make the solution more optimal and thereby keep a low time overhead compared to ad-hoc solutions. In fact, we can clarify that a fully secure technique has to be considered during the hardware design phase whether it is a hardware or software technique. If a verified boot feature will be added to an arbitrary system with no support for verified boot features, the Root of Trust will consist of a larger part of the boot process since that part cannot be modified by the device owner but only by the manufacturer. In this case, as the Root of Trust concept name implies, we only trust this subset of the boot process which in theory might be relatively insecure.

The security of an embedded system also depends on the system properties and needs. A safety critical real-time system is an extreme case compared to the wide variety of existing analyzed systems. This is due to the requirement of high security at the same time as a low boot-up time is desired. Fortunately, the Trusted Computing Group provides a standard for mobile systems (Mobile Trusted Module) which can be merged into already existing designs. The research areas of security and real-time systems in embedded platforms are already decently large, since all the necessary components for implementing such systems exist, the only remaining work is to merge and test such systems to improve its performance further.

With certain assumptions such as the lack of the physical access to the system, a software-based solution could provide certain security properties without significantly deteriorating the performance of the system.

Finally, we conclude that a secure verified boot process has to be considered during the design phase of the embedded system. Otherwise, an adequate and optimal secured solution will be quiet unreasonable and difficult to achieve. In addition, the incomplete implementation will add a radical overhead to your boot time without promising much. Hence, adding it in after-hand will potentially increase the time overhead due to non-optimal hardware combination and more implications will ap-

pear when implementing the technique. As the potential of growth and popularity in safety critical real-time systems in the recent times easily can be seen, the application of these systems in vehicles are increasing and the world is becoming more and more connected with the Internet of Things which pose a security threat to our confidentiality or in some cases our lives.

9.1 Future work

The thesis has evaluated selected verification techniques but has bypassed the rest due to the lack of hardware support and time. Future work includes a wider evaluation including the remaining techniques by optimizing them and a more inclusive comparison among them with respect to security and time overhead.

Apart from evaluations, there can also be further experiments to retrieve results which might be useful in analysis. Examples include testing the verification of different sized kernel images and the evaluation on the time overhead as well. Another future work subject is to have a fully extended Chain of Trust implementation which ranges from the Boot ROM to at least the kernel image execution. Different memory types can be tested to evaluate whether or not they affect the boot-up time due to their architecture as discussed in chapter 6.1.2.I.

Furthermore, there exist more cryptographic algorithms rather than those that have used in Verified U-Boot. Therefore, it is expected by U-Boot to support more crypto-algorithms (e.g. Elliptic Curves Cryptography (ECC)). In this way, a more broad evaluation of the verified U-Boot would be into consideration.

We can also mention that the technique described in sub-chapter 5.3.3.2 (USB key) can be applied to an embedded system with slight modifications. Its suitability for vehicular systems and similarity to common embedded platforms makes it extra interesting to investigate in the future.

Bibliography

[1] R. P. foundation, "Raspberry pi 3 model b." https://www.raspberrypi.org/products/raspberry-pi-3-model-b/, 2016.

[2] T. Int, "Beaglebone black c." https://http://beagleboard.org/black, 2015.

[3] Cryptotronix, "Cryptocape." https://cryptotronix.com/products/cryptocape, 2017.

[4] R. P. foundation, "Downloads." https://www.raspberrypi.org/downloads/, 2016.

[5] R. P. foundation, "Raspberry pi hardware." https://www.raspberrypi.org/documentation/hardware/raspberrypi/README.md, 2016.

[6] W. Mauerer, *Professional Linux Kernel Architecture*. Birmingham, UK, UK: Wrox Press Ltd., 2008.

[7] M. Kerrisk, *The Linux Programming Interface: A Linux and UNIX System Programming Handbook*. San Francisco, CA, USA: No Starch Press, 1st ed., 2010.

[8] A. S. Berger, "Chapter 10 - the intel x86 architecture," in *Hardware and Computer Organization* (A. S. Berger, ed.), Embedded Technology, pp. 265 – 294, Burlington: Newnes, 2005.

