Software Authenticity Protection in Smartphones using ARM Trustzone

Pedro Miguel dos Santos Mendonça

Thesis to obtain the Master of Science Degree in

Information Systems and Software Engineering

Supervisor(s): Prof. Miguel Nuno Dias Alves Pupo Correia

Examination Committee

Chairperson: Prof. José Carlos Martins Delgado
Supervisor: Prof. Miguel Nuno Dias Alves Pupo Correia
Member of the Committee: Prof. Ibéria Vitória de Sousa Medeiros

October 2017

Acknowledgments

I would like to express my gratitude to Professor Miguel Pupo Correia for guiding me throughout my work, as well as Sileshi Demesie Yalew for all the help he gave me, as without them this would not have been possible.

I would also like to thank both my family and friends for their support.

Resumo

O actual aumento de pirataria de software, em conjunto com o problema de malicious host, levou ao desenvolvimente de várias técnicas de tamperproof de software com o objectivo de garantir a integridade do software. Neste projeto desenvolvemos um serviço que faz uso da extensão ARM Trustzone para verificar a autenticidade de aplicações a correr num smartphone. Este serviço usa uma combinação de técnicas (hashes criptográficas, watermarks estáticas, watermarks dinâmicas) para alcançar o objectivo desejado. Este serviço foi implementado numa placa de hardware que emula um dispositivo móvel, a qual usamos para fazer a avaliação experimental do nosso serviço.

Palavras-chave: Autenticidade de Software, ARM Trustzone, Resistência a Modificações de Software, Segurança de Software, Aplicações para Smartphones

Abstract

The current increase of software piracy in conjunction with the malicious host problem lead to the development of several tamperproofing techniques that aim to ensure the integrity of software. In this project we provide a service that leverages the ARM Trustzone extension to verify the authenticity of the applications running in a smartphone. This service uses a combination of techniques (cryptographic hashes, static watermarking and dynamic watermarking) to achieve the desired goal. The service was implemented in a hardware board that emulates a mobile device, which was used perform the experimental evaluation of the service.

Keywords: Software Authenticity, ARM Trustzone, Software Tamper Resistance, Software Security, Smartphone Applications

Contents

Acknowledgments . . . iii

Resumo . . . v

Abstract . . . vii

List of Tables . . . xiii

List of Figures . . . xv

Nomenclature . . . 1

Glossary . . . 1

1 Introduction **1**

1.1 Objectives . . . 1

1.2 Thesis Outline . . . 2

2 Related Work **3**

2.1 Software Authenticity and Integrity at Deployment . . . 3

2.1.1 Software Piracy . . . 3

2.1.2 Checksums and Digital Signatures . . . 5

2.1.3 Static Watermarking . . . 6

2.1.4 Obfuscation . . . 8

2.2 Software Authenticity and Integrity in Runtime . . . 10

2.2.1 Dynamic Watermarking . . . 10

2.2.2 Protection with Dongles . . . 11

2.2.3 Dynamic Tamperproofing Techniques . . . 12

2.3 Trusted Execution Environments . . . 14

2.3.1 ARM Trustzone . . . 15

2.4 Trusted Computing and Attestation . . . 17

2.4.1 Remote attestation . . . 18

2.5 Measuring Protection . . . 19

3 Authenticity Protection Service **21**
3.1 Threat Model and Assumptions . . . 21
3.2 Architecture . . . 22
3.3 Authenticity Verification Process . . . 23
3.3.1 Verification Key . . . 23
3.3.2 Cryptographic Hash . . . 24
3.3.3 Static Watermark . . . 25
3.3.4 Dynamic Watermark . . . 26
3.3.5 Overview of the Authenticity Verification Process . . . 27
3.4 Normal World Integrity Verification . . . 28
3.4.1 Trusted Boot . . . 29
3.4.2 System Integrity Verification . . . 29
3.4.3 Tracer Integrity Verification . . . 29

4 Service Implementation **31**
4.1 Runtime Environment . . . 31
4.2 Service Components . . . 31
4.2.1 Secure Storage . . . 32
4.2.2 System Verifier – Trusted Boot . . . 32
4.2.3 Application Verifier . . . 32
4.2.4 Syscall Tracer . . . 33

5 Experimental Evaluation **35**
5.1 Authenticity Verification . . . 35
5.1.1 Hashes . . . 36
5.1.2 Static Watermaks . . . 36
5.1.3 Dynamic Watermaks . . . 37
5.2 Performance Overhead . . . 38
5.2.1 Hashes . . . 38
5.2.2 Static Watermarks . . . 39
5.2.3 Dynamic Watermarks . . . 40
5.2.4 Integrity Verification Process . . . 40
5.3 Tradeoffs on Detection Techniques . . . 41

6 Conclusions **43**
6.1 Summary and Contributions . . . 43

6.2 Future Work 44

Bibliography **45**

List of Tables

5.1 Evaluation of dynamic watermarking. 38
5.2 Detection rate for real repackaged apps. 39
5.3 Time to create hashes. 39
5.4 Time to do static watermarking. 40
5.5 Time to do trace conversion and comparison. 40
5.6 Time to do integrity verification. 41
5.7 Comparison of the authenticity verification process of the three techniques. . . . 41
5.8 Summary comparison between the three techniques. 41

List of Figures

2.1 Example of watermarking 7

2.2 Overview of the ARM Trustzone 16

3.1 Architecture of a mobile device running our service. 22

3.2 Static watermark validation data in the VK. 25

3.3 Syscalls trace example. 26

3.4 Authenticity verification scheme. 27

Chapter 1

Introduction

Proving that the software being executed is authentic – is the software produced by a certain company – is important in fighting against software piracy, which according to [1] in 2002 it was already a "12 billion dollar per year industry". However this is also a challenging problem if the software is executed in the device of an untrusted user, or if the device was compromised by a hacker or malware, as the device controls the software: it can modify it, block some system calls, its communication, etc. This is known as the *malicious host problem*, and is currently unsolvable, only mitigable [1].

In this project we were especially concerned with the use of authentic software to access a secure software component that has been developed in project PCAS[1]. This component, the Secure Portable Device (SPD), is a kind of a sleeve with a microcontroller that will be connected to a smartphone. The SPD will be used to store personal data, e.g., personal health records, so it is critical that it is not accessed by modified applications running in the phone.

This work was partially published as:
TruApp: A TrustZone-based Authenticity Detection Service for Mobile Apps
Sileshi Yalew, Pedro Mendonça, Gerald Maguire, Seif Haridi and Miguel Correia
IEEE 13th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)
October 2017

1.1 Objectives

The objective of the project is to ensure the authenticity of applications running in a smartphone that want to communicate with the SPD. For that purpose we explored software protection

[1] https://www.pcas-project.eu

mechanisms created to protect software from piracy - from being used when copied illegally. These mechanisms involve using *dongles*, i.e., small hardware devices attacked to PCs using USB ports or other interfaces [2]. Dongles can help protecting an application in several ways: by authenticating the application periodically; by storing cryptographic keys necessary to decrypt parts of the application; by storing parts of the application code; etc. We explored these mechanisms but with a different objective: proving that the software being executed is authentic.

Moreover, we did not use dongles but a security extension available today in ARM processors: the Trustzone [3, 4, 5, 6]. This extension allows software running in those processors to be divided in two execution environments: the *normal world* where untrusted applications run alongside the *rich operating system* (e.g., Android), and the *secure world*, where trusted components that are relevant security-wise run. Programs running in the normal world cannot access the secure world, but the contrary is possible. The idea will be to run the code that would normally run in the dongle in the secure world, whereas the application will run the normal world.

The mechanism was implemented in a Freescale i.MX53 QSB development board running Android, containing an ARM processor with Trustzone.

1.2 Thesis Outline

The rest of this document is organized as follows: Chapter 2 presents the current state of the art on various methods that aim to ensure software authenticity and software tamper resistance. It also talks about the technologies that we used in order to implement our service. Chapter 3 presents the architecture of our service. Chapter 4 contains the implementation of the service. Chapter 5 corresponds to the experimental evaluation of our service performed on an hardware board emulating a mobile device using ARM Trustzone. Finally, in Chapter 6 we present the conclusion to our work.

Chapter 2

Related Work

Nowadays there are already various methods that aim to ensure software authenticity and software tamper resistance. The aim of this chapter is to present these methods and to analyze them in order to better understand which ones are the most useful and practical for the implementation of this project. Section 2.1 presents the state of the art in software authenticity verification, and software integrity and tamper resistance at deployment. In Section 2.2 we expand on the previous section, with techniques developed for runtime verification. Section 2.3 presents the ARM Trustzone and its uses, and briefly compares it to other trusted execution environments from other manufacturers. In Section 2.4 we talk about software attestation. Finally, Section 2.5 presents methods for measuring the protection of software.

2.1 Software Authenticity and Integrity at Deployment

Software authenticity verification is concerned with ensuring that a software is genuine; and *Software integrity verification* is a sub-problem of the previous, being concerned with the verification that the software was not modified in an unauthorized way by a third party. One of the main goals of this project is to ensure the authenticity of the software running in a smartphone.

This section presents a few methods that are currently used to ensure the integrity of the software, and how to measure the software authenticity and integrity during the deployment of the software, i.e., during its installation or before executing the software.

2.1.1 Software Piracy

Software tampering is defined as unwanted or unauthorized software modification [7]. It can be performed by end users that modify the code manually, or by other automated programs. The protection methods that seek to prevent the modification of the code are commonly referred to

as *tamperproofing* [8]. If a tamperproofing mechanism detects an unwanted modification in the code it will prevent the program from running. These methods can be implemented directly in the program, or can be external to it.

In order to better understand these tamperproofing methods and what they are precisely trying to accomplish, first we need to get acquainted with the various types of software piracy that are most common nowadays, and what vulnerabilities in the code they try to exploit.

Software piracy is both the illegal copying and resale of applications, and according to [1] and [7] in 2002 it was already a 12 billion dollar per year industry, leading to losing revenue worth billions of dollars every year. It is therefore of extreme importance to combat software piracy. The more commonly found types of piracy include [7]:

- *Crack and serials.* The performer of this attack, usually referred to as *cracker*, obtains a legal copy of the software he wants to attack and breaks the protective methods implemented by the developers [9]. The legal copy is obtained by using a license code or applying a patch which undoes its protection. The attacker then modifies the program by removing its protection and creates copies referred to as *cracked software*, which are then illegally distributed. It is one of most widely known and used type of piracy due to the high number of available software debugging and editing tools that target the specific parts of the programs that are used for the protection.

- *Softlifting and hard disk loading.* Software usually has a license that defines, among other things, the number of computers in which the software can be installed. Both softlifting and hard disk loading consist of installing the software in more computers than is allowed by its license. Softlifting consists of installing a legally obtained copyrighted software in more computers than allowed, while in hard disk loading the software is directly written on the hard disk, circumventing the installation process.

