REMOTE RUNTIME DETECTION OF TAMPERING AND OF DYNAMIC ANALYSIS ATTEMPTS FOR ANDROID APPS

UNIVERSITY OF TRENTO - Italy

By

LEONIDAS VASILEIADIS

A thesis submitted in partial fulfillment of
the requirements for the degree of

MASTER OF SCIENCE

UNIVERSITY OF TRENTO
Department of Information Engineering and Computer Science - DISI

JULY 2019

Supervisor: Mariano Ceccato
University of Trento

Co-Supervisor: Andreas Peter
University of Twente

© Copyright by LEONIDAS VASILEIADIS, 2019
All Rights Reserved

ACKNOWLEDGMENT

To my parents and my sister for their undying support, who throughout my studies they were motivating me and were always keen on learning more on what I am doing although it is likely they do not grasp it entirely.

I am grateful to all the rest of my family members who supported me financially and psychologically during the first steps of my Master studies and never missed a chance to cheer me up.

To my mentor and supervisor Dr. Mariano Ceccato, I will be forever grateful, for introducing me to the intriguing world of Android security and for his help and guidance in every step.

A very special gratitude to the EIT Digital Master School, for this wonderful opportunity to have a master degree studying in two different top universities and for fully funding and assisting me in every aspect required in these past two years.

I would also like to thank the company *2Aspire* for hosting me as an intern and actively providing me all the tools and assistance I required.

REMOTE RUNTIME DETECTION OF TAMPERING AND OF DYNAMIC ANALYSIS ATTEMPTS FOR ANDROID APPS

Abstract

by Leonidas Vasileiadis, MSc
University of Trento
July 2019

Supervisor: Mariano Ceccato, Andreas Peter

Given the fact that Android is the operating system in the majority of the portable devices, we realize why Android applications remain an attractive target for tampering and malicious reverse engineering. Many companies attempted to introduce anti-tampering and anti-debugging features and the Android itself is more secure than ever, but apparently it all ends up in a game of cat and mouse, with the malicious reverse engineers always one step ahead of the defenders.

Unprotected apps expose themselves to an attacker that can take advantage and compromise the integrity of the application and its resources along with the privacy of the user. That would lead the company behind the app to revenue losses and the users into falling victims of malicious intents. Protection against tampering and malicious reverse engineering becomes essential in defending app behavior integrity and user privacy.

Our goal in this thesis is to introduce a comprehensive and efficient approach into evaluating the state of an application itself along with the environment it is running on, thus being able to identify attempts of tampering or malicious reverse engineering made to the application. Our solutions are capable of safeguarding an app against tampering and dynamic binary instrumentation while at the same time are light-weight to resource consumption and resistant against known bypassing techniques for detection. The proposed approach can be embedded effortlessly into an android app by the developers.

Contents

List of Tables

List of Figures

Chapter One

Summary

1.1 Context and Motivations

The extensive diffusion of Android systems today and its open source nature has attracted many users with intents of tampering and malicious reverse engineering [25], and although Android has shown many security improvements there is still room for more. Even recently we witnessed banking protected applications being reversed engineered and proven weak upon analysis of the state of the Android security hardening [16]. It sure seems intriguing why defending against application tampering appears to be a difficult task to accomplish and further what are the protections employed by big corporations in order to shield their interests. Which techniques are more effective against tampering an application and is it possible to have one solution to cover all apps?

1.2 Problem Definition

Developers of Android apps frequently face deadlines and pressures for shorter times to market, therefore the security of an app is not always their primary concern. That is the reason Google has stepped up and performs basic checks of an application that is submitted to Play Store [35]. These are precautions employed by Google at a basic level of security, mainly protecting, developers from introducing unwillingly a path to easy tampering of the app and inexperienced users from harming their devices and their privacy. Yet, it still remains a significant part of an app that is never tested on how resistant it is against tampering or malicious reverse engineering. This is the entry point for attackers, in successfully reverse engineering Android applications with the purpose of injecting malware or adware in it.

Let us assume a famous music application that offers a freemium model, where users may have access to selected music after paying a fee. The security concern we face in this case is twofold. On the one side the company is loosing revenue if the app is reversed and offered to users without paying any fee to the company and on the other hand as a side effect, users who on their attempt to have access to the resources of the aforementioned app, without paying for it, download and install it from untrusted third party sources.

1.3 Proposed Approach & Contribution

Our approach includes the architecture used by the company *2Aspire*, under of which, our internship took place. This architecture as can also be seen in figure 1.1, includes the existence of two components shipped with the Android app and a server which collects, evaluates and responds accordingly depending on the request. The *check component*, is the component to which we contributed and does the testing of the integrity and the detection of malicious reverse engineering. The other component, named *split*, extracts a part of the application that the developer decides and transfers it to the server, where upon a successful check from the *check component*, the server allows the application to have access to that part. Lastly the *server* itself is responsible to evaluate the results from the *check component* and upon verifying that the device is safe, allow access to the removed from the *split component* part of the app. The main principle behind this architecture is to force the application to interact

with the server in order to be able to have a fully working code. This way a response from the *check component* is expected to arrive to the server in order to be evaluated. After the evaluation, the server will mark the device as "trusted" if the check was successful otherwise nothing will happen. When there is the *Split Code Request* the server will verify that the device requesting the code belongs to the "trusted" ones and if it is, it will reply with the code. The advantage of this communication pattern is that an attempt to completely bypass the check will result in the server refusing to serve the app as it will not be authorized.