[9] D. Seal, *ARM Architecture Reference Manual*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2nd ed., 2000.

[10] R. Love, *Linux Kernel Development*. Addison-Wesley Professional, 3rd ed., 2010.

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

[12] M. Major, *A Taxonomic Evaluation of Rootkit Deployment, Behavior and Detection*. ProQuest Dissertations Publishing, 2015.

[13] T. Yadav and A. M. Rao, *Technical Aspects of Cyber Kill Chain*, pp. 438–452. Cham: Springer International Publishing, 2015.

[14] Lockheedmartin, "Cyber kill chain." http://www.lockheedmartin.com/us/what-we-do/aerospace-defense/cyber/cyber-kill-chain.html, 2017.

[15] X. Li, Y. Zhang, and Y. Tang, "Kernel malware core implementation: A survey," in *2015 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery, CyberC 2015, Xi'an, China, September 17-19, 2015*, pp. 9–15, IEEE Computer Society, 2015.

[16] Wikipedia, "Advanced persistent threat." https://en.wikipedia.org/wiki/Advanced_persistent_threat, 2017.

[17] D. Mulligan and A. K. Perzanowski, "The magnificence of the disaster: Reconstructing the sony bmg rootkit incident," *Berkeley Technology Law Journal*, vol. 22, p. 1157, January 2007.

[18] Wikipedia, "Extended copy protection." `https://en.wikipedia.org/wiki/Extended_Copy_Protection`, 2017.

[19] Wikileaks, "Vault 7: Cia hacking tools revealed." `https://wikileaks.org/ciav7p1/`, 2017.

[20] E. Snowden, "Snowden doc search." `https://search.edwardsnowden.com/`, 2003.

[21] E. Cole, *Advanced Persistent Threat: Understanding the Danger and How to Protect Your Organization*. Syngress Publishing, 1st ed., 2013.

[22] M. Kerrisk, *The Linux Programming Interface: A Linux and UNIX System Programming Handbook*. San Francisco, CA, USA: No Starch Press, 1st ed., 2010.

[23] C. Cowan, F. Wagle, C. Pu, S. Beattie, and J. Walpole, "Buffer overflows: Attacks and defenses for the vulnerability of the decade," in *DARPA Information Survivability Conference and Exposition, 2000. DISCEX'00. Proceedings*, vol. 2, pp. 119–129, IEEE, 2000.

[24] J. P. Anderson and J. P. Anderson, "Computer security technology planning study," tech. rep., Air Force Electronic Systems Division, 1972.

[25] D. H. Aristizabal, D. M. Rodriguez, and R. Y. Guevara, "Measuring aslr implementations on modern operating systems," in *2013 47th International Carnahan Conference on Security Technology (ICCST)*, pp. 1–6, Oct 2013.

[26] Y. Park, Y. Lee, H. Kim, G.-J. Lee, and I.-H. Kim, *Hardware Stack Design: Towards an Effective Defence Against Frame Pointer Overwrite Attacks*, pp. 268–277. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006.

[27] K.-S. Lhee and S. J. Chapin, "Buffer overflow and format string overflow vulnerabilities," *Softw. Pract. Exper.*, vol. 33, pp. 423–460, Apr. 2003.

[28] M. Tran, M. Etheridge, T. Bletsch, X. Jiang, V. Freeh, and P. Ning, *On the Expressiveness of Return-into-libc Attacks*, pp. 121–141. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011.

[29] J. Li, Z. Wang, X. Jiang, M. Grace, and S. Bahram, "Defeating return-oriented rootkits with "return-less" kernels," in *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, (New York, NY, USA), pp. 195–208, ACM, 2010.

[30] Z. S. Huang and I. G. Harris, "Return-oriented vulnerabilities in arm executables," in *2012 IEEE Conference on Technologies for Homeland Security (HST)*, pp. 1–6, Nov 2012.