- *Software counterfeiting.* Copyrighted software is illegally copied and duplicated, but is distributed while appearing to be genuine. These copies usually have their anti-tamper protections removed or have malicious code inserted into them.

- *Mischanneling.* This is a type of piracy to which software cannot be protected from, as it consist in the use of the software for a purpose that it was not intended to.

- *Reverse engineering.* Reverse engineering consists of using tools such as debuggers, disassemblers, and decompilers [7], as well as manual methods, to observe the behavior of a program and try to recreate its source code, or understand some aspects of its functioning

[1]. It can be used to identify the various components of the code and how they interact with each other [10].

- *Tampering.* Expanding on what was said in the beginning of this subsection, tampering can sometimes occur in conjunction with other forms of attack, as described in [1], where a reverse engineer tampers with code in order to extract modules of interest.

- *Hardware techniques for breaking copy protection.* These include software execution trace analysis, mod-chip spoofing devices, and machine emulator [7]. With software execution trace analysis an attacker can obtain information on a software by logging and analyzing software traces, since most programming primitives are predictable even if they are encrypted. Mod-chips are used to record and replay memory and bus transactions, as well as hijacking the signal from another device. Emulators can execute software without authorization.

In summary [8]: a software vendor applies some sort of protection mechanism to the source code of its application so that the compiled code maintains this mechanism; a software pirate will try to decompile the code to obtain a partial source code, and by using methods such as reverse engineering tools he removes the protection and obtains an almost identical source code to the software vendor; he can then compile the code and sell it illegally.

With this in mind, we will go over the state of the art in protection mechanisms against software piracy in the remaining of this section, as well as the next one.

2.1.2 Checksums and Digital Signatures

The use of *checksums* is a simple and straightforward method for verifying the integrity of a software. It compares a program or a section of code to what the original one is supposed to be [8]. The most commonly found way to do this is to compute a hash of the program or code and compare it to some baseline. A *checksum file* contains the computed hash of the original program and can be used to verify the integrity.

This method is widely used by code-hosting web-sites, to allow the user to check the authenticity of the software that is downloaded. Alongside the software, the web-site also provides the checksum file. The user then computes the hash of the downloaded software and compares it to the value contained in the checksum file.

Since classical hash functions are not collision resistant they do not give enough guaranties that the software was not tampered with. Therefore, *cryptographic hash functions* are often used. Among these functions, those that are commonly used include MD5 and SHA-1, although

it is recommended as of 2012 to use other functions such as SHA-2 or SHA-3 [11].

In the case of automated checksum verification, where the program calculates its own checksum, there is the drawback of an attacker being able to identify the part of the code that performs this, since it is an operation that is not typical and sticks outs, and use this information to exploit the system [12].

Digital signatures [13, 14] are used to verify the authenticity of a message or document that is transmitted between users. The signature is capable of ensuring the following:

- *Authentication.* The message was sent by the sender.
- *Integrity.* The message was not altered during transmission.
- *Non-repudiation.* The sender can not deny having sent the message.

Digital signatures use a public-key cryptosystem where the users have a pair of public and private keys. To create the digital signature the sender first creates a signature of the message using a hash function. This signature is encrypted using the senders private key. The receiver deciphers the signature with the senders public key, and by computing the hash of the message, he can compare it with the received signature to verify its authenticity.

Digital signatures are much adopted today to ensure the authenticity and integrity of applications transferred through the Internet. For example, in Java, applets can be executed or not depending on being correctly signed [15]. Moreover, applets can get different permissions depending on their signature (e.g., they may or may not be able to access the Internet or the file system).

It is possible to inject malicious code into Android applications without damaging the digital signatures [16]. The authors of that paper demonstrate how to enable *TruStores*, which is an application certification system for Android.

2.1.3 Static Watermarking

Software watermarking is a mechanism used to enable software developers to prove their ownership over their intellectual property [8]. It accomplishes this by embedding a copyright notice in the software code [1]. The copyright notice can then be extracted from the program [7] for verification.

In order to accomplish this there are a few properties that the watermark needs to ensure [1], [8]:

- It needs to be resilient to both commonly occurring transformations of the software (e.g., software updates) as well as dewatermarking attacks, which will be explained ahead.

- It can be easily located and extracted from the software for verification.
- Embedding the watermarking into the software must not affect its performance, nor change any statistical properties.
- There has to be a property that shows that its presence is the result of deliberate actions.

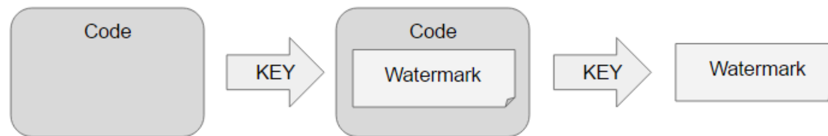

Figure 2.1: Example of watermarking

Figure 2.1 [1] illustrates the main idea of watermarking: the developer of the software embeds the watermark in his program with, for example, the use of a secret key. Whoever wants to verify the ownership of the program needs to extract the watermark with the key, which then confirms the ownership.

A technique that is also related to watermarking is *fingerprinting* [7]. In fingerprinting the watermark is different for each copy of the software. This adds the benefit of being able to identify not only the software, but also the system that used said software.

It is necessary to understand the different types of watermarking attacks that are possible in order to evaluate the resilience of a watermarking technique. Such attacks include [7]:

- *Additive attack*: In these types of attacks, the attacker adds his own watermark to an already watermarked program, with the aim of overwriting the original. To combat this attack it is therefore necessary to detect which watermark precedes which.
- *Distortive attack*: An attacker applies a sequence of semantic-preserving transformations to the program, trying to have it distorted to a point where the watermark is no longer recognizable, while having the program not become so degraded that it no longer is of use to the attacker.
- *Collusive attack*: An attacker with access to several copies of a program, each with a different fingerprint, can compare them in order to locate said fingerprints and attempt to remove them.

In *static watermarking techniques* the watermark can be stored in the application executable or the source code. The watermark is divided in two types: data watermark and code watermark. In a data watermark the watermark is stored in some data structure of the program; it is common

for a program variable to contain a copyright string. A code watermark uses specific rules for formatting the code. It leverages the redundancy of the code and the indenpendability of code statements to perform a rearrangement of the code following some rule, such as lexicographic ordering [8]. In this case the watermark is verified by checking that all methods of the code are in lexicographic order. In practical cases, it is necessary to choose a rule that is difficult to perceive by simply examining the code.

2.1.4 Obfuscation

To end this section we will talk about obfuscation, since it is of extreme importance in aiding tamperproof techniques, as it can be used to prevent software attackers from distinguishing the parts of the code that implement tamperproofing from the rest of the code [8].

In general, obfuscation is a set of techniques that are used as a deterrent to reverse engineering. It aims at transforming a program into a more complex one while maintaining its functionality [7], making it harder to understand its behavior either through static or dynamic analysis [17]. This new program is considerably harder to reverse engineer than the original. Since given enough time and resources an attacker can eventually reverse engineer any code, obfuscation aims at making it not economically attractive to do so. Although security through obscurity is viewed with disdain in the security community, it is the strongest technique in regard to preventing reverse engineering attacks [7] [8] [1].

In obfuscation, a set of transformations is applied to the code that seeks to make it harder to read and understand, while maintaining an equivalent execution as the original. The obfuscation transformations and the obfuscated program resulting from those transformations need to ensure a set of properties [1]:

- The obfuscated program has the same observable behavior as the original.
- Reverse engineering the obfuscated program is strictly more time consuming than the original.
- It is either difficult or extremely time consuming to reverse the transformations applied to the program.
- The statistical properties of the obfuscated code are similar to those of the original code.
- The difference in the execution time and space penalties resultant from the transformations between the original and obfuscated program is minimal.

In [1] black-box study is defined as the study of the input-output relations of a program, and white-box study is decompiling and examining the code of program. The upper bound of the time needed to reverse engineer a program depends on the black-box study of said program, and the time needed to create a new program with the same behavior. The goal of obfuscation is therefore to create a program in which the white-box study provides no useful information, and the time required to reverse engineer it approaches the upper bound of the original program.

Obfuscation can then be divided into different categories: lexical obfuscation, data obfuscation, and control obfuscation.

In *lexical obfuscation* only the lexical structure of the code is modified, which has no performance overhead [18]. A typical example is to remove commentaries and change the names of variables and methods in the code to a random string of characters. These transformations make it harder for an attacker to understand the code and guess its functionality , but given enough time a reverse engineer will be able to circumvent the random identifiers and deduct their meaning as well as what the code is supposed to do [1], [7].

In *data obfuscation* the data structures are obfuscated. This is accomplished by representing a data structure using other different forms and representations. Basic examples include splitting an integer into two integers, and having its value be calculated by adding the two integers [8], or having a boolean be represented by two shorts, each being either 0 or 1, and the boolean logic operations are performed on these shorts [1].

In *control obfuscation* the control flows of the program and the individual functions are made to be hard to understand. These techniques typically use *opaque predicates*, in which conditional statements use predicates which evaluation is known prior to obfuscating the code, and usually evaluate to either true or false. The idea is to have the correct code on the branch of the conditional statement that we know will be executed, while having dummy code on the other branch. Sometimes predicates that can evaluate to either true or false are also used. In this case both branches of the conditional statement have code that while appearing different, have identical execution. It is important that this does not create excessive computational overhead. The resilience of these obfuscation transformations is directly related to the resilience of the opaque predicates used, therefore it is necessary to create strong predicates. These predicates need to be resistant to static analyzers. If the analyzer detects that the predicate always evaluates to true or false, it defeats the purpose of this technique.

2.2 Software Authenticity and Integrity in Runtime

Methods that verify authenticity and integrity during deployment alone may not be sufficient depending on the type and complexity of the software that we are trying to protect. Therefore this section will introduce methods that ensure integrity in runtime, and how to verify authenticity and integrity during runtime, adding a new layer of protection to the software.

2.2.1 Dynamic Watermarking

Dynamic watermarking techniques, in contrast to the previously introduced static watermarks, create the watermark during runtime. The dynamic state of the program is used to represent the watermark [8].

The watermark is generated by having the watermarked program execute with a predetermined and rarely occurring set of inputs, defined by its developer. The dynamic state that the program reaches after executing with these inputs is used to represent the watermark.

Dynamic watermarking techniques are divided in three categories:

- *Easter Egg Watermarks.* These techniques create a watermark that is perceptible to the extractor immediately after the special input sequence is entered. Usually a message is displayed containing the copyright message. The main problem with this approach is that since the watermark is easy to locate it provides an easy point of attack.

- *Execution Trace Watermarks.* The watermark is embedded in the trace of the program. By monitoring some property of the address trace of the program or the sequence of operations that are executed the watermark can be extracted.