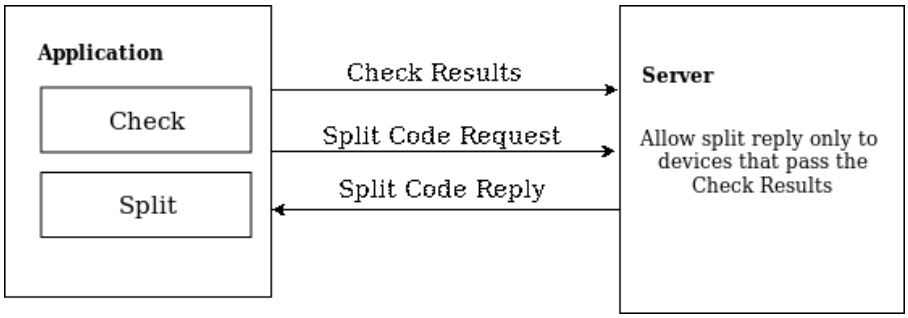

Figure 1.1 2Aspire Architecture

Our contribution is towards identifying tampering and malicious reverse engineering of an app and it composes the *check component* in the aforementioned architecture. It is of critical importance to have a high reliability and precision check in the place of the *check component* as, a failure to identify a tampering of an application would result in accessing a fully functional application, since the server would respond with the part of the code missing.

We implemented the initialization of the *check component* to be independent of a prior communication to the server, so the communication for the check is only one-way. In order to protect from replay attacks due to this pattern, we have added a protection mechanism which we named *"magic numbers"*, which are seemingly random values generated by the tool that the server can identify and verify their uniqueness. Further for detecting any tampering on the application itself we do a signature checking of the apk, employing a cutting-edge method highly efficient against known attacks.

We employ more than 20 different checks in regard to detecting malicious reverse engineering attempts, which either directly or indirectly identify a threat environment to the app. We implement, detection of usage of improper keys in the build, finding paths and files with unusual permissions, identifying suspicious Android properties, evaluating the state of SELinux, directly detecting Root access or root managers, checking for signs of emulators and the detection of dynamic binary instrumentation frameworks. The majority of the implementations are done in *C* using the *Java Native Interface*.

The techniques we introduce in our proposal are innovative, accurate and resource efficient and as such, are bypassing known tools built for Android application tampering. Ensuring this way that no automated tool or script kiddie will manage to bypass all the security checks.

1.4 Outcomes

Overall we implemented more than 20 security checks covering a wide range of security aspects in Android. As mentioned, we were seeking for a lightweight and efficient solution which would not clutter a device and would not slow down the performance of an application. Our performance evaluation showed excellent results with mid and high range devices to be able to run even two security checks for every second and the CPU consumption to increase around 1%. In regard to memory consumption we discovered that approximately 10 Mb of RAM is enough to run any number of security checks as often as needed.

The empirical validation showed that emulators and devices with behavior that deviates from the expected of an untouched physical device, no matter the precautions we took as an attacker, are always detected by at least one of the checks. Further, attempts to tamper with the application and repackage it in such a way to conceal the repackaging, again at all times were caught by the security checks.

Chapter Two
Background & Related Work

This chapter describes required knowledge for understanding the underlying principles employed in the attestation techniques that are proposed in this thesis accompanied by a presentation of the related research about tampering and malicious reverse engineering. Detailed information about android apk signing will be given and we will describe why it is considered a security feature. Further, the android boot process will be explained and we will elaborate on some key points that prove to be essential for the security of android. We will see that for app-anti-tampering much effort has been put into signature-based schemes however little have been done for root detection and the evaluation of the execution environment.

2.1 Background

2.1.1 Android Signing Process

Google Play along with other app-stores required applications to be digitally signed by their developers. The purpose of signing an app is to be able to identify the author of the application. The signing process is considered as a bridge of trust between Google and the developer, but also the developer and the application. Due to the fact that the secret key used for signing is known only to the developer, no one else is able to sign the application with that key and thus the developer knows that the presence of his signature in the app means that the application is not tampered. Additionally the signature means that the developer can be held accountable for the behavior of his application.

In Android, *Application Sandbox* for each application is defined by the app signature. The certificate, is used to identify which user ID is associated with which application. This means that different applications run with different user IDs. The package manager verifies the proper signing of any application installed on the device. If the public key in the certificate matches any key used to sign other applications in the device it is possible for the application that is being installed to be allowed to share a user ID with other apps signed with the same public key. This has to be specified in the manifest file of the app being installed.

It is important to note that Android does not perform CA verification for the certificates in the applications. Therefore, applications can be signed by a third party or even being self-signed. Additionally applications are able to declare security permissions based on the signature they bare. Applications being signed with the same key, means they are developed from the same author, so a shared *Application Sandbox* can be allowed via the shared user ID feature [4].

Versions

The most recent is the third version of Apk signature scheme. V1 method was offering a wide attack surface by not protecting some parts of the Apk, such as the ZIP metadata. This had a twofold negative impact, first the *APK Verifier* had to process lots of untrusted data and second all that data needed to be uncompressed, consuming time and memory. The V2 Signature Scheme was introduces with Android 7.