[31] F. M. David, E. M. Chan, J. C. Carlyle, and R. H. Campbell, "Cloaker: Hardware supported rootkit concealment," in *2008 IEEE Symposium on Security and Privacy (sp 2008)*, pp. 296–310, May 2008.

[32] A. Prakash, E. Venkataramani, H. Yin, and Z. Lin, "Manipulating semantic values in kernel data structures: Attack assessments and implications," in *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 1–12, June 2013.

[33] N. L. Petroni, Jr., T. Fraser, A. Walters, and W. A. Arbaugh, "An architecture for specification-based detection of semantic integrity violations in kernel dynamic data," in *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15*, USENIX-SS'06, (Berkeley, CA, USA), USENIX Association, 2006.

[34] N. L. Petroni, Jr. and M. Hicks, "Automated detection of persistent kernel control-flow attacks," in *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, (New York, NY, USA), pp. 103–115, ACM, 2007.

[35] H. Moon, H. Lee, I. Heo, K. Kim, Y. Paek, and B. B. Kang, "Detecting and preventing kernel rootkit attacks with bus snooping," *IEEE Transactions on Dependable and Secure Computing*, vol. 14, pp. 145–157, March 2017.

[36] B. Grill, A. Bacs, C. Platzer, and H. Bos, *"Nice Boots!" - A Large-Scale Analysis of Bootkits and New Ways to Stop Them*, pp. 25–45. Cham: Springer International Publishing, 2015.

[37] Wikipedia, "Stoned boot-kit." `https://en.wikipedia.org/wiki/Stoned_(computer_virus)`, 2017.

[38] D. E. Rodionov, A. Matrosov, and D. Harley, "Bootkits: Past, present and future," *Virus Bulletin*, 2014.

[39] V. Bashun, A. Sergeev, V. Minchenkov, and A. Yakovlev, "Too young to be secure: Analysis of uefi threats and vulnerabilities," in *14th Conference of Open Innovation Association FRUCT*, pp. 16–24, Nov 2013.

[40] S. Berger, K. Goldman, D. Pendarakis, D. Safford, E. Valdez, and M. Zohar, "Scalable attestation: A step toward secure and trusted clouds," *IEEE Cloud Computing*, vol. 2, pp. 10–18, Sept 2015.

[41] K. Dietrich and J. Winter, "Secure boot revisited," in *2008 The 9th International Conference for Young Computer Scientists*, pp. 2360–2365, Nov 2008.

[42] F. Devic, L. Torres, and B. Badrignans, "Securing boot of an embedded linux on fpga," in *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, pp. 189–195, May 2011.

[43] G. Fedorkow, "What's the difference between secure boot and measured boot?." `http://forums.juniper.net/t5/Security-Now/What-s-the-Difference-between-Secure-Boot-and-Measured-Boot/ba-p/281251`, 2015.

[44] J. D. Tygar and B. S. Yee, "Dyad: A system for using physically secure coprocessors," *Technical Report CMU-CS-91-140R*, 1991.

[45] SimonGlass, "Das u-boot – the universal boot loader." `http://www.denx.de/wiki/U-Boot/`, 2014.

[46] S. Glass, "Verified u-boot." `https://lwn.net/Articles/571031/`, 2013.

[47] T. C. O. team, "Verified boot." `http://www.chromium.org/chromium-os/chromiumos-design-docs/verified-boot`, 2009.

[48] T. A. team, "Verifying boot." `https://source.android.com/security/verifiedboot/verified-boot`, 2017.

[49] I. P. Project Management Institute, *A guide to the project management body of knowledge (PMBOK guide)*. Project Management Institute, Inc. (PMI), 2013.

[50] S. Papa, W. Casper, and S. Nair, "Placement of trust anchors in embedded computer systems," in *2011 IEEE International Symposium on Hardware-Oriented Security and Trust*, pp. 111–116, June 2011.