- *Data Structure Watermarks.* The watermark is embedded in the state of the program. By evaluating the values of the program variables (global, heap, stack, etc.) the watermark can be extracted.

Both execution trace watermarks and data structure watermarks are susceptible to obfuscation techniques that compromise their watermarks by destroying the dynamic state while maintaining semantic equivalence.

Since the behavior of the program can be examined while running with the set of inputs, it is a more reliable solution than static watermarking techniques [7].

Some challenges regarding dynamic watermarking include the fact that if the extractor of the watermark is not familiar with the method used by the watermarked program, then the program needs to supply some information to the extractor on how to find the watermark. This

information can be exploited by attackers. A system where the extractor and the watermarked program are developed by the same company circumvents this problem, since the method to extract the watermark is already defined, and the program does not need to supply any extra information.

2.2.2 Protection with Dongles

A dongle is a hardware component used with the goal of strengthening software copy protection [2]. The software vendor supplies the costumer with the dongle alongside the software, and the software will not install or run without the presence of the dongle. The dongle connects to the computer through an I/O port, such as USB or parallel port, and the software communicates with it through the dongle API. The software queries the dongle at regular intervals, and if it detects that the dongle is not connected to the computer, or the dongle gives an unexpected answer, it will stop running.

The usual components found in a dongle are:

- unique serial number
- unique software vendor identification number
- access password
- persistent memory
- asymmetric encryption engine and key storage

The basic way a dongle works is to have the software send a nonce to the dongle. The dongle then encrypts this nonce with its encryption key and sends its to the software, which then decrypts the nonce to verify that it was sent by the dongle.

The dongle itself is not the point of attack in this system, it is the interaction between the software and the dongle API: an attacker can just remove the part of the code that checks for the presence of the dongle to circumvent all the protection. A typical solution to this is to have the dongle contain part of the code needed for the program to run. Even if an attacker removes the communication with the dongle from the code, the program still needs the part of the code that is in the dongle.

Some software only checks for the presence of the dongle on start-up, as opposed to regular intervals. This raises the problem that a user can use a single dongle to run the program on any number of different computers.

2.2.3 Dynamic Tamperproofing Techniques

A *self-checking* program [17] is a program that is able to detect unwanted modifications to itself, either during start-up or while running, triggering an appropriate response, usually stopping the execution. This approach is commonly used in conjunction with other tamperproofing techniques, such as obfuscation and watermarking, in order to increase its protection level, since by itself it is not sufficient to protect software.

When developing a self-checking mechanism there are two angles of attack that need to be considered: *discovery* and *disablement*.

Discovery methods aim at identifying the parts of the code that perform the self-check. They include:

- *Static Inspection.* A static analysis of the code. It is usually thwarted by the use of obfuscation techniques.

- *Debuggers.* By performing dynamic analysis of the program they try to detect the memory references reads that the program performs in order to perform the self-check.

- *Generalization.* When the self-checking mechanism is divided into various components, if one of them is found by an attacker, it may be possible to use information regarding that component to discover the remaining ones. To avoid this, the various components should be customized to either do different tasks, or do the same task but with different code.

- *Collusion.* By comparing various copies of the program it may be possible to discover the self-checking code. Software developers sometimes use a mechanism called *customization* where each copy of the program has a significant difference compared to the others, to foil this kind of attack.

If any point of attack is discovered by an attacker, they can try to disable it in order to shut down the protection mechanism - disablement. As such, developers should aim to eliminate single points of failure in the self-checking mechanism.

A solution to the single point of failure problem is discussed in [9] where the authors propose the use of *guards* to prevent these single points of failure. Instead of having the code that tamperproofs the program in a single component, a number of smaller components called guards work together in a network. Each guard has a specific task, which adds redundancy to the system, as well as increasing the protection as a whole by having the guards perform integrity checks on each other. In this scheme, an attacker disabling or removing a single guard is detected by the remaining ones, and they prevent the program from running. By customizing each guard, the

information available to an attacker by discovering a single guard does not disclose the location of the remaining. therefore, in order to completely disable the protection, all guards need to be discovered and removed.

The authors of [12] propose a new mechanism called *oblivious hashing*, which works by creating a hash based on the actual execution of the program. Since the trace of a program depends on a series of instructions, memory locations and memory configuration, as well as an instruction counter and its initial configuration, a hash of this trace is used as a signature for the behavior of the whole program or a single function. This idea takes advantage of the fact that a slight modification in the code would generate a different hash, given that enough information about the execution is present in the trace. The name "Oblivious Hashing" comes from the fact that an attacker is oblivious to the fact that the program is calculating its own hash during its execution. This is accomplished by injecting the hashing instructions in the code. Since these instruction read and write data, like any other instruction in the original program, in order to calculate the hash, if injected with care, they can blend in with the rest of the code. Although in theory using Oblivious Hashing on the whole program would give the highest level of security, due to the fact that some parts of the code may depends on external values, such as time or user identification, or even unpredictable data, such as return values of system calls, those parts would generate arbitrary hashes that could not be used. Therefore it is important to correctly choose what parts of the code to fingerprint.

Self-Checksumming is the act of a program performing verification actions on itself during runtime to detect any code modification, validating its integrity [9] [19]. The program calculates the hash of the critical parts of its code, and compares it to the expected value. The authors of [20] present an attack that circumvents any self-checksumming mechanism, by creating and modifying a copy of the original program, and running this copy in a modified kernel. The kernel redirects the data reads performed by the checksum methods to the data of the original program. This is possible because self-checksumming assumes that the memory is *von Neumann*, i.e., the data and instruction memory space is shared, but the attack creates a virtual *Harvard* architecture, i.e., the data memory and instructions memory are separate. In [19] a method to strengthen self-checksumming is introduced. This methods aims at detecting any violation of the von Neumann assumption, leveraging self-modifying code. Self-cheksumming relies on three assumptions:

- Through the use of code obfuscation, static analysis of code is hard, preventing an attacker from identifying and therefore altering or removing the parts of the code that perform the

checksum.

- An attacker wants to run the program without any significant downgrade to the performance. Since the program can run in untrusted platforms with untrusted users, it is possible to perform code manipulations that are undetectable, but incur serious performance downgrades.

- Any program that uses self-checksumming assumes a von Neumann architecture. As shown above, if this assumption is broken it is possible to break any self-checksumming mechanism.

If any of the previous assumptions is violated, an attacker can successfully defeat the protection of the program. The focus of this paper is on the third assumption, and the authors present an algorithm that enables the self-checksumming program to detect if the memory appears to be Harvard while on a von Neumann machine, signifying an attempt of attack. A process can identify the underlying memory architecture by following the steps defined in [19]:

1. Overwrite an existing instruction with a new one that alters the execution of the code.

2. Read the instruction using data memory reads.

3. Execute the instruction.

In a von Neumann architecture, the first step changes both the data memory and the instruction memory, resulting in the value read by the second step and the instruction executed in the third step to both correspond to the new instruction. On the contrary, on a Harvard architecture, since the first step modifies only the data memory, the second step will read the new instruction while the third step will execute the original instruction. Since the instructions produce distinguishable executions, it is possible to determine what the underlying architecture is. By using this self-modifying code the program can detect if the memory is Havard, violating the von Neumann assumption, and therefore react accordingly. In conjunction with the checksums there is an higher integrity guarantee.

2.3 Trusted Execution Environments

A Trusted Execution Environment (TEE) is an environment contained in the secure area of the main processor that runs alongside the normal environment where the rich operating system and the normal applications run [21] [22]. The data in the TEE has its integrity and confidentiality protected by the security mechanisms available, since the TEE is isolated from the normal

environment. By developing applications that have their sensitive operations restricted to the TEE, the security of their sensitive data is guaranteed. The applications running in the TEE are referred as *trusted applications*, and each is independent from the other, not sharing any data between them. These applications have access to the TEE internal API which contains secure resources to use. An application running in the normal environment can use the TEE client API to request access to certain data inside the TEE, which the TEE can allow access to or not.

In this section we will talk about one implementation of the TEE, the ARM Trustzone, and some of its applications.

2.3.1 ARM Trustzone

ARM processors are based on the RISC architecture [3], which results in fewer transistors being needed compared to processors that are typically found in computers. ARM processors are therefore ideal for portable and small devices, like smartphones or tablets, since they have reduced power usage and dissipate less energy.

ARM Trustzone is a hardware security extension that covers the processor, memory and peripherals of an ARM System-on-Chip, enabling trusted computing [4] [5]. The Trustzone provides code isolation and security services to the trusted applications running in it. It accomplishes this by having it execution environment divided into two environments: the secure world where trusted code runs, and the normal world where untrusted code is executed. The physical core of the processor is divided into two virtual cores, each operating in each one of the execution environments.

The secure world is isolated from the normal world, each having its own memory addresses and privileges. While code running in the normal world can't access the memory addresses of the secure world, the contrary is possible under certain circumstances. Trusted applications have access to security services provided by the secure world. Therefore, the Trustzone provides secure execution for embedded applications.

The Trustzone has a special processor bit - NS-bit - that distinguishes in which world the processor is executing. Since the processor can only execute in one world at a time, there is a secure monitor that performs the switches between the secure and normal world. The switch occurs through a secure monitor call or exceptions [3] [5]. Figure 2.2 [3, 4] illustrates how the execution environment is divided in parallel executions.

Each world has access to a Memory Management Unit (MMU) that performs the translation from their virtual memory to the physical memory [3]. The secure world MMU can create a

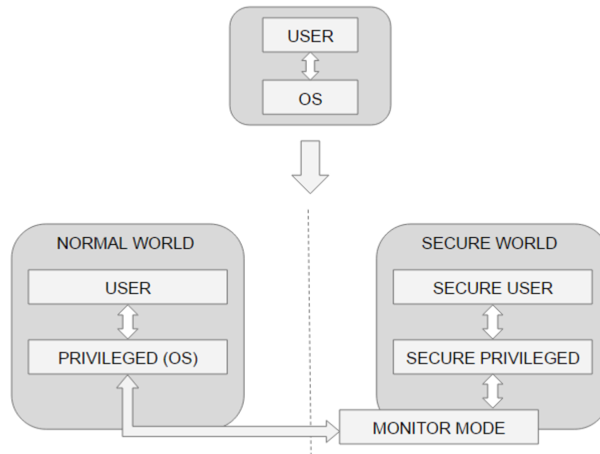

Figure 2.2: Overview of the ARM Trustzone

mapping to the normal world physical memory, giving the secure world access to the normal world memory. This mapping needs to have the NS-bit set to 1 to denote that the memory is unsafe, while the translations within the secure world memory have the NS-bit set to 0. If an application tries to access memory for which the current world does not have permission to access, an external system abort trap occurs [6].

Alongside the Trustzone CPU that runs the trusted applications isolated from the normal world, the System-on-Chip contains a secure ROM, a one-time programmable memory used to store cryptographic keys, secure RAM and other peripherals that are accessible to the trusted applications [4].