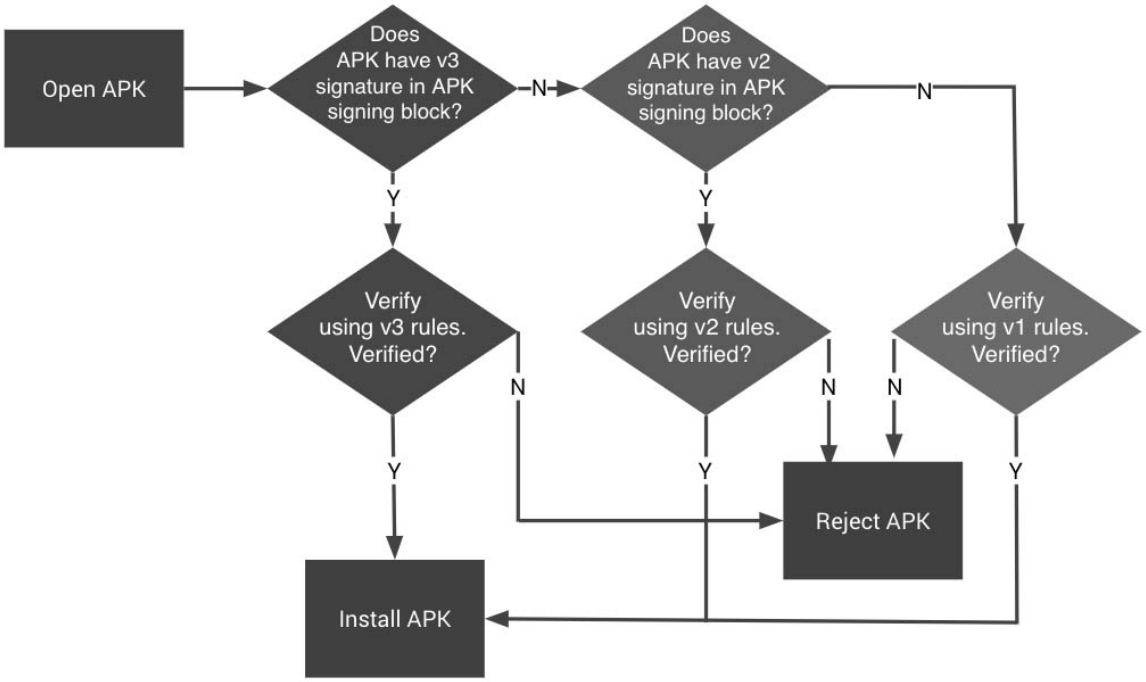

Figure 2.1 APK signing Process

Structure of APK

As the foundation for the APK format, the widely known ZIP format was used. An APK is essentially a ZIP archive that includes all the required files for an application to be executed by the Android system. For the purposes of protecting the APK, it is splitted in four regions shown in figure 2.2. The section of the APK Signing Block protects the integrity of the rest of the sections along with the signed data blocks which are included in the same block.

1. Contents of ZIP entries	2. APK Signing Block	3. ZIP Central Directory	4. ZIP End of Central Directory

Figure 2.2 APK sections after signing

Inside every signed APK we will find a directory named *META-INF* which contains information about the public key used to sign the application. The file *CERT.RSA* inside the *META-INF* directory, is of special importance as it contains the public key, which is used to verify the integrity of the application.

Procedure

The difference between V2 scheme and V3, is that V3 supports additional information in the signing block. From Android 9 and onward V3 scheme is supported. The procedure of signing starts with the contents of the APK being hashed and then the *signing block* created being added into the Apk. During the validation process the entire Apk contents except the *signing block*, are verified. This automatically means that any modification to the contents will invalidate the signature and thus the attacker can no longer sign the app with the original signature. This is why the signing process is treated as a security feature. In figure 2.1 we see the decision process that occurs upon trying to install an apk to Android. The installer verifies the integrity of the apk based on the version of the signature it discovers.

Tampering

App signing proves to be an important feature in tampering an application. A tampered application not only would create a different hash of contents but also since the attacker does not have access to the private key of the developer, it would eventually have a different signature as well. An attacker would have to circumvent any check made by the application itself to verify that the correct signature is in place.

2.1.2 Android Boot Process

The boot process seems essential to Android Security. For an attacker taking advantage at the right time during startup could potentially mean a much greater risk in security. Following we are showing briefly the boot process explaining the basic parts.

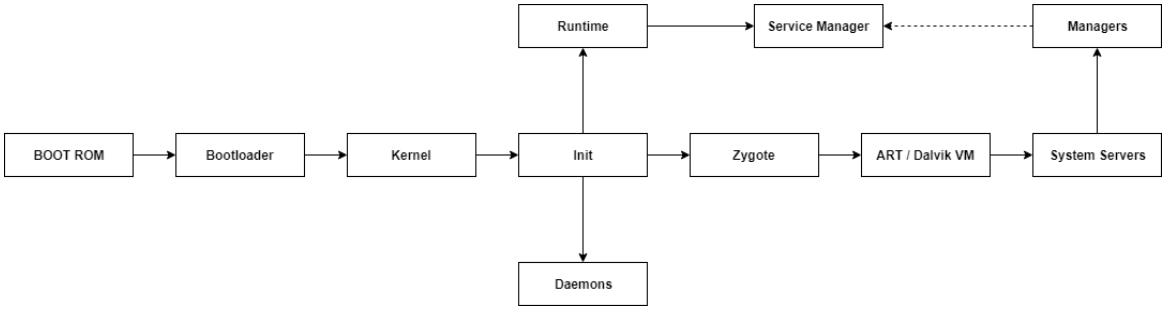

Figure 2.3 Android Boot Process

Boot Rom: After we hold the power button in order to start the device, the code responsible for the boot rom is starting from a predefined location in memory. The task of the code is to load the bootloader into ram and execute it.