[51] Y. Li, Y. Zhang, P. Li, and P. Guo, "An efficient trusted chain model for real-time embedded systems," in *2015 11th International Conference on Computational Intelligence and Security (CIS)*, pp. 428–432, Dec 2015.

[52] H. C. A. van Tilborg and S. Jajodia, eds., *TCG Trusted Computing Group*, pp. 1279–1279. Boston, MA: Springer US, 2011.

[53] T. T. community, "Trusted computing." `https://trustedcomputinggroup.org/trusted-computing/`, 2017.

[54] T. T. community, "Trusted platform module (tpm) summary." `https://trustedcomputinggroup.org/trusted-platform-module-tpm-summary/`, 2008.

[55] T. T. community, "Trusted platform module (tpm)." `https://trustedcomputinggroup.org/work-groups/trusted-platform-module/`, 2017.

[56] "Information technology – Trusted Platform Module – Part 1: Overview," standard, ISO/IEC JTC 1 Information technology, May 2009.

[57] "TPM Main Part 1 Design Principles," standard, Trusted Computing Group, March 2011.

[58] T. Morris, *Trusted Platform Module*, pp. 1332–1335. Boston, MA: Springer US, 2011.

[59] Z. Yan, *Trust Modeling and Management in Digital Environments: From Social Concept to System Development: From Social Concept to System Development*. IGI Global research collection, Information Science Reference, 2010.

[60] L. Wilson, "Software stack." `https://trustedcomputinggroup.org/work-groups/software-stack/`, 2017.

[61] D. Challener, *TSS*, pp. 1336–1338. Boston, MA: Springer US, 2011.

[62] P. England, *Sealed Storage*, pp. 1087–1088. Boston, MA: Springer US, 2011.

[63] E. (k33a), "Trusted platform module (tpm)." `https://www.slideshare.net/k33a/trusted-platform-module-tpm`, 2014.

[64] H. C. A. van Tilborg and S. Jajodia, eds., *Remote Attestation*, pp. 1042–1042. Boston, MA: Springer US, 2011.

[65] I. Bente, B. Hellmann, T. Rossow, J. Vieweg, and J. von Helden, "On remote attestation for google chrome os," in *2012 15th International Conference on Network-Based Information Systems*, pp. 376–383, Sept 2012.

[66] SourceForge, "Trousers." `http://trousers.sourceforge.net/faq.html#2.3`, 2008.

[67] Wikipedia, "Trusted computing - remote attestation." `https://en.wikipedia.org/wiki/Trusted_Computing#Remote_attestation`, 2017.

[68] A. S. Kushwaha, "A trusted bootstrapping scheme using usb key based on uefi," *International Journal of Computer and Communication Engineering*, vol. 2, pp. 543–546, 09 2013. Copyright - Copyright IACSIT Press Sep 2013; Last updated - 2014-04-11.

[69] W. A. Arbaugh, D. J. Farber, and J. M. Smith, "A secure and reliable bootstrap architecture," in *Proceedings. 1997 IEEE Symposium on Security and Privacy (Cat. No.97CB36097)*, pp. 65–71, May 1997.

[70] O. Khalid, C. Rolfes, and A. Ibing, "On implementing trusted boot for embedded systems," in *2013 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, pp. 75–80, June 2013.

[71] W. Fang, C. Zhou, Y. Zhang, and L. Zhang, "Research and application of trusted computing platform based on portable tpm," in *2009 2nd IEEE International Conference on Computer Science and Information Technology*, pp. 506–509, Aug 2009.

[72] A. L. team, "Arm trustzone." `https://www.arm.com/products/security-on-arm/trustzone`, 2017.

[73] "Building a secure system using TrustZone technology," standard, ARM Security Technology, April 2009.

[74] S. Pinto, J. Pereira, T. Gomes, M. Ekpanyapong, and A. Tavares, "Towards a trustzone-assisted hypervisor for real time embedded systems," *IEEE Computer Architecture Letters*, vol. PP, no. 99, pp. 1–1, 2017.