In a normal use of the Trustzone, it is expected for the secure world to be used only for the sensitive and critical operations, with the remaining execution belonging to the normal world [6].

The Trustzone has some main features available to it that include being able to identify and authenticate the underlying platform, I/O access control, provides safe data storage, cryptographic keys and certificates management, and integrity checking [4].

While the operating system runs in the normal world and is therefore not guaranteed to be secured, the Trustzone can perform integrity checks to try and ensure its security. These integrity checks can occur before the operating system is booted, and while it is running focusing on the critical paths. Furthermore, some actions can be performed inside the secure world.

This approach by the ARM Trustzone offers a few key benefits [4]:

- The data contained in the chip is secured by this safe environment, enabling handling of sensitive data, which could not be done with an unsafe operating system.

- It provides fast memory access speed through full bus-bandwidth access.

- Since the Trustzone consists of both software and hardware elements, it provides flexibility in customization.
- The secure world can perform integrity checks on applications running in the normal world ensuring security even in untrusted applications.
- Reduced implementation risk and development costs.
- Easy certification of applications.
- Compatibility between different Trustzone System-on-Chips.
- System performance is minimally impacted.

While the Trustzone provides some security properties, it can be vulnerable to sophisticated and sustained attacks, and physical attacks [23]. The manufacturers need to take precautionary steps while developing their Trustzone-based systems.

Yalew et al. presented two services based on the TrustZone. T2Droid [24] is a service that leverages the Trustzone extension to perform a dynamic analysis on applications running in Android devices, by using traces of kernel system calls and Android API function calls to detect malware. DroidPosture [25] is a service that also leverages Trustzone to evaluate the level of trust that can be given to a mobile device.

2.4 Trusted Computing and Attestation

Trusted Computing (TC) are a set of specifications defined by the Trusted Computing Group[1] for the development of hardware and software with the goal of ensuring security, namely integrity and authenticity, of a system and the software executing in it, as well as being able to provide guarantees on this security to a challenging third party – attestation [26, 27]. TC is of extreme importance since the generally used security measures only work properly if the underlying system is secure [28]. This is achieved by having a Trusted Platform Module (TPM) embedded in the hardware of the underlying system that provides functionalities such as cryptography and protected storage. The TPM cannot be compromised by neither a malicious host nor user. Leveraging the TPM, Trusted Computing offers the following core mechanisms:

- *Secure boot.* The system can perform integrity checks on its own boot, to ensure that it boots into a trusted OS, terminating it if it suspects that the OS is compromised [27, 26]. The TPM contains a pair of public-private keys with which it signs a hash of the BIOS

[1] http://www.trustedcomputinggroup.org/

and stores it in its secure storage. On boot, it recomputes the hash and compares it with what was stored to ensure that it was not tampered with. Then the BIOS performs a similar procedure with the boot-loader, using its own pair of keys. The boot-loader then performs this task again with the operating system and its key-pair. At the end of the checks the operating system is confirmed to be untampered. Important to note that just because a system is trusted in some point in time does not guarantee that it is trusted later on [28] . This occurs because a system is vulnerable to attacks after the boot is complete. To counteract this it is important to offer strong isolation to prevent anything from altering the system after it has booted, and also prevent the a malicious applications from tampering the others [26].

- *Remote attestation.* A third party can ask the system or a software executing in it to provide a guarantee that it is authentic and untampered. The attestation of the integrity of the system boot is called *authenticated boot* or *trusted boot*, and unlike secure boot, the hashes are only calculated but not verified: the verification is performed by the third party. Remote attestation will be covered in higher detail further ahead in this section.

- *Sealing.* Data can be encrypted by the TPM [28] alongside information of the boot configuration at the time the data was encrypted. Only the TPM can decrypt this data, but it only shares it if the current configuration is the same as the stored one. This ensures that no one has access to the data if the configuration of the system is altered.

Although it would be useful to have the secure boot cover the configuration of both the operating system and the application executing in it, the secure boot usually only covers the steps up until the bootstrap loader. This occurs since in order for it to cover up to the application layer it would have to overcome a few challenges [27]:

- large variety of executable content in the intermediary steps.
- the order in which the contents are loaded is random.
- the checks at each step of the verification cause a significant increase in performance overhead.

A mechanism to extend the secure boot to the application layer in Linux is covered in [27].

2.4.1 Remote attestation

Remote attestation gives guarantees to a third party that the software executing in the computer that implements the trusted computing is authentic and was not tampered with [28]. In other

words, "attestation is the process of establishing the integrity of the platform" [29]. A common method for performing attestation [28] on a software is to have the third party – the challenger – have the initial state (the sequential computation of the configuration of the software) stored prior to the attestation. The challenger sends this initial state to the TPM that is then forwarded to the software through a secure channel. The TPM ensures that the software was loaded with the correct configuration through methods similar to the secure boot, and confirms this to the challenger through a secure communication channel between them.

The authors of [28] identify some drawbacks of the attestation system defined by the TCG, namely since verifying the trustworthiness of each platform configuration is not practical, it usually focus on mainstream configurations, making it so that some less popular operating systems or programs can be incompatible with this mechanism. They propose an alternative in which the attestation does not depend on the configuration of the hardware and software, but instead it depends on the properties offered. Their main idea is to have an extra component that translates those properties into platform configurations so that the challenger can attest the system. Details of this implementation can be found on their paper.

The authors of [29] state that these remote attestation methods are not compatible with Android operating systems due to their software stack having a unique virtual machine based architecture. They introduce the design for an integrity measurement mechanism tailored for Android. Refer to their work for detailed information.

2.5 Measuring Protection

There are currently no means by which we can completely and reliably measure the protection of software [2] [30] [31]. A piece of software can be considered secure until an attacker finds a new vulnerability in the code that he can exploit. Therefore, when trying to measure the protection of software, the usual methods use *threat models* that cover the techniques and attacks that an attacker can use to try to subvert the protection mechanisms. These models offer the guidelines needed to identify the possible points of attack of a software, in order to then evaluate if the protective mechanisms implemented in said software are sufficient to protect it.

Sandmark [31] is a tool developed with the goal of measuring the protection offered by the various common tamperproofing techniques being used regularly. Sandmark evaluates watermarking techniques according to the following criteria:

- *Data rate.* The ratio between the size of the watermark and the size of the program.
- *Embedding overhead.* Is the watermarked program considerably slower and larger than the

original.

- *False positive rate.* The probability of an invalid watermark being recognized as a valid watermark.
- *Stealthiness.* How difficult it is for an attacker to locate and attack the watermark.
- *Resilience to distortive and collusive attacks.* Is the watermark resilient to the distortive and collusive attacks described in section 2.1.3.

In [2] a model is proposed for the measurement of the security strength of software protected by a dongle. This methodology follows several steps:

1. Create a *defense pattern catalog* that lists the various defenses available (e.g., AES challenge-response, code obfuscation, proper error handling). Each of these defenses has a set of measurable attributes regarding the strength of defense provided. The defense is measured either by tools or manual checks.
2. Compile an *attack pattern catalog* containing the various possible attacks (e.g., remove dongle checks, memory tampering) and each step necessary to accomplish said attack.
3. Create *experimental cost-to-break functions* which are flow diagrams that represent how each attack from the attack pattern catalog is conducted. In each step of these diagrams there is an approximate time effort necessary to execute it. These times are defined using experimental knowledge. Each diagram will therefore have an approximate total time necessary to perform the attack depending on the times of each individual step.
4. Combine the previous into an *attack tree model* that links the possible attacks. By propagating the values of the cost-to-break functions through the tree, the estimated time to break the defenses is obtained, depending on the attacks and the defenses available. This model should be posteriorly validated by comparing the predicted values with field data.

Chapter 3

Authenticity Protection Service

In this chapter we go over the architecture we chose for our service. From the work seen in Chapter 2, we concluded that a single protection mechanism is most likely not to be enough in order to ensure the authenticity of an application. Our solution therefore uses a combination of software integrity verification using hashes and watermarking techniques, leveraging the security mechanisms available in the ARM Trustzone.

The goal of this project was to ensure that an application running in a smartphone is authentic, so that it can access the Secure Portable Device (SPD), although it can be extended to be compatible with other secure components, such as the Square Register developed by Square, Inc.[1]. These applications need to belong to an access list determined by the SPD or other secure component. To do so, we developed an *authenticity verification service*. The service runs in the secure world of the Trustzone, while the target application runs in the normal world. We define guidelines that the developers will have to follow so that their application is compatible with our service.

3.1 Threat Model and Assumptions

As stated previously, we leverage the mechanisms available in the ARM Trustzone to run most of our service in the secure world where it is isolated from the OS (i.e., Android). We assume that the code running in the secure world is therefore protected from unauthorized access from the normal world. We also assume that the code running in the secure is trustworthy, contrary to the code running in the normal world, as well as the underlying OS, which we assume to be untrusted, as it may be malicious, or otherwise compromised by hackers and/or malware.

We assume that both the service provider and the application vendors have a public-private

[1] https://squareup.com/global/en/pos

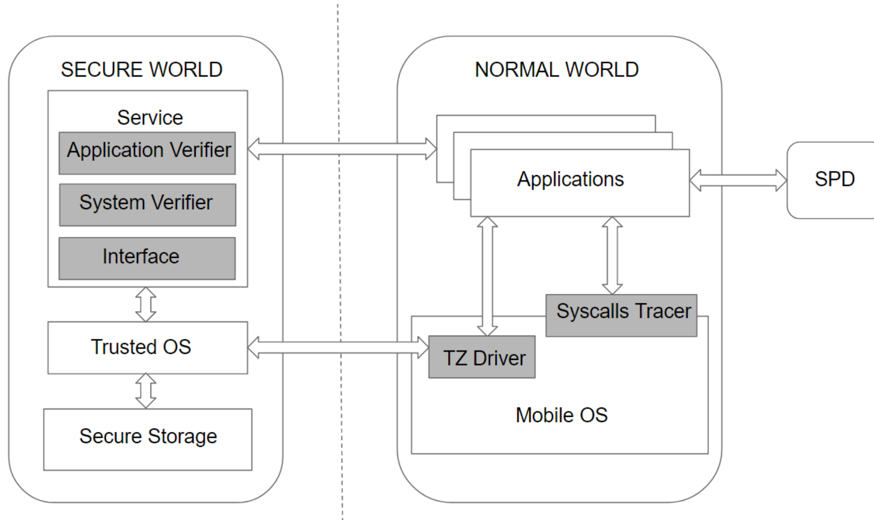

Figure 3.1: Architecture of a mobile device running our service.

key pair, with a secure key size (e.g., 3072 bits for RSA [32]). The service provider installs its public key in the service, while keeping the private key for itself. Furthermore, each instance of the service running in each smartphone has its own public-private key pair and public key certificate, which is generated by the service provider, stored in the secure storage of the Trustzone.