Bootloader: As a first step the bootloader detects the available RAM and as a second step it executes an application to set the low-level memory management, the network and any security options defined usually by the manufacturer of the device.

Kernel: Initially kernel is responsible to start all hardware, driver and file system related operations. After the process of setting everything up comes to an end, the kernel searches for the *init* in the system files. This is how the first user-space process is started.

init Process: Init process as the first process has two main tasks, the first is to run the *init.rc* script and the secong to run a similar file but with device specific initialization. The result of this procedure is starting processes called daemons. Furthermore, init starts the *Runtime Process* which is responsible for the service manager which handle the service registration and lookup. Finally the init process starts up the (in)famous process called Zygote.

Zygote & ART / Dalvik VM: The *Zygote* process includes all standard core packages and it is the process being forked when a new application is launched in our device. It enable code sharing across ART/Dalvik VM which results in minimal startup times of the applications. It is a very important step in the Android environment since all new applications are based on it. We could say that this would be an ideal point of attack for someone who would want to tamper an application. At last *Zygote* forks a new ART/Dalvik instalnce and starts the *System Server*.

System Servers: The system server started by *Zygote* starts is its turn system servers like *SurfaceFlinger* and System Services like *Power Manager* or *Telephony Registry* and many more.

After all of the above, the system is ready to launch the home application. Our main point of interest in the entire boot process is the understanding of how important the role of *Zygote* actually

is.

2.1.3 Android Root Access

In Linux and Unix-Like operating systems we refer to *"root"* as an entity with administrative/superuser capabilities. Root is able to have full control of all aspects of the operating system with no limitations. On the other hand, a normal user in a system like this, has confined access to the filesystem and very specific permissions as to what she/he is capable of doing. In Android, limitations and boundaries as to what a user is allowed to do are specified by the carriers and the hardware manufacturers, in order to protect the device from users accidentally or not, altering or replacing system applications and settings [37].

A major benefit from rooting a device, is that it gives the ability (or permission) to install and run applications that require root level access and alter system security settings otherwise inaccessible to a normal Android user. Further, the entire operating system can now be replaced with a custom one (Custom ROM) that would not apply any restrictions to the user letting her/him modify any part of it at will.

Common reasons as to why users root their devices is, to remove "bloat" content that the manufacturers install to the device and can not be removed without root access or to use applications like *GameGuardian*[1] which allow memory manipulation in order to tamper an app. There is no limitation to what the user can do and thus this is identified as a dangerous environment for an application to run, as tampering might be imminent.

There are, two main ways in obtaining root access, one through discovering a vulnerability in the device and the second through executing the "su" binary from within the device. The first one is refereed to, as "soft root" while the later one as "hard root". Both methods result in altering system files and properties, that can be used as a mean to identify their existence in the system.

The release of Jelly Bean led to a transition to the *su daemon mode* as the *su binaries* ceized to work. This daemon is launched during the boot process in order to handle all the superuser requests made by the applications when they execute the *su binary*. This option though was viable only for a limited time as the release of Android 5 was set with the SELinux being strictly enforcing, limiting this way even what an account with root access can do. The solution to this issue was to start a new service with unrestricted *Super Context* and in order to do that a patch to the SELinux policy in the kernel was required. This is what done for later versions after Android 5 and this is also what Magisk[2] is doing.

How Magisk Works and Why it hard to detect it

In order for Magisk to be installed to a device it is required to have an unlocked bootloader so that a custom recovery can be installed and modify the *boot.img* or a directly modified *boot.img* could be "flashed" from *fastboot*. After the device is capable of booting using the patched *boot.img*, a privileged Magisk daemon runs. This daemon has full capabilities of root with an unrestricted SELinux context. When an app from the device requests root access, it executes the Magisk's *su binary*, which is accessible to it, by Discretionary Access Control and Mandatory access control. This way of accessing the *su binary* does not change the *UID/GID*, it only connects the daemon through a *UNIX socket* and provides the requesting app a root shell. Additionally, the daemon is hooked with the Magisk Manager app, an app that can display to the user the requests made from other apps for root access and let the user decide whether to grant the access or not. A database is maintained containing all the granted or denied permission requests.

Magisk replaces the *init* file with a custom one, which patches the SELinux policy rules with an unrestricted context and defines the service to launch the Magisk daemon with this context. After that

[1] https://gameguardian.net
[2] https://github.com/topjohnwu/Magisk

step the original *init* file is executed to continue the booting process normally. This way the Magisk daemon manages to have unrestricted root access beyond SELinux limitations.

Magisk is referred to, as *Systemless root*, due to the modified *init* file is inside the *boot.img* and thus the */system* is not required to be modified. In case of devices where the *init* is located in the */system* partition then Magisk places the modified *init* in the recovery partition that is used to boot Magisk.

This systemless approach has a major benefit of allowing the use of Magisk Modules. A Magisk module may refer to modifications a user may require to do to the *system* partition, like adding a binary under the */system/bin/* or modify the contents of a file. This happens without actually modifying the */system* partition, using the Magisk Mount, which is a property of Magisk based on bind mounts[3]. Magisk supports adding and removing files by overlaying them and this way the *systemless* feature is retained.

