Software Authenticity Protection in Smartphones using ARM Trustzone

Pedro Miguel dos Santos Mendonça
Instituto Superior Técnico, Universidade de Lisboa

***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[1] 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**

I. 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[2]. 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.

A. 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 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]–[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.

II. Related Work

In this section we will go over the main topics we researched for the implementation of this project.

A. Static Watermarking

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

In order to accomplish this there are a few properties that the watermark needs to ensure [1], [7]:

- 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.

It is necessary to understand the different types of watermarking attacks that are possible in order to evaluate the

[1]A version of this work was published as: Sileshi D. Yalew, Pedro Mendonça, Gerald Maguire Jr., Seif Haridi, Miguel Correia. TruApp: A TrustZone-based Authenticity Detection Service for Mobile Apps. In Proceedings of the 13th IEEE International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob), Oct. 2017.

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

resilience of a watermarking technique. Such attacks include [8]:

- *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 [7]. 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.

B. 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 [7].

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.

C. 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] 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 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 [9]. The manufacturers need to take precautionary steps while developing their Trustzone-based systems.

III. Authenticity Protection Service

In this section we go over the architecture we chose for our service. Our solution uses a combination of software integrity verification using hashes and watermarking techniques, leveraging the security mechanisms available in the ARM Trustzone.

A. *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

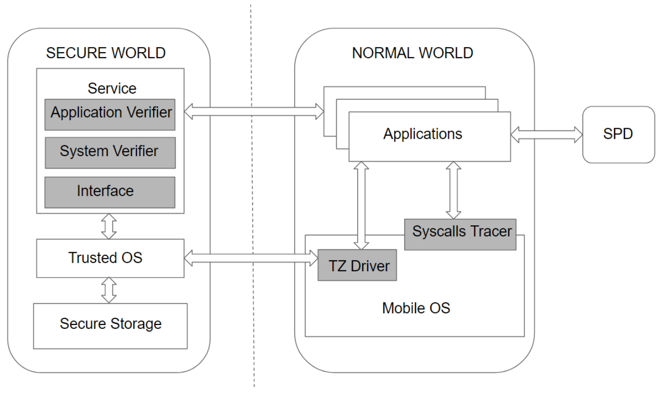


Fig. 1: Architecture of a mobile device running our service.

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 key pair, with a secure key size (e.g., 3072 bits for RSA [10]). 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 [11], as well as a secure symmetric encryption system with cipher block chaining (CBC) mode [10].

B. *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 III-C1.

Our service runs mostly in the secure world, but some components need to run in the normal world. Figure 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.

- *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.

C. 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 1).

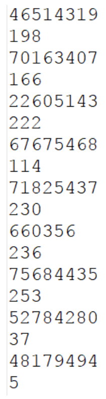
1) Verification Key: The developers of the applications need to provide a *Verification Key* alongside the application that contains the necessary information for our service to correctly verify the authenticity of said application. Since our verification process has three techniques which are used (hashes, static watermarks and dynamic watermarks), we need information on how to validate each one of them. The exact contents of the VK are discussed in each of the following sections.

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 [12]. 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.



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

Fig. 2: Static watermark validation data in the VK.

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.

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 comparing 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) 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 2 represents an hypothetical possibility for such information. The first line corresponds to the byte position and the next line

```
clock_gettime(CLOCK_MONOTONIC, {8102, 682499704}) = 0
epoll_pwait(19, {{EPOLLIN, {u32=23, u64=23}}}, 16, -1, NULL,
recvfrom(23, "\1\0\0\0\0\0\0\0\315\0\0\0\0\0\0\0\247\244Q`
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}}
sendto(24, "\1\0\0\0\310`\214o\1\0\0\0\320f\263\247\0\247\24
clock_gettime(CLOCK_MONOTONIC, {8110, 291055838}) = 0
write(18, "W", 1)                           = 1
recvfrom(23, 0xb512e410, 2264, 64, 0, 0) = -1 EAGAIN (Try ag
clock_gettime(CLOCK_MONOTONIC, {8110, 293747138}) = 0
epoll_pwait(19, {{EPOLLIN, {u32=24, u64=24}}, {EPOLLIN, {u32
read(17, "W", 16)                           = 1
recvfrom(24, "\3\0\0\0G\241\203q\1\0\0\0\0\253\320\26", 2264
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
```

Fig. 3: Syscalls trace example.

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 V-B2 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.

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 [13].

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 III-D 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 represents an example syscall trace.

In the secure world, the *application verifier* module compares the trace obtained from the *syscall tracer* with a trace

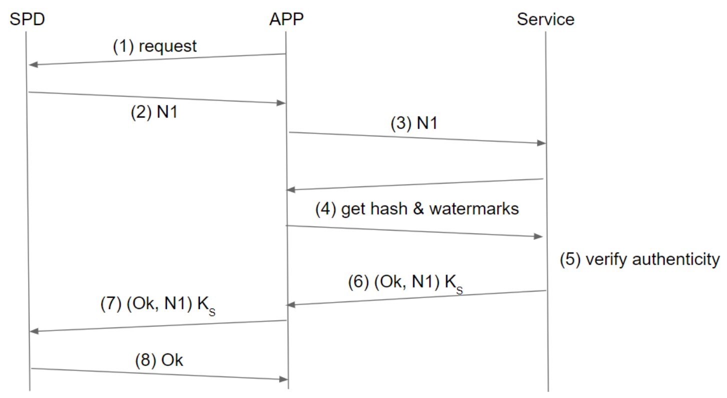


Fig. 4: Authenticity verification scheme.