We assume the existence of a collision-resistant hash function [33], as well as a secure symmetric encryption system with cipher block chaining (CBC) mode [32].

3.2 Architecture

Our service uses a combination of hashes, static watermarks and dynamic watermarks to ensure the authenticity of the applications. For each one of these mechanisms, we need information on how to validate the authenticity, e.g., hash of the application to compare to. Therefore the application developers need to provide a *Verification Key* (VK), alongside the application, which contains such information. These mechanisms and the VK will be discussed in detail in Section 3.3.

Our service runs mostly in the secure world, but some components need to run in the normal world. Figure 3.1 depicts this architecture. Components in dark grey represent the components of the service. In the secure world, our service runs on top of a trusted OS which provides basic functions, such as file access and process management. There is also a secure storage in the secure world which our service uses to store important information (e.g., the VKs of various applications). The service itself is composed of 5 modules:

- *Application Verifier*: implements the set of techniques used to verify the authenticity of

an application.

- *System Verifier*: verifies the integrity of the components of our service that run in the normal world, since they are vulnerable to attacks.
- *Interface*: provides the interface between the normal world and our service. It receives and replies to requests from the applications. Furthermore it validates the data that is being transmitted to the secure world to prevent attacks such as code injection and buffer overflows, through the use of an existing framework that will be described in Section 4.
- *TZ Driver*: kernel driver that supports the communication between software running in the normal world and our service running in the secure world. It provides a shared memory buffer to transmit data between the worlds.
- *Syscalls Tracer*: intercepts and logs system calls on the kernel level made by the application running in the normal world. It is used in the dynamic watermark scheme.

Application Verifier, *System Verifier* and *Interface* run in the secure world, while *TZ Driver* and *Syscalls Tracer*, as well as the applications and the mobile OS, run in the normal world.

3.3 Authenticity Verification Process

In this section we will go over the process of the authenticity verification, and the main components involved in it. This mainly encompasses the *application verifier* module and the *interface* module of the architecture (Figure 3.1).

3.3.1 Verification Key

As briefly stated in Section 3.2, the developers of the applications need to provide a *Verification Key* alongside the application that contains the necessary information for our service to correctly verify the authenticity of said application. Since our verification process has three techniques which are used (hashes, static watermarks and dynamic watermarks), we need information on how to validate each one of them. The VK needs to contain information for all three of the methods, or if agreed upon by the service provider and the developer, it may contain only information for two of the used methods. The VK can either be created by the application developer, or the service provider, granted that the developers supply the service provider with the necessary information for creating it. The exact contents of the VK are discussed in each of the following sections (Sections 3.3.2, 3.3.3, 3.3.4).

It is of utmost importance that the VK follows the following security properties:

- *authenticity* – the VK must have been created either by the developer of the application, or our service provider. This ensures that an attacker is not able to create a false VK in order to have his application be classified as genuine.
- *integrity* – the contents of the VK itself can not be modified. The reason for this is the same as above – an attacker could modify the VK to have a non-genuine application appearing to be genuine.
- *confidentiality* – only the service or the entity that created the VK can have access to its contents. Since the VK contains sensitive data on how to validate the authenticity of the application, an attacker could obtain significant information on how to create a pirated version of application that would also be considered authentic by the service.

The solution we found for this problem is divided in two steps:

1. first the VK needs to be *digitally signed*. This signature can be obtain in either one of two ways. Upon request of the application provider, the service provider creates the signature of the VK using its own private key (refer to Section 3.1). Alternately, the application developer can sign the VK with its private key. In this case, the developer needs to also provide a public key certificate that is signed by the service provider with its private key.
2. lastly, the VK needs to be encrypted using hybrid encryption [34]. This means creating a random secret key K_s, encrypting the contents of the VK with it, and then encrypting K_s with the public key of the service instance. Therefore, only the service can decrypt K_s and then decrypt the contents of the VK. In our project we used the AES-256 algorithm for creating the secret key K_s.

The above steps guarantee the security properties defined earlier.

The VK can be either obtained directly with the application when it is downloaded from the Google Play Store, or can be directly supplied by the application developer upon request. Since the VK follows the security properties defined above, the service can validate it independently of where it was obtain from.

Once the service verifies the authenticity and integrity of any VK it obtains, it stores its contents in the secure storage of the Trustzone for later use. This bypasses the need of having to verify the authenticity every time it needs to access its contents.

3.3.2 Cryptographic Hash

The first technique we use to verify the authenticity of an application is cryptographic hashes. This means calculating the hash of the app using a collision-resistant hash function, and com-

```
46514319
198
70163407
166
22605143
222
67675468
114
71825437
230
660356
236
75684435
253
52784280
37
48179494
5
```

Figure 3.2: Static watermark validation data in the VK.

paring it to the expected value. This expected value is stored in the VK. Since the hash function is collision-resistant, any small change to the app itself leads to a different hash value. Therefore the hash is a very good indicative of the app being modified.

In our service, the *application verifier* module running in the secure world can access the resources of the normal world, and access the *Android app package* (APK) file stored in the internal memory of the device to calculate its hash.

3.3.3 Static Watermark

In our static watermarking scheme, we defined the watermark as being represented by the values of specific bytes in the app bytecode. The app developer would choose which bytes represent the watermark and store their positions and their values in a file in the VK. Figure 3.2 represents an hypothetical possibility for such information. The first line corresponds to the byte position and the next line corresponds to its value. This is repeated for as many bytes as the watermark uses.

The number of bytes to use to represent the watermark can be left to the app developer's discretion, although the higher the number of bytes used, and the more scattered they are, the lower the changes of two different applications having those same bytes in common. In Section 5.2.2 we study the performance overhead incurred from using different number of bytes.

In our service, the *application verifier* module reads the positions of the bytes from the VK, and compares the values of those bytes in the APK file stored in the normal world with the expected values. This is once again possible since the module running in the secure world has access to the resources of the normal world and can inspect the APK file.

```
clock_gettime(CLOCK_MONOTONIC, {8102, 682499704}) = 0
epoll_pwait(19, {{EPOLLIN, {u32=23, u64=23}}}, 16, -1, NULL, 8) = 1
recvfrom(23, "\1\0\0\0\0\0\0\315\0\0\0\0\0\0\0\247\244Q`\7\0\0\377\377\377\377\1\1\0\0"..., 2264,
clock_gettime(CLOCK_MONOTONIC, {8110, 286530969}) = 0
clock_gettime(CLOCK_MONOTONIC, {8110, 287298501}) = 0
epoll_ctl(19, EPOLL_CTL_ADD, 24, {EPOLLIN, {u32=24, u64=24}}) = 0
sendto(24, "\1\0\0\0\310`\214o\1\0\0\0\320f\263\247\0\247\244Q`\7\0\0\377\377\377\377\1\1\0\0"..., 64
clock_gettime(CLOCK_MONOTONIC, {8110, 291055838}) = 0
write(18, "W", 1)                       = 1
recvfrom(23, 0xb512e410, 2264, 64, 0, 0) = -1 EAGAIN (Try again)
clock_gettime(CLOCK_MONOTONIC, {8110, 293747138}) = 0
epoll_pwait(19, {{EPOLLIN, {u32=24, u64=24}}, {EPOLLIN, {u32=17, u64=17}}}, 16, 2498, NULL, 8) = 2
read(17, "W", 16)                       = 1
recvfrom(24, "\3\0\0\0G\241\203q\1\0\0\0\0\253\320\26", 2264, MSG_DONTWAIT, NULL, NULL) = 16
ioctl(16, BINDER_WRITE_READ, 0xbef32920) = 0
clock_gettime(CLOCK_MONOTONIC, {8110, 299172636}) = 0
ioctl(16, BINDER_WRITE_READ, 0xbef322a0) = 0
ioctl(16, BINDER_WRITE_READ, 0xbef322a0) = 0
ioctl(16, BINDER_WRITE_READ, 0xbef32280) = 0
ioctl(16, BINDER_WRITE_READ, 0xbef32280) = 0
ioctl(16, BINDER_WRITE_READ, 0xbef322d0) = 0
clock_gettime(CLOCK_MONOTONIC, {8110, 307636809}) = 0
clock_gettime(CLOCK_MONOTONIC, {8110, 308091832}) = 0
getuid32()                              = 10017
```

Figure 3.3: Syscalls trace example.

3.3.4 Dynamic Watermark

For our dynamic watermark scheme, we decided to try a different approach, and attempted to use a trace of the system calls made by the application to the underlying OS as the dynamic watermark. This is possible because Android is based on Linux, therefore it provides applications with system calls in user mode to perform low level system operations. These include file operations (e.g., `read`, `write`), network operations (e.g., `socket`, `connect`), and process operations (e.g., `fork`, `exec`). These system calls (syscalls) made by the app provide important data on its runtime behavior [24].

The *syscall tracer* module of our service runs in the normal world and logs all the syscalls made by the app (we assume the integrity of the *syscalls tracer* module, refer to Section 3.4 for an explanation on how this assumption is enforced). It does so using the `ptrace` syscall. After a predetermined time or number of logged syscalls, the module sends the trace, i.e., the sequence of the syscalls, to the secure world via the *TZ Driver*. Figure 3.3 represents an example syscall trace.

In the secure world, the *application verifier* module compares the trace obtained from the *syscall tracer* with a trace that is stored in the VK. This trace contained in the VK is considered the default behavior of the application. The challenge here is that even if the app is genuine, there are no guarantees that the app produced the same trace as the one stored in the VK. This is due to the fact that the exact syscalls and their order depends on a lot of external factors, such as timing and interactions with other components, and there may be syscalls that appear in a trace and not in the other.

The solution we found was to use the Needleman-Wunsch algorithm [35], which is normally

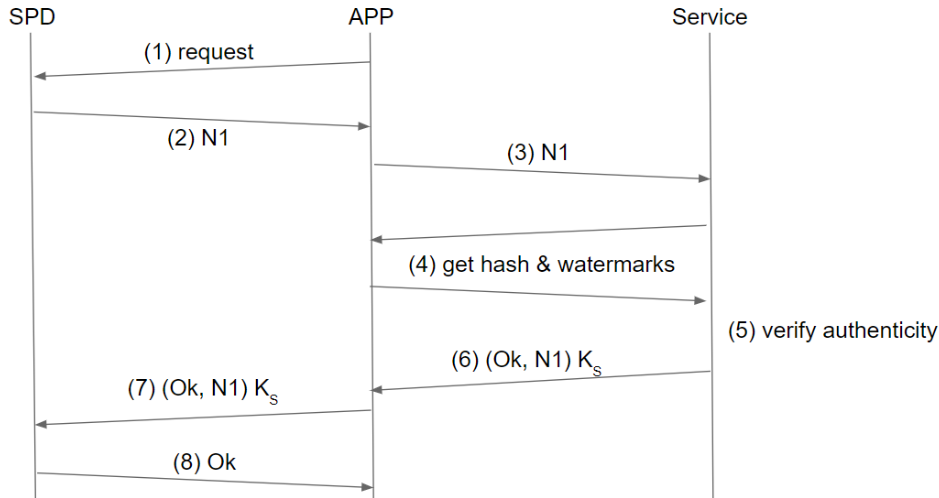

Figure 3.4: Authenticity verification scheme.

used to compare similarities in the amino-acid sequence of two proteins. The algorithm compares two sequences of letters, and assigns a final value of similarity between those sequences. This value depends on finding matches, mismatches and gaps between the two sequences. When a match is found some points are added, points are detracted when a gap is found, i.e., letters match but have some other letters in between that need to be discarded, and when a mismatch is found a larger number of points is usually detracted.