Magisk supports hiding its presence from applications security checks. This way it becomes hard for an application to realize it is running in a rooted environment. This feature is possible by taking advantage the bind mounts we mentioned. The *Zygote* is essential to this as it is monitored for any new forked processes and it efficiently hides from them any sign of modification based on user choice. It is worth noting here, that the security service introduced by Google, named SafetyNet, which can tell apps if the device is safe to be trusted, can be easily bypassed using the properties of Magisk.

2.2 Related Work

In this thesis we focus on tampering detection and techniques to avoid malicious reverse engineering in Android. We split our work into two main categories, techniques to detect that an application has been tampered with, meaning that they are dealing with the problem after the application has already been modified and techniques that evaluate the execution environment on whether it is safe or not for an application to trust it.

2.2.1 Application Integrity

Application repackaging has been to the center of attention for quite some time now. Developers and researchers have been interested in identifying apps that are repackaged and work has been done towards this direction. In [28] and [41] they analyze statically a wide range of application from marketplaces and based on some different features extraction they create a database which compares against the similarity of each app found. If the threshold, in each case, is between a predefined range, they know that this app has probably been repackaged. This is a great solution when talking for a bulk number of app comparison but requires the use of a database to compare and it assumes the attacker has not made extensive changes to the app, in order to not exceed the limit of the similarity match. Additionally in this case we presume that we have in our hands the tampered app so we can analyse it which is not always the case.

Another hard to remove by an attacker, approach has been described in [40]. In this paper they propose a watermarking mechanism capable of being identified in an application even after it has been repackaged and resilient against its removal. The entire process is automatic and can help identify that an app is indeed the one we are looking for. This idea though, again as the previous one suggests that we have the tampered application in our hands to evaluate whether the watermark is present or not.

Another more general approach is given by [18] after analysing the security of bank applications in Korea and they were able to repackage them. They propose three measures against repackaging, first being a self-signing restriction which though also violates the Android‘s open policy, second code obfuscation which would substantially increase the time an attacker needs to reverse the app and third and most important one, the use of TPM (Trusted Platform Module). TPM is a hardware based

[3]https://github.com/topjohnwu/Magisk/blob/master/docs/details.md

security measure that could be embedded into a smartphone but the drawback is that there would be increased costs from additional hardware. They mention additionally that TPM would enable a remote attestation check which would check whether the app has been forged before any interaction with the app. However our implementations bypass the requirement for TPM in order to run remote attestation checks.

Finally in [23] they attempt to directly access the public key of the signature of the apk using multiple points in the code to do so. They obfuscate in different ways the multiple checks they make in the code making a seamless integration in a way that it is very hard for an attacker to identify them. Additionally upon detecting a repackaging attack instead of triggering a failure instantly they transmit the detection result to another node, making it hard for an attacker to pinpoint the location of origin of the failure. However, they acquire the public key from the apk using the *Package Manager*, which as shown in Chapter 3.1 the call can be hooked easily and in that case no matter how many checks are introduced a spoofed response will be sent back. This defeats the entire idea presented by the paper as no matter how many checks they are introduced they are all based on the *Package Manager*.

2.2.2 Remote Attestation

A vital part of the evaluation of the execution environment is identifying the existence of root in the device. According to[34], after an extensive test of applications they concluded that from the available techniques they studied to detect rooted devices, all of them were able to be evaded. It is important to note here that based on our searches for papers dedicated to root detection, we could not find any other sources published within the last four years.

Protection against potential malicious debugging and dynamic analysis of the app becomes essential as it is usually what attackers resort to in order to bypass app protection. A scheme for protection apps is described in [8], where they employ timing checks and direct debugging detection checks. In timing checks they evaluate the time taken between two points in the flow of the code, as it normally the execution is a single step but when a debugger is attached this part might take longer after the attacker has stopped the execution at some point. This way the program will fail and the execution will stop not allowing the attacker to continue debugging. For the direct debugging checks they detect using *signals* send to the debugger and through the *TracerPID's* value from reading the */proc/PID/status*.

A kernel based approach is proposed by [36] where a kernel-level Android application protection scheme can protect all application data. An encryption system is designed and implemented based on Linux kernel *3.18.14* that is able to verify the validity of the design. This is a great approach given the fact that the kernel will not be altered by the attacker. Though this is not the case, as usually the attacker will have full control over the device she/he uses and therefore the kernel will be substituted with one of the attackers choice.

In the paper [7] the researchers approach the remote attestation for Google Chrome OS, which although may be different from Android they share some basic principles. Their approach is based on the combination of two integral aspects of Chrome OS: (1) its Verified Boot procedure and (2) its extensible, app-based architecture. We are interested in the first part which is the verified boot procedure. Verified Boot ensures the integrity of the static operating system base including firmware, kernel and user land code. Most of the tampering capabilities require an unlocked bootloader in order to be able to patch the kernel and perhaps also the firmware of the device. This would mean that any modification to the kernel or the firmware would trigger the security check suggested here. This is an excellent security measure if implemented in Android but has one drawback, how is it going to be enforced. Depending on how it is enforced would allow or not the attackers to remove it. For example if a company that produces mobile devices does not offer any capability to its clients to unlock the bootloader then it would be difficult and would require a lot of time for an attacker to unlock the bootloader. But on the other hand it would result in many unhappy clients since they do not have full

control over their devices and from a selling point of view it has a negative impact to the company. Once again we see that the security is inversely proportional to the convenience of the clients.