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 [14], which is normally 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. To do so we follow some deterministic criteria to transform each system call found in the trace into a letter. 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 $\mathtt{Tresh}_{\neq}$ to decide if the apps that produced the two traces are the same.

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 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.

D. 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. 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 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.

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 [15]. 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.

In our system, we implemented this basic idea, and the details are covered in Section IV-B2.

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) 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 V-C.

IV. 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.

A. Runtime Environment

The kernel used in the secure world is based on Genode [16], 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 [17]. The Android kernel is patched to be executed in the normal world.

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

B. Service Components

In this section we go over the implementation of specific parts of our service that, as can be seen in Figure 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 worlds 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.

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.

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. It stores the hashes in a vector.

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 [18]

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.

V. 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 V-A); (2) What is the performance overhead of running our service? (Section V-B). We then compare the three different techniques (hashes, static watermark, dynamic watermark) used in our solution (Section V-C).

A. 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[3] 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 [19]:

1) unpack the APK file using the Apktool [20];
2) convert the bytecode (DEX file) to Smali code (human-readable bytecode) using the same tool;
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;

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

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 II.

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.

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 V-B2. 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.

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 [21]. 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[4], 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:

$Accuracy = (TP + TN)/(TP + TN + FP + FN)$
$False\ Positive\ Rate\ (FRP) = FP/(FP + TN)$
$False\ Negative\ Rate\ (FNR) = FN/(FN + TP)$
$Recall = True\ Positive\ Rate\ (TPR) = TP/(TP + FN)$
$Precision = TP/(TP + FP)$
$Fmeasure = 2 \times Recall \times Precision/(Recall + Precision)$

TABLE I: Evaluation of dynamic watermarking.

$\mathtt{Tresh}_{\neq}$	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

These results are found in Table I. We can see that for each $\mathtt{Tresh}_{\neq}$ the accuracy does not vary much, but as the False Positive Rate decreases, the False Negative Rate increases. The same is true with Recall and Precision. The best $\mathtt{Tresh}_{\neq}$ value is chosen depending on the metric considered, e.g., if we choose Fmeasure to be the most important metric then $\mathtt{Tresh}_{\neq}$ = 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 II. 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.

B. Performance Overhead

We evaluated the performance overhead incurred by each technique to study which one is the best in this aspect, as

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

TABLE II: Detection rate for real repackaged apps.

APKs \ $\texttt{Tresh}_{\neq}$	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 III: Time to create hashes.

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

well as the performance overhead incurred in the normal world integrity verification process.

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 III, 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.

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

TABLE IV: Time to do static watermarking.

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 V: Time to do trace conversion and comparison.

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 VI: Time to do integrity verification.

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*

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 IV, and the trend observed is $t = 3.0056 \times n - 90.24$.

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 VI. 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 $t = 0.5042 \times l + 58.534$.

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 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 V). 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.

C. Tradeoffs on Detection Techniques

To summarize, Table VII contains the values of the metrics for the authenticity verification for the three techniques. For dynamic watermarks we consider the values of Tables II (the

TABLE VII: Comparison of the authenticity verification process of the three techniques.

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 II)	–	–	1	0	–	–
Dynamic watermarks (Table I)	.980	.009	.092	.908	.947	.927

TABLE VIII: Summary comparison between the three techniques.

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

worst case; some metrics are not filled as there is no value for TN) and I (for $\texttt{Tresh}_{\neq}$ = 1830).

The general comparison between the three techniques is found in Table VIII. 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*), 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 VII, 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 I), fail to detect when an app has been repackaged (Table II). 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.

VI. Conclusion

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.

References

[1] C. S. Collberg and C. Thomborson, "Watermarking, tamper-proofing, and obfuscation-tools for software protection," *IEEE Transactions on software engineering*, vol. 28, no. 8, pp. 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*, vol. 5, no. 6, pp. 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*, vol. 3, no. 4, pp. 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*, vol. 42, no. 1. ACM, 2014, pp. 67–80.

[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*. IEEE, 2012, pp. 1161–1166.

[7] G. Naumovich and N. Memon, "Preventing piracy, reverse engineering, and tampering," *Computer*, vol. 36, no. 7, pp. 64–71, 2003.

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

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

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

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

[12] R. Cramer and V. Shoup, "Design and analysis of practical public-key encryption schemes secure against adaptive chosen ciphertext attack," *SIAM Journal on Computing*, vol. 33, no. 1, pp. 167–226, 2003.

[13] 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*, Aug. 2017, pp. 25–36.

[14] 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*, vol. 48, no. 3, pp. 443–453, 1970.

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

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

[17] Witekio, "NXP i.MX 53 reference BSP," http://witekio.com/cpu/i-mx-53/.

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

[19] W. Du, "SEEDlabs: Android repackaging attack lab," http://www.cis.syr.edu/~wedu/seed/Labs_Android5.1/Android_Repackaging/.

[20] C. Tumbleson and R. Wisniewski, "Apktool," https://ibotpeaches.github.io/Apktool/.

[21] Monkey, https://developer.android.com/studio/test/monkey.html.