[75] X. team, "Fpga vs. asic." `https://www.xilinx.com/fpga/asic.htm`, 2017.

[76] C. Hallinan, "Reducing boot time in embedded linux systems." `http://www.linuxjournal.com/magazine/reducing-boot-time-embedded-linux-systems`, 2009.

[77] M. Åsberg, T. Nolte, M. Joki, J. Hogbrink, and S. Siwani, "Fast linux bootup using non-intrusive methods for predictable industrial embedded systems," in *2013 IEEE 18th Conference on Emerging Technologies Factory Automation (ETFA)*, pp. 1–8, Sept 2013.

[78] C. W. Chang, C. Y. Yang, Y. H. Chang, and T. W. Kuo, "Booting time minimization for real-time embedded systems with non-volatile memory," *IEEE Transactions on Computers*, vol. 63, pp. 847–859, April 2014.

[79] E. Upton, "Ten millionth raspberry pi, and a new kit." `https://www.raspberrypi.org/blog/ten-millionth-raspberry-pi-new-kit/`, 2016.

[80] eLinux, "Rpi software." `http://elinux.org/RPi_Software`, 2014.

[81] T. Instruments, "Boot sequence." `http://processors.wiki.ti.com/index.php/Boot_Sequence`, 2017.

[82] Sparkfun, "Cryptocape." `https://learn.sparkfun.com/tutorials/cryptocape-hookup-guide`, 2017.

[83] W. Stallings, *Cryptography and Network Security: Principles and Practice*. Pearson Education, 3rd ed., 2002.

[84] R. L. Rivest, "The md5 message-digest algorithm." Internet RFC 1321, April 1992.

[85] Q. Dang, "Changes in federal information processing standard fips 180-4, secure hash standard," *Cryptologia*, vol. 37, pp. 69–73, Jan. 2013.

[86] E. Thompson, "Md5 collisions and the impact on computer forensics," *Digital Investigation*, vol. 2, pp. 36–40.

[87] Google, "Announcing the first sha1 collision." `https://security.googleblog.com/2017/02/announcing-first-sha1-collision.html`, 2017.

[88] J. G. Levine, J. B. Grizzard, and H. L. Owen, "Application of a methodology to characterize rootkits retrieved from honeynets," in *Proceedings from the fifth IEEE Systems, Man and Cybernetics Information Assurance Workshop*, pp. 15–21, June 2004.

[89] V. van der Veen, Y. Fratantonio, M. Lindorfer, D. Gruss, C. Maurice, G. Vigna, H. Bos, K. Razavi, and C. Giuffrida, "Drammer: Deterministic rowhammer attacks on mobile platforms," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, (New York, NY, USA), pp. 1675–1689, ACM, 2016.

[90] E. Database, "Google android 5.0 < 5.1.1 - stagefright .mp4 tx3g integer overflow (metasploit)." `https://www.exploit-db.com/exploits/40436`, 2016.

[91] C. T. R. Labs, "ordow v2.0." `https://blog.comodo.com/comodo-news/comodo-warns-android-users-of-tordow-v2-0-outbreak/`, 2016.

[92] Linaro, "How to run arm-trusted-firmware and u-boot with verification enabled." `https://wiki.linaro.org/WorkingGroups/Security/Verified-U-boot`, 2017.

A

Root Access

In our threat model we have made the assumption that the attacker gets root access privileges before deploying a Rootkit attack. Here we provide some techniques that have been used to gain root access under certain conditions.

A.1 Rowhammer Attack

Rowhammer is a behavior observed in physical memory where consecutive access attempts could cause the adjacent bits to flip [89]. New sophisticated attacks have exploited this phenomenon to bypass all modern security defenses. Consequently, an attacker can alter parts of physical memory by only using memory read operations.

However, the previous is possible under certain conditions. The physical memory has to use dynamic random-access memory (DRAM) technology. In addition, the continuous and consecutive read access requests have to be fast enough to cause a bit flip in adjacent cells.

In details, the main cause of the previous problem is based in the way DRAM chips are constructed. More details are provided here [89].