Chapter Three
Threat Model

In this chapter we show the possible attacks that can be performed in order to alter the behavior of an application according to the goals of the attacker. We describe our threat model and provide an analysis on how it results in compromising the integrity of the application and bypass possible existing checks. The consequences of such attacks is partial or even complete alteration of behavior of the application. The threat model we investigate is for app tampering and app debugging.

In all the cases we suppose that a malicious user does not have infinite time or resources and that the basic principles of obfuscation and encryption are used where applicable.

3.1 App Tampering

A famous digital music streaming service that offers a freemium model, has employed integrity checks at runtime that verify the signature of the app running is the one used by the company to sign the apk. A popular attack, malicious users take advantage of, is app repackaging [**Androidrepackaging**]. In this attack, our music application would be reverse engineered and payloads that offer the premium services for free, plus any additional code the attacker decides will be embedded into the app. Then the attacker using a private key of her/his own would sign the app and uploaded into open marketplaces where it would gain attention due to it offers the premium parts of the app for free. In such a scenario both the company behind the app and the user, are victimized, as the app security checks against tampering are failing to serve their purpose. Following we present you how this signature bypassing techniques are working and what are the conditions required each time.

3.1.1 Signature Spoofing

Signature spoofing is an attack which tricks existing security mechanisms of an app into believing that the signature of the application is the original one. The usefulness of this attack resides in the fact that it is now possible to bypass requirements enforced by Google Play, allowing devices with custom operating system to use an open implementation of the Google Play Services Framework [26]. Although not widely known, any application that runs on Android does direct certificate access, due to the way Google Play Services works [20]. By direct certificate access we mean a request to get the signature of a package using the package manager's *GET_SIGNATURES* feature. This way Google verifies the availability of *Play Services app* and *Play Store* along with the integrity of them, by checking the key used to sign them (3.1).

Another very frequent use case, which is of more interest to us, is the bypassing of DRM. This one perhaps is the most popular use case, since it allows tampered applications to report the original signature, thus bypassing integrity security measures implemented within the application. In many cases developers of applications in an attempt to secure the contents of the application and also protect their users, they also do a direct certificate access. This means that if signature spoofing is not enabled then the application would fail the check enforced by the developer and subsequently alert the user of the tampering of the application or do some other preconfigured action. Following we divide signature spoofing into two categories and analyze the basic usage in each one.