To use this algorithm with our dynamic watermark scheme, we have to transform the syscall traces in sequences of letters. In this project we only use the names of the system calls and do not take into account the parameters of each of those calls. Therefore, to create the sequence of letters, we need to go through the trace stored in the VK and for each new syscall we find, we assign it an unused letter. Then we go through the collected trace and create the correspoding sequence of letters. The final result is a string of letters, where each different letter represents a different syscall. Then we configure the Needleman-Wunsch algorithm with appropriate values found through experimentation (in our case we ended on match being added 4 points, gap being deducted 1 point, and mismatch being deducted 2 points), and compare the two traces. This gives us the similarity value for the two traces. We then just need to define a threshold $\texttt{Tresh}_{\neq}$ to decide if the apps that produced the two traces are the same.

3.3.5 Overview of the Authenticity Verification Process

The application is the one that initiates the authenticity verification process by asking the service to authenticate it. Figure 3.4 represents an example of the process for when an app wants to communicate with an external device (in this example the SPD developed in the PCAS project,

but it might be another device). In it we are assuming that the VK of the app was already verified by the service and its contents were already stored in the secure storage of the Trustzone. The steps in the detail are as follow:

1. The application sends a request to the SPD.
2. The SPD responds with a nonce (N1) – a number that is never reused.
3. The application informs the service that it wants its authenticity to be verified, by sending N1 to the *TZ Driver* module, which in turn passes it to the *interface* module in the secure world.
4. The service interacts with the app, obtaining the hash and/or the watermarks.
5. The service retrieves the contents of the VK for this application from the secure storage, and performs the necessary comparisons to validate the authenticity of the app.
6. Once the authenticity is verified, the service sends N1 signed with its private key to the application. This acts as a certificate that the application supplies to the SPD to prove that the service identified it as being genuine.
7. The application forwards the certificate to the SPD.
8. The SPD verifies the authenticity of the certificate with the service's public key, and verifies that the value of N1 corresponds to the initial value. If so the SPD will execute the request sent in step 1 and replies with OK to confirm it.

3.4 Normal World Integrity Verification

There are some modules of our service that run in the normal world, which means that we need to ensure the integrity of these modules to make sure they were not compromised by a hacker or malware. In Section 3.3.4 we assumed the integrity of the *syscall tracer* module, and in this section we are going to show how this assumption is enforced. This enforcement is implemented in the *system verifier* module, which in turn is divided in three sub-modules: *boot support*, *system verifier* and *tracer checker*. These sub-modules are in charge of respectively of the three aspects involved in the enforcement of the assumption: *trusted boot*, *system integrity verification*, and *tracer integrity verification*.

There are two moments when the normal world integrity verification is made:

1. during the boot of the device, where the *trusted boot* is executed.

2. whenever an authenticity verification process (Section 3.3) is requested by an application, where there is a *system integrity verification* followed by a *tracer integrity verification.*

If any of these verifications fail, the authenticity verification process terminates with failure.

3.4.1 Trusted Boot

When the device boots it involves executing a sequence of modules in a specific order, e.g., starting with the BIOS, then the bootloader, the OS kernel, and so on. In a *trusted boot* process, we designate the first module that is executed as the *static root of trust for measurement* (SRTM), and the trust that we have in this component influences the integrity of the whole boot [36]. In this process, each module, starting with the STRM, uses a collision-resistant hash function to get the hash of the next module and stores it in a safe place. Already available in many PCs, the best-known implementation of this process uses the Trusted Platform Module (TPM) discussed in Section 2.4. In this implementation, the hashes are stored in an array of Platform Configuration Registers (PCRs). More recently, the same idea has been implemented using Intel SGX technology, which provides trusted execution environments (enclaves) similarly to TrustZone [37].

In our system, we implemented this basic idea, and the details are covered in Section 4.2.2. In short, this process allows the verification of whether or not the normal world has been compromised.

3.4.2 System Integrity Verification

The *system verifier* module is used to verify if the normal world has been compromised or not. The module calculates the hashes of the various components and compares them with the hashes calculated and stored during the *trusted boot* process. Due to the collision-resistant properties of the hash functions used, we can be sure that if the hashes do not differ, then the normal world has not been compromised.

3.4.3 Tracer Integrity Verification

The *tracer verifier* module is in charge of verifying the integrity of the *syscall tracer* module that runs in the normal world. Since the *syscall tracer* is responsible for gathering the syscalls traces that are used to verify the authenticity of the application, then it is crucial to ensure that this module has not been compromised. This is done by having the hash of the module stored in the secure storage of the secure world. The value of this hash needs to come with the service,

as to not be compromised. The *tracer verifier* is then responsible for calculating the hash of the *syscall tracer* and comparing it with the stored one.

It is important to note here that this solution is not completely secure, as the *syscall tracer* might be modified after the integrity check verification. While the system integrity verification still provides assurance that software running in kernel mode is not compromised, the *syscall tracer* can still be modified through a vulnerability in the kernel or the *syscall tracer* itself. This proves to be a negative point about the whole dynamic watermark scheme chosen, and is taken into account when we compare the different techniques in Section 5.3.

Chapter 4

Service Implementation

In this chapter we go over the main points about the implementation of our service. For this project we chose to use an i.MX53 QSB board equipped with a Cortex-A8 single core 1 GHz processor, 1 GB DDR memory, and a 4GB MicroSD card. One of the reasons behind choosing this board is that in most TrustZone enabled commercial mobile devices, the secure world is locked in such a way that it is not possible to use it. While most of the implementation is independent of the device, there are a some aspects that depend on the hardware that we used.

4.1 Runtime Environment

The kernel used in the secure world is based on Genode [38], a framework of components to implement small OS kernels. In the secure world we use a small kernel based on a custom kernel (*base-hw*) provided by the Genome project for our board. This kernel provides the *tz_vmm* driver to support calls from the normal world to the secure world.

In the normal world, we installed a version of Android for the i.MX53 series from Adeneo/Freescale [39]. The Android kernel is patched to be executed in the normal world.

The SD card is used as the persistent memory of the board.

4.2 Service Components

In this section we go over the implementation of specific parts of our service that, as can be seen in Figure 3.1, run in both the normal world and the secure world.

In the secure world, we implemented the *app verifier*, *system verifier*, and *interface* modules based on a user level VMM app called *tz_vmm* that runs on top of the Genode kernel. The TrustZone configuration within Genode partitions the DDR RAM between the secure world and *tz_vmm* is able to request the normal world's RAM via an IOMEM session during its start-up

routine. The memory is mapped as uncached to the secure world's address space, thus the whole normal world memory can be accessed by the *system verifier* module running in the secure world. We also configured the Android file system partitions to be accessed by the secure world, so that the *app verifier* module can access the app's APK in the normal world to verify its integrity and authenticity.

4.2.1 Secure Storage

The secure storage is implemented by partitioning the SD card we use with the board. Since there is sensitive data that needs to be stored persistently (e.g., the contents of the VKs of the various applications) there needs to be a part of the SD card that is only accessible by the secure world. We use the Genode partition manager (*part_blk*) for this purpose. It provides a block session for each partition on the SD card, allowing each partition to be addressable as separate block sessions, enabling us to define who has access to each of them.

4.2.2 System Verifier – Trusted Boot

In this section we define how the *trusted boot* is implemented. We assume that when the device boots, the secure world is booted first, then it passes control to the normal world. Therefore, in our case, the secure world is the SRTM, so it computes a hash over the normal world. The module in charge of obtaining this hash is the *boot* module. The boot module boots the Android kernel and computes its hash. When the Android kernel starts to run, it does a measurement of the *init* program, in charge of initializing several elements of Android, and passes this hash to the secure world boot module. To do so it contacts the *TZ Driver* in a manner similar to calling a remote procedure

```
TZDriver_store_measurement(hash_t hash);
```

Then, *init* measures the program *app_process*, which when executed becomes the *zygote* process, i.e., the first instance of the Dalvik VM (the VM that executes all Android apps). Again, *init* calls the same function to pass the measurement to the boot module. The secure world does not have PCRs as it is not a TPM, but instead it stores the hashes in a vector that plays a similar role to the array of PCRs. Notice that the function does not take any input other than the hash; the hashes are simply stored by the boot module in the order it obtains them.

4.2.3 Application Verifier

The *application verifier* module is essentially composed of three functions, one for each of the techniques used for the authenticity verification process – hashes, static watermarks, dynamic

watermarks.

One thing all these functions have in common is having to access the secure storage to obtain the contents of the VK. Afterwards, in each of these functions, the first step is to collect the necessary data from the application, be it calculate its hash, retrieve the values of the bytes for the static watermark, or collect the syscalls trace for the dynamic watermark. Then a simple comparison is performed to validate the authenticity. In the case of the dynamic watermark, our implementation of the Needleman-Wunsch algorithm was based on the *seq-align* library [40]. The hash of the applications are calculated using SHA-512.

4.2.4 Syscall Tracer

We used *strace* to implement the *syscalls tracer* in the normal world. *strace* is a debugging tool for Linux that can be used to trace the syscalls made by a process. It relies on *ptrace* syscall and can be consider as an interface between the user space and *ptrace* syscall. The *syscalls tracer* module records only the name of each system call.

Chapter 5

Experimental Evaluation

In this chapter we evaluate the our solution in terms of detection and performance overhead. Specifically we wanted to answer two questions: (1) Is our solution able to detect non-genuine apps that pretend to impersonate genuine apps? (Section 5.1); (2) What is the performance overhead of running our service? (Section 5.2). We then compare the three different techniques (hashes, static watermark, dynamic watermark) used in our solution (Section 5.3).

5.1 Authenticity Verification

Each of the three techniques (hashes, static watermark, dynamic watermark) were tested individually. The experimental evaluation of each technique focused on deriving its detection performance. The tests we conducted were mainly separated in two phases.

In phase (1), we started by testing if each mechanism could correctly identify if an app was authentic using its VK. For this purpose we downloaded a set of apps from the Play Store[1] and manually constructed a VK for each of them with the correct values for each technique. We also tested if when using incorrect VKs (i.e., either the data in the VK is purposely wrong, or it is the VK of another app) the techniques were able to identify the apps as not authentic.