In the paper "Drammer: Deterministic Rowhammer Attacks on Mobile Platforms" [89], researchers have successfully managed to launch the Drammer attack into Android smartphones with certain specifications. The result of the exploitation is to escalate user privileges into root access.

A.2 Integer-overflow

Another method that an attacker can use to elevate the privileged access on the device is an integer-overflow exploit. The attack is divided into two stages. In the first stage, the attacker is using a software bug in which after an arithmetic operation the value of integer variable exceeds the maximum size that is defined for the integer variable type. With the integer-overflow, the attacker gets access to a system library address space.

The second stage is to deploy a payload and execute a chain of ROP gadgets which we have explained in Chapter 4. The outcome is a shell with root privileges.

A reported attack can be found in Android devices, and the reference number is CVE-2015-3864 [90].

A.3 Trojan virus

A more common method of getting privilege is through specially crafted malware such as a Trojan virus.

The Trojan is being installed into the device by deceiving the user to install software that includes useful functionality but also includes a malicious payload inside. During the installation, the software asks for privilege rights which the user need to accept in order to install the software. After that, the attacker can use the payload to use the privileged access.

A reported attack has been found in Android devices by Comodo Threat Research Labs with the name Tordow 2.0 [91]

B

U-boot Configurations

In this appendix, we describe an overview of the steps taken in order to configure the set-up in the different platforms which have been used in the thesis.

B.1 Verified U-Boot for Raspberry Pi 3 Model B

Since Raspberry Pi by default is not using U-Boot, it is initially necessary to download the source code of U-Boot which contains the specific configuration and dtb file for the correct Raspberry Pi model. The U-Boot bootloader do however not replace the second stage bootloader, in this case, we do have 3 stages of bootloaders where the final one is U-Boot.

The kernel can either be compiled or an already existing kernel can be downloaded. Since raspberry provides Raspbian which is based on Linux version 4.4, this kernel is executed by U-Boot. A dtb file is provided with the kernel "*bcm2710-rpi-3-b.dtb*" which is a comprehensive dtb file with more hardware devices included comparing to the dtb file which U-Boot will be using. So the necessary dtb file for U-Boot "*bcm2837-rpi-3-b.dtb*" is found in the U-Boot source folder in order to be used to insert the public key and be compiled into U-Boot.

The configuration file for Raspberry Pi 3 (*rpi_3_32b_defconfig*) has to be changed to enable verified boot and print the statistical data to maintain the measured boot time. Another file which needs to be written is an *.its* file (image tree source) which is the source file for the FIT image (*.itb*). This file contains all the configurations for the verification process; the algorithm to use for hashing, signing, the kernel image format and location, the dtb file location and finally which files to sign.

The public and private key pair is generated with OpenSSL where the algorithm and key size is specified. Finally the U-Boot Raspberry Pi 3 dtb file (bcm2837-rpi-3-b.dtb) and the necessary kernel image and kernel dtb file (bcm2710-rpi-3-b.dtb) are placed in the folder location where the FIT image will be compiled. All these files are used to generate the FIT image as well as the public key implemented in the U-Boot dtb file by using the U-Boot tool *mkimage* as seen in Figure B.1.