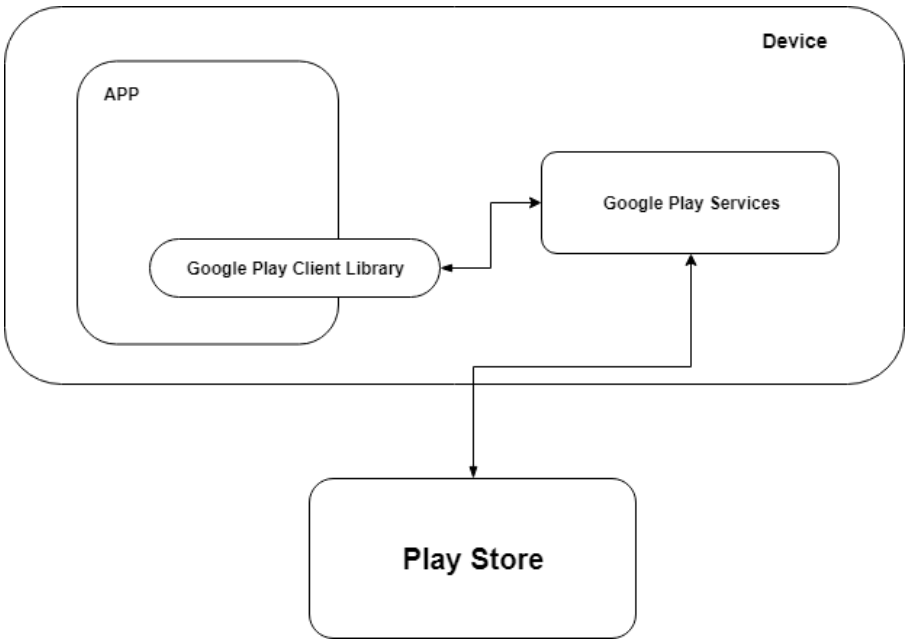

Figure 3.1 Google Play Client Library Calls

Device Specific

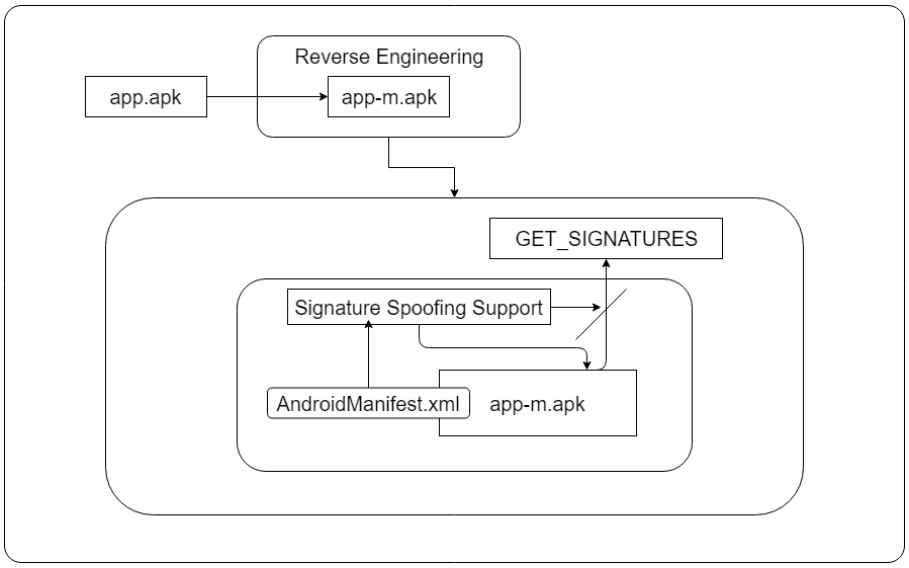

Figure 3.2 Signature spoofing process in Device

Signature spoofing can be device specific, in the sense that the owner of the device is required to have modified the device to support this feature. There are three main ways someone can add support for signature spoofing. We will not analyze them as the logic behind the mechanisms they employ are similar.

- Using Xposed Module[1]
- Patching the system with a tool like *Tingle*[2]
- Using custom ROM

[1]https://repo.xposed.info/
[2]https://github.com/ale5000-git/tingle

There is an increasing number of custom ROMs already including signature spoofing and at least three updated tools offering system patching for it at the date of writing (May 2019). In a device with signature spoofing enabled any application who wants to use the feature must first, announce the spoofed certificate that it wants to be reported back in the *AndroidManifest.xml* and secondly it is required to request the *Android.permission.FAKE_SIGNATURE* permission. This implies that the user is notified and must consent, and for Android 6 and later the user knows even more explicitly about the permission and can decide to grant it or not.

Following our example scenario we set in the beginning of this chapter, we depict in figure 3.2 an overview of the entire process. An attacker reverse engineers the music streaming app *app.apk* and modifies it to offer for free the premium content it includes. Then repackages and signs it with her/his own key, which would be different from the secret one the original owner of the application has. The attacker has already specified within the *AndroidManifest.xml* in the modified *app-m.apk* the spoofed certificate to be used and has requested the *Android.permission.FAKE_SIGNATURE* permission. Then she/he installs the *app-m.apk* in a device that supports signature spoofing and starts the application. The application shows the dialog allowing the user to decide if he wants to grant the required permission to the application, and the attacker accepts. Any integrity checking security feature based on the package manager's *GET_SIGNATURES* is now compromised as the Android System will report the spoofed certificate upon demand.

The minimum difficulty in bypassing the most known check for the signature of an APK, immediately implies the need for an innovative solution to the issue. We showed that for a malicious user it is enough to have access to the aforementioned tools and a limited knowledge and understanding in order to accomplish the given goal.

Application Specific

This method is mostly used as a mean to bypass DRM and the success of it is based on the fact that a user does not need to modify the device he owns. Everything is handled within the application minimizing this way the effort and time someone needs to invest to be able to run a tampered application with signature checking security capabilities. The attack inserts code into Application class - or creates it, hooks the *getPackageInfo* method of the PackageManager then returns an original signature on demand (3.3).

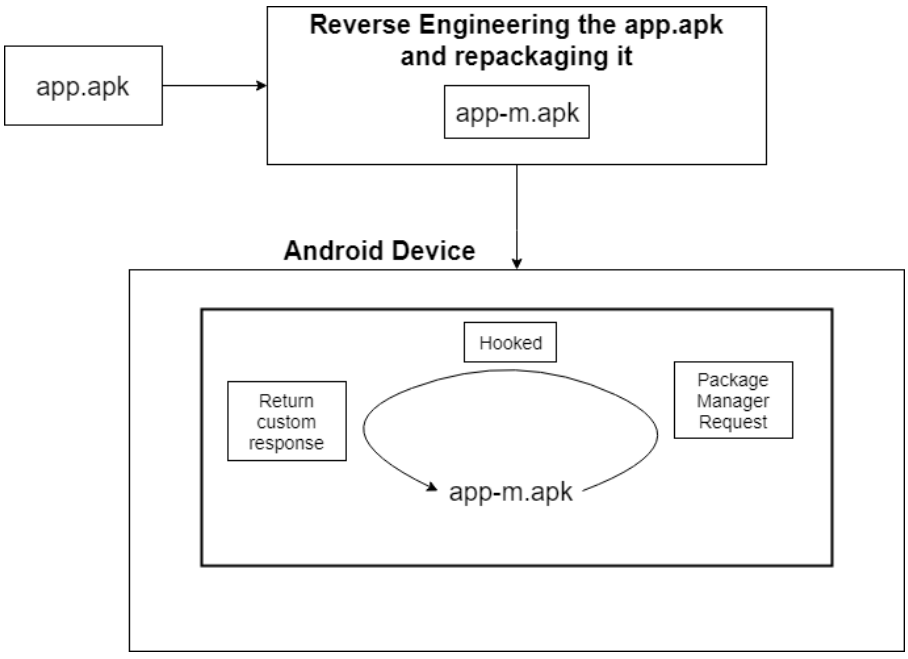

Figure 3.3 Signature spoofing within the APK

The result of this method renders useless any check within the application based on the package man-

ager's *GET_SIGNATURES* feature. The simplicity and portability of this method establishes it as a very powerful attack against the most known way to retrieve the signature of an app in Android.

Further, we should mention that, it requires no knowledge of the application's structure and there are available tools exploiting signature spoofing as described above, where the only requirement is to provide the original *apk* file in order to extract the correct signature from it [22].