In phase (2), we used the techniques to compare apps with repackaged versions of themselves. In order to do so, we manually repackaged applications, and gathered real repackaged applications. Regarding the manually repackaged applications, we took 41 apps from the Play Store and followed the following steps in order to create repackaged versions of those apps [41]:

1. unpack the APK file using the Apktool [42];
2. convert the bytecode (DEX file) to Smali code (human-readable bytecode) using the same tool;

[1] https://play.google.com/store

3. add to the Smali code a simple malicious code snippet that deletes the user's contacts;

4. modify the file *manifest.xml* to give the app more permissions (in this case to read and write the contacts) and to trigger the code when the mobile device finishes booting;

5. repack the app with the Apktool;

6. sign the APK file and add a self-signed certificate.

As for the real repackaged applications we found repackaged apps and their corresponding genuine app. This apps were mostly *mod games*, i.e., games that were modified to have a different behavior from normal. We found 20 pairs of apps in these conditions, and their comprehensive list can be found in table 5.2.

5.1.1 Hashes

When using the hashes of the applications for the authenticity verification, in both phases (1) and (2), our service always managed to correctly identify an app as being genuine or not. This is due to the fact that the hashing algorithm used is collision resistant, therefore each different app has a different hash value. Furthermore, since the repackaging of apps modifies the byte-code, the resulting repackaged application will certainly have a different hash than the original one.

5.1.2 Static Watermaks

In phase (1), we used 100 applications downloaded from the Play Store, and generated the static watermark signature for each one of them using different sizes and random byte positions. The overhead of using different sizes for the watermark is calculated in Section 5.2.2. For each of these apps, we generated 10000 different signatures, and compared each app with each of its signatures, as well as each app with each signature from a different app. In all cases the service correctly identified when an app was genuine or fake.

In phase (2), we generated a signature watermark for each original app in both the pairs of manually repackaged apps and real repackaged apps. We then compared the repackaged version of each app with the signature from the original app. In each one of them the service also correctly identified that the apps were repackaged. This is due to the fact that the repackaging of apps introduces code snippets and creates a shift of the bytes past the position where the snippet is added, so the original byte values are not found in the expected position.

5.1.3 Dynamic Watermaks

While both hashes and static watermark proved to be 100% accurate due to their deterministic nature, the real important case to study are the dynamic watermarks.

In order to obtain the execution traces of the apps used for the dynamic watermark, we needed to execute the apps with the same sequence of inputs. To accomplish this, we ran and collected the traces of the apps using the Monkey application exerciser [43]. By using the same seed value each time, Monkey simulates the interaction with the app with a deterministic set of inputs (e.g., clicks, screen touches, key presses). The collection of the traces using Monkey were done in Google's Android Emulator[2], emulating an ARM CPU.

In phase (1), we collected 31 traces for 41 apps that were taken from the Play Store. We chose the trace that had the most similarity with the remaining ones and defined it as the signature for the dynamic watermark. We then compared each of the other 30 traces of each app with the signature to see if they were the same. Additionally, we compared the 30 traces of each app with each signature from each other app to see if they were identified as not being the same. We considered different threshold values for the Needleman-Wunsch algorithm to determine if the traces were from the same app or not. For each threshold value we measured six common performance metrics. We consider:

- a true positive (TP) is when a trace is correctly identified as being from the same app;
- a true negative (TN) is when a trace is correctly identified as not being from the same app;
- a false positive (FP) is when a trace is incorrectly identified as being from the same app;
- a false negative (FN) is when a trace is incorrectly identified as not being from the same app.

The metrics used are:

$Accuracy = (TP + TN)/(TP + TN + FP + FN)$

$False\ Positive\ Rate\ (FRP) = FP/(FP + TN)$

$False\ Negative\ Rate\ (FNR) = FN/(FN + TP)$

$Recall = True\ Positive\ Rate\ (TPR) = TP/(TP + FN)$

$Precision = TP/(TP + FP)$

$Fmeasure = 2 \times Recall \times Precision/(Recall + Precision)$

These results are found in Table 5.1. We can see that for each $\texttt{Tresh}_{\neq}$ the accuracy does not vary much, but as the False Positive Rate decreases, the False Negative Rate increases. The

[2] https://developer.android.com/studio/run/emulator.html

$\texttt{Tresh}_{\neq}$	**Accuracy**	**FPR**	**FNR**	**Recall**	**Precision**	**Fmeasure**
1750	0.977	0.019	0.051	0.949	0.894	0.921
1770	0.975	0.017	0.061	0.939	0.902	0.920
1790	0.977	0.015	0.071	0.929	0.910	0.919
1810	0.978	0.012	0.083	0.918	0.928	0.913
1830	0.980	0.009	0.092	0.908	0.947	0.927
1850	0.981	0.003	0.112	0.888	0.978	0.930

Table 5.1: Evaluation of dynamic watermarking.

same is true with Recall and Precision. The best $\texttt{Tresh}_{\neq}$ value is chosen depending on the metric considered, e.g., if we choose Fmeasure to be the most important metric then $\texttt{Tresh}_{\neq} = 1850$ is the best Threshold value.

In phase (2), we first used Monkey to collect 30 traces for each of the manually repackaged apps. We then compared each trace from each app with the trace corresponding to the original app. Unfortunately the detection rate for these set of apps was 0, i.e., the technique failed to detect the repackaged apps as non-genuine. The reason for this comes from the fact that the malicious code snippet that we injected in the app runs only after a reboot, therefore it was never executed during the collecting of the traces.

We repeated this experiment, but this time with the real repackaged apps. The results of this experiment are found in Table 5.2. These results are inconsistent, with most of the values near 0. This can be attributed to the fact that most of these repackaged applications have a behavior that is very similar to the original one.

5.2 Performance Overhead

We evaluated the performance overhead incurred by each technique to study which one is the best in this aspect, as well as the performance overhead incurred in the normal world integrity verification process.

5.2.1 Hashes

We evaluated the time for the *measurement checker* module to calculate the hash of the app using SHA-512 and compare it against the hash value present in the VK. We used different sized APK files for this experiment, since the time it takes to calculate the hash is mostly dependent on the size of the APK itself. The results are found in Table 5.3, and show that the time (t) grows linearly with the size (s) of the APK file. The trend observed is $t = 0.9388 \times s - 0.0562$, i.e. on average 0.93 seconds are necessary for checking the integrity of an app of size 1 MB by measurements module.

APKs \ `Tresh`$_{\neq}$	1700	1750	1800	1850	1900
Angry Birds 7.5	0	0	0	0	0
Bomb Squad Pro 1.4.121	0.95	1	1	1	1
CCLeaner 1.19.76	0	0	0	0	0
Clash of Clans 9.24.15	0	0	0.05	0.05	0.05
Clash Royale 1.9.2	0	0	0	0	0
Crossy Road 2.4.4	0	0	0	0	0
FIFA Mobile Soccer 6.1.1	0.1	0.1	0.15	0.2	0.2
Flags Quiz 2.4	1	1	1	1	1
Flick Kick FootballLegends 1.9.85	0	0	0	0	0
Last Day on Earth 1.5.6	0	0	0	0	0
Last Hope TD 3.31	0.25	0.25	0.25	0.25	0.25
Magikarp Jump 1.1.0	0	0	0.1	0.1	0.1
Mo n Ki World Dash 1.6	0.15	0.15	0.2	0.2	0.35
Once Upon a Tower 3	1	1	1	1	1
Realm Defense 1.8.4	0.05	0.05	0.1	0.1	0.1
Sniper 3D Assassin 2.0.2	0	0	0	0	0
Super Mario Run 2.1.1	0	0	0	0	0.1
Zombie Castaway 2.8.1	0	0	0	0	0.05
8 Ball Pool 3.10.3	0	0	0	0	0.1
8 Ball Pool 3.10.1	0	0	0	0	0

Table 5.2: Detection rate for real repackaged apps.

5.2.2 Static Watermarks

The overhead of our static watermarking technique is measured by the time it takes the *static watermarker* module running in the secure world to read the position of the bytes listed in the *VK* and comparing the values of the bytes in those positions in the target app bytecode with the expected values in the *VK*. We repeated this experiment with different number of bytes (n), since the time (t) is dependent on them. The results are found in Table 5.4, and the trend observed is $t = 3.0056 \times n - 90.24$.

Size (MBytes)	Time (ms)
3.3	3,090
4.9	4,580
13.3	12,430
17.5	16,350
18.6	17,370
25.6	23,970
28.3	26,510
37.7	35,160
59.0	55,540
91.5	85,790

Table 5.3: Time to create hashes.

No. Bytes	Time (ms)
4	66
8	68
16	72
32	81
64	97
128	130
256	196
512	1,556
1024	3,059
2048	6,071

Table 5.4: Time to do static watermarking.

No. Letters	Conversion (ms)	Comparison (ms)	Total (ms)
200	96.97	107.43	204.4
400	114.68	108.72	223.4
600	132.53	188.42	321.0
800	159.73	312.63	472.4
1000	168.89	415.21	584.1

Table 5.5: Time to do trace conversion and comparison.

5.2.3 Dynamic Watermarks

Since the time to extract a trace is configurable, i.e., we can choose to limit the number or type of events used by Monkey, we do not measure the time of the whole process. We measure the time it takes for the *syscall tracer* to send the trace data to the *interface module* in the secure world. This encompasses the time it takes for the context switching between the two worlds, the copying of the data to the shared buffer, and sending it to the secure world. We measured this time with different sized traces, and the average throughput to perform these operations is 17.51 MB/s.

Additionally, we measured the time it takes for a syscall trace to be converted into a sequence of an alphabetical letter, as well as the time it takes to compare two traces using the Needleman-Wunsch algorithm. These tests were also repeated for different sized traces, and results can be found in Table 5.6. The time (t) it takes to completed both the conversion and comparison of the traces depends on the number of letters in the trace (l). The trend we observe is $t = 0.5042 \times l + 58.534$.

5.2.4 Integrity Verification Process

The *system verifier* checks the integrity of the normal world by calculating hashes of the Android kernel, init, and app_process using SHA-512, and comparing them against their known-good

File name	Size (Kbytes)	Time (ms)
`app_process`	5.7	7.48
`init`	90.1	48.71
syscalls tracer module	1126	829.55
Android kernel	8324	932.66
Total	*9545.8*	*1818.4*

Table 5.6: Time to do integrity verification.