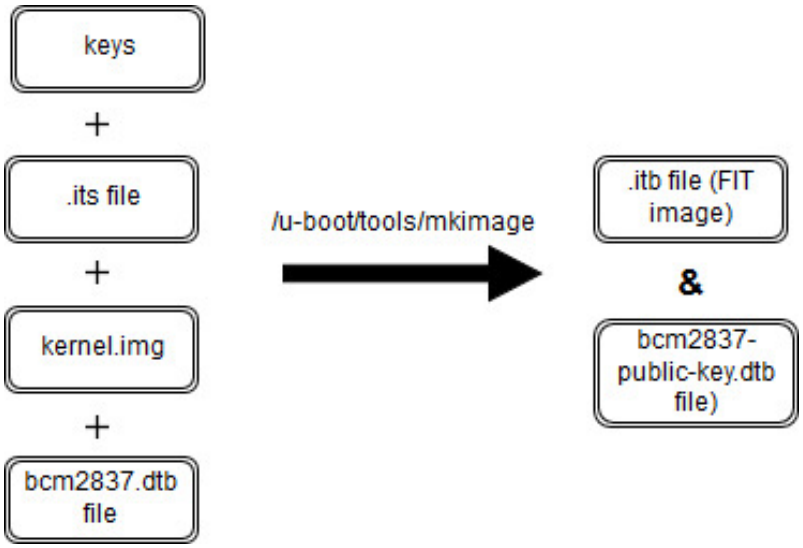

Figure B.1: The FIT image and public key dtb file creation process

At this stage, the public key which is a dtb file has to be used when cross-compiling U-Boot, a problem, however, is that the source file (.dts) for the new dtb file is necessary for this process. So the tool DTC is used to reverse compile the dtb file into a dts file and the .dts file is then placed in the required location to compile the new U-Boot-dtb.bin file.

When the U-Boot-dtb.bin file and FIT image (.itb file) are acquired, we are ready to place these files in the boot partition on the MicroSD card. The next step is to connect U-Boot to the FIT image in order for U-Boot to read and load the FIT image. In this case, there are three alternatives, alternative 1 is to create a script file (.scr) for U-Boot which reads all necessary commands to load the FIT image and boot up with it. This option was, unfortunately, causing too much time overhead and was replaced with the second option. Option 2 is to code the configurations into the environment variables of U-Boot and generate a uboot.env file which is read at each boot-up instance. This method was preferred and used. Option 3 which normally works in systems which use U-Boot by default is to create a uEnv.txt file which can be seen as a script file and is read as soon as U-Boot initiates to set its environment variable. The uEnv.txt file was however not an already supported feature for the U-Boot configuration of Raspberry Pi. The option is to program this feature into U-Boots environment variable section but since the second option worked perfectly fine, we preferred to omit the remaining options (1 and 3).

B.2 Verified U-boot for BeagleBone black

The first step to implement a verified version of U-boot in BeagleBone is to obtain the source code of U-boot, enable the option of verified boot and build it. We need a device tree file to insert the public key into in order to verify the kernel image.

The next step is to obtain the kernel source code and build a suitable version for

Beaglebone Black. The configuration of the kernel is unimportant for our purposes. The outcome of this step will be two files, the image (kernel) and the dtb (device tree blob) specific to the Beaglebone Black.

Next, we need to configure the configuration file .its (image tree source) with the type of kernel image and the algorithms that we will use for hashing (SHA-256) and singing (RSA). Also, we need to create a pair of public and private key that will be used for verification purposes. We use the open source OpenSSL 1.0.2g to create the keys.

Next, we sign the kernel with a private key which is created in the previous step. We use the *mkimage* tool which is provided by the U-boot project for singing the kernel and at the same time to put the public key into the U-boot device tree file.

Finally, we re-build the U-boot overridden with the device tree with the one that contains the public key from the previous step.

The files that have been produced are the following:
- MLO - 1st stage bootloader
- u-boot.img - 2nd stage bootloader
- image.fit - Image that contain the signed kernel & device tree

A more detailed guide for configuring the verified U-boot for BeagleBone Black can be found here [92].

B.3 Boot time measurement

B.3.1 Measure U-Boot boot time

For purposes of measurement, the boot time of the SPL (Secondary Program Loader) a module pre-exist in core built tree of the U-Boot. The module is enabled with the macro CONFIG_BOOTSTAGE and is using a 32 bit monotonic counter; it produces a time summary at end of the SPL in microseconds.

B.3.2 Measure kernel boot time

To measure the boot time of the kernel we enable the macro CONFIG_PRINTK_TIME in the configuration file. The module simply keeps time-stamps of the major events of the kernel; the time-stamps can be printed into the serial output or produce a report in the /sys/log file system.