Technique	Accur.	FPR	FNR	Recall	Prec.	Fmeas.
Hashes	1	0	0	1	1	1
Static watermarks	1	0	0	1	1	1
Dynamic watermarks (Table 5.2)	–	–	1	0	–	–
Dynamic watermarks (Table 5.1)	.980	.009	.092	.908	.947	.927

Table 5.7: Comparison of the authenticity verification process of the three techniques.

values. The *tracer checker* also does the same operations for the *syscall tracer* module running in the normal world to verify its integrity. Therefore, we measured the time to perform those operations (Table 5.5). The results are the average of 1000 repetitions. The table shows both the size and the time to check the integrity of the modules. The total time required to check the integrity of the normal world is around 2 seconds in our board.

5.3 Tradeoffs on Detection Techniques

To summarize, Table 5.7 contains the values of the metrics for the authenticity verification for the three techniques. For dynamic watermarks we consider the values of Tables 5.2 (the worst case; some metrics are not filled as there is no value for TN) and 5.1 (for $\texttt{Tresh}_{\neq} = 1830$).

The general comparison between the three techniques is found in Table 5.8. The column *Protection* refers to protection from the normal world. We can see that, since both hashes and static watermarks run only in the secure world, they are protected from the normal world. Dynamic watermarks on the other side run part of the code in the normal world (*syscall tracer*),

Technique	Protection	Detection	Delay
Hashes	best	best (collision resistance)	worst
Static watermarks	best	high	best
Dynamic watermarks	high	worst	average

Table 5.8: Summary comparison between the three techniques.

therefore the degree of protection is obviously lower, although it is mitigated due to the integrity verification of the *syscall tracer* module. The column *Detection* refers to the authenticity verification detection. From what we could see from Table 5.7, both hashes and static watermarks provided the best detection rate, although it can be said that hashes are theoretically better since they leverage the collision resistance property of hash functions. Static watermarks depend on the modifications being made to the app to also modify the bytes checked for the watermark, although with a high enough number of bytes used it should not be a problem. Dynamic watermarks, while being able to detect when two apps are not the same with some confidence level (Table 5.1), fail to detect when an app has been repackaged (Table 5.2). Finally, the column *Delay* refers to the performance overhead. Hashes are clearly the most time consuming, due to the time it takes to obtain a cryptographic hash. Dynamic watermarks also take some time, but not as much as hashes. Static watermarks are the fastest when compared to the other.

Chapter 6

Conclusions

This chapter describes the main conclusions obtained from our research and the development of the project.

6.1 Summary and Contributions

Ensuring the authenticity and integrity of software is a challenging task, especially when there are no guarantees that the system on which the software is executed is not malicious. Our goal was to develop a service that leveraged the Trustzone hardware extension available to ARM processors that could verify the authenticity of an application running in the normal world. While most of the service is executed in the secure world, some components still run in the normal world, but their integrity is verified by a module of the service running in the secure world. This document carefully studied various methods used to tamperproof software and verify its authenticity and integrity. In order to reduce single points of failure, our service uses a combination of cryptographic hashes and watermarking techniques. We implemented the service on a i.MX53 QSB and performed an experimental evaluation of the different techniques used.

This work was partially published as:
TruApp: A TrustZone-based Authenticity Detection Service for Mobile Apps
Sileshi Yalew, Pedro Mendonça, Gerald Maguire, Seif Haridi and Miguel Correia
IEEE 13th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)
October 2017

6.2 Future Work

In this project we provided the general idea for leveraging the ARM Trustzone extension in order to verify the authenticity of applications running in a smartphone. What can still be further developed and researched upon are different techniques from the ones we used (cryptographic hashes, static watermarking, dynamic watermarking).

Bibliography

[1] C. S. Collberg and C. Thomborson. Watermarking, tamper-proofing, and obfuscation-tools for software protection. *IEEE Transactions on software engineering*, 28(8):735–746, 2002.

[2] U. Piazzalunga, P. Salvaneschi, F. Balducci, P. Jacomuzzi, and C. Moroncelli. Security strength measurement for dongle-protected software. *IEEE Security & Privacy*, 5(6):32–40, 2007.

[3] F. Kvant and M. Kellner. A development environment for arm trustzone with globalplatform support. Master's thesis, Department of Electrical and Information Technology, Faculty of Engineering, LTH, Lund University.

[4] T. Alves and D. Felton. Trustzone: Integrated hardware and software security. *ARM white paper*, 3(4):18–24, 2004.

[5] N. Santos, H. Raj, S. Saroiu, and A. Wolman. Using arm trustzone to build a trusted language runtime for mobile applications. In *ACM SIGARCH Computer Architecture News*, volume 42, pages 67–80. ACM, 2014.

[6] J. Winter. Experimenting with arm trustzone–or: How I met friendly piece of trusted hardware. In *2012 IEEE 11th International Conference on Trust, Security and Privacy in Computing and Communications*, pages 1161–1166. IEEE, 2012.

[7] J. Korhonen. Piracy prevention methods in software business. B.S. Thesis, University of Oulu, Department of Information, 2015.

[8] G. Naumovich and N. Memon. Preventing piracy, reverse engineering, and tampering. *Computer*, 36(7):64–71, 2003.

[9] H. Chang and M. J. Atallah. Protecting software code by guards. In *ACM Workshop on Digital Rights Management*, pages 160–175. Springer, 2001.

[10] E. J. Chikofsky and J. H. Cross. Reverse engineering and design recovery: A taxonomy. *IEEE software*, 7(1):13–17, 1990.

[11] Nist's policy on hash functions. Available at `http://csrc.nist.gov/groups/ST/hash/policy_Sept2012.html`, 2016. [Online; accessed 27-December-2016].

[12] Y. Chen, R. Venkatesan, M. Cary, R. Pang, S. Sinha, and M. H. Jakubowski. Oblivious hashing: A stealthy software integrity verification primitive. In *International Workshop on Information Hiding*, pages 400–414. Springer, 2002.

[13] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[14] M. Bellare and P. Rogaway. The exact security of digital signatures-how to sign with rsa and rabin. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 399–416. Springer, 1996.

[15] S.Oaks. *Java Security*. O'Reilly, 2 edition, 2001.

[16] Y. Zhauniarovich, O. Gadyatskaya, and B. Crispo. Demo: Enabling trusted stores for android. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 1345–1348. ACM, 2013.

[17] B. Horne, L. Matheson, C. Sheehan, and R. E. Tarjan. Dynamic self-checking techniques for improved tamper resistance. In *ACM Workshop on Digital Rights Management*, pages 141–159. Springer, 2001.

[18] M. Ceccato and P. Tonella. Codebender: Remote software protection using orthogonal replacement. *IEEE software*, 28(2):28–34, 2011.

[19] J. T. Giffin, M. Christodorescu, and L. Kruger. Strengthening software self-checksumming via self-modifying code. In *21st Annual Computer Security Applications Conference (ACSAC'05)*, pages 10–pp. IEEE, 2005.

[20] G. Wurster, P. C. van Oorschot, and A. Somayaji. A generic attack on checksumming-based software tamper resistance. In *2005 IEEE Symposium on Security and Privacy (S&P'05)*, pages 127–138. IEEE, 2005.

[21] S. A. Bailey, D. Felton, V. Galindo, F. Hauswirth, J. Hirvimies, M. Fokle, F. Morenius, C. Colas, J.-P. Galvan, G. Bernabeu, et al. The trusted execution environment: Delivering enhanced security at a lower cost to the mobile market. *GlobalPlatform White Paper*, 2011.

[22] J.-E. Ekberg, K. Kostiainen, and N. Asokan. Trusted execution environments on mobile devices. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 1497–1498. ACM, 2013.

[23] S. Ravi, A. Raghunathan, and S. Chakradhar. Tamper resistance mechanisms for secure embedded systems. In *VLSI Design, 2004. Proceedings. 17th International Conference on*, pages 605–611. IEEE, 2004.

[24] S. D. Yalew, G. McGuire, S. Haridi, and M. Correia. T2Droid: A TrustZone-based dynamic analyser for Android applications. In *Proceedings of the 16th IEEE International Conference on Trust, Security and Privacy in Computing and Communications*, pages 25–36, Aug. 2017.

[25] S. D. Yalew, G. Q. Maguire Jr., S. Haridi, and M. Correia. DroidPosture: A trusted posture assessment service for mobile devices. In *Proceedings of the 13th IEEE International Conference on Wireless and Mobile Computing, Networking and Communications*, oct 2017.

[26] V. Haldar, D. Chandra, and M. Franz. Semantic remote attestation: a virtual machine directed approach to trusted computing. In *USENIX Virtual Machine Research and Technology Symposium*, volume 2004, 2004.

[27] R. Sailer, X. Zhang, T. Jaeger, and L. Van Doorn. Design and implementation of a tcg-based integrity measurement architecture. In *USENIX Security Symposium*, volume 13, pages 223–238, 2004.

[28] A.-R. Sadeghi and C. Stüble. Property-based attestation for computing platforms: caring about properties, not mechanisms. In *Proceedings of the 2004 workshop on New security paradigms*, pages 67–77. ACM, 2004.

[29] M. Nauman, S. Khan, X. Zhang, and J.-P. Seifert. Beyond kernel-level integrity measurement: enabling remote attestation for the android platform. In *International Conference on Trust and Trustworthy Computing*, pages 1–15. Springer, 2010.

[30] D. M. Nicol. Modeling and simulation in security evaluation. *IEEE security & privacy*, 3 (5):71–74, 2005.

[31] C. S. Collberg, G. Myles, and A. Huntwork. Sandmark-a tool for software protection research. *IEEE Security and Privacy*, 1(4):40–49, 2003.

[32] ENISA. Algorithms, key size and parameters report – 2014. Nov. 2014.

[33] A. J. Menezes, P. C. V. Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography.* CRC Press, 1997.

[34] R. Cramer and V. Shoup. Design and analysis of practical public-key encryption schemes secure against adaptive chosen ciphertext attack. *SIAM Journal on Computing*, 33(1): 167–226, 2003.

[35] S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3): 443–453, 1970.

[36] B. Parno, J. M. McCune, and A. Perrig. *Bootstrapping Trust in Modern Computers*. Springer, 2011.

[37] I. Anati, S. Gueron, S. Johnson, and V. Scarlata. Innovative technology for CPU based attestation and sealing. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, 2013.

[38] G. Labs. ARM TrustZone, an exploration of ARM TrustZone technology. `http://genode.org/news/an-exploration-of-arm-trustzonetechnology`, 2014.

[39] Witekio. NXP i.MX 53 reference BSP. `http://witekio.com/cpu/i-mx-53/`.

[40] I. Turner. seq-align: Smith-Waterman & Needleman-Wunsch alignment in C. `https://github.com/noporpoise/seq-align`.

[41] W. Du. SEEDlabs: Android repackaging attack lab. `http://www.cis.syr.edu/~{}wedu/seed/Labs_Android5.1/Android_Repackaging/`.

[42] C. Tumbleson and R. Wisniewski. Apktool. `https://ibotpeaches.github.io/Apktool/`.

[43] Monkey. `https://developer.android.com/studio/test/monkey.html`.