3.2 Dynamic Binary Instrumentation

When static analysis comes to be unfruitful, dynamic analysis can yield better results. Debugging is a valuable resource when developing an application, but is also highly effective in the malicious reverse engineering process. To take fully advantage of the tools available for debugging, usually a suitable environment is requested, which for example would include root access and capabilities of emulation. A common practice while reverse engineering Android applications is to use a *hooking* framework or dynamic instrumentation tools. There are many available tools for this purpose including Frida and Xposed Framework [11], [6], [29], [12]. These tools differ in the way the hooking to the methods occur. The main difference is in the depth from the perspective of invocation of the method, nevertheless the result in each case remains the same. Following we present an analysis of how the Xposed Framework works along with an attack scenario of an application.

3.2.1 Xposed Framework

Zygote is a special process in Android able to handle the forking of each new application process. Every application starts as a fork of Zygote. The process starts with the */system/bin/app_process* which loads the appropriate classes and invokes the initialization methods. Upon installation of the Xposed Framework, an extended *app_process* is created in */system/bin*. This enables the framework to add an additional jar to the classpath and call methods at specified locations. Now Xposed is able to call methods at any time during spawning a new process, even before the main method of Zygote has been called[29].

The main advantage Xposed Framework offers is the ability to hook into method calls. The hooking capabilities allow an attacker to inject his own code right before or right after a method call and thus modifying the behavior of the application. Additionally there is the option to change the method type to *native* and link the method to a native implementation of the attacker. This would mean that every time the hooked method is called the implementation of the attacker will be called instead of the original one as can be also depicted in figure 3.4. Furthermore we should mention here that Xposed Framework supports creating modules. With the use of modules, a modification a user has managed to make in one application, he can automate it and deploy it into a module where it would be available for everyone to add to their own installation of the framework in their device. Modules are created as simple Android applications with specific characteristics to be recognisable by the Xposed Framework.

Again returning in our example with the music streaming app. We know that for additional security the developers of the application decided to "hide" a sensitive part of the application that does the checking of whether the user is a premium member or not, using the *Java Native Interface*. This was decided, due to the Java bytecode is closer to the source code and thus easier to reverse engineer than the compiled code of of C [31]. Therefore, the developers of the music app, thought that an implementation in C, would delay presumably an attacker even more since he would have to also reverse the *.so* files included. The attacker now, after analyzing the application can decide what is the best interest to him in regard to the checks the application performs. The best course of action in this case is using the Xposed Framework to hook into the *native method* and alter the behavior of that part. We can see, that although the developer of the application went to greater depths to protect the application it made only a slight difference to the effort required by the attacker. Furthermore, the module created by the attacker can be shared with anyone else using the framework and thus automate the way to bypass the check made by this specific music app.

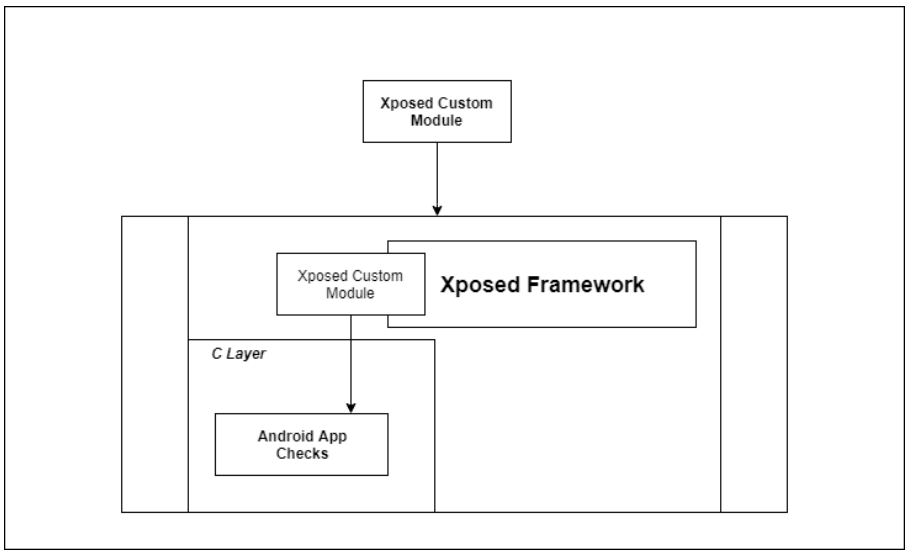

Figure 3.4 Hooking with module in Xposed Framework

There are many more frameworks currently active in Android dynamic binary instrumentation, offering a lot more capabilities than Xposed. We analyzed Xposed only to show that even a slightly outdated framework can manage to bypass most of the security checks implemented now-days.

Chapter Four
Remote Attestation

In this chapter, we start by explaining the entire architecture which is used as a full countermeasure against application tampering. Furthermore, we present what checks are included in the remote attestation and what purpose they serve. It is worth noting, that all the security checks are made as to require no special permissions from the Android device. This way the implementations can be combined with any Android app without interfering or imposing permissions that the author of the app might not want.

4.1 Architecture

The techniques and implementations presented in this thesis constitute only a part in the entire process used against application tampering. The final product, under the company *2Aspire SRL*, is designed to work with the interaction of all the components together and although the part described in this thesis could work as standalone, in order to get a solid defence system against application tampering, all the parts are required.

The service offered by the company as protection of an Android app comes in the form of a Gradle Plugin. A developer of an application who is concerned about the security of the application and would like to mitigate any substantial risk for tampering would purchase this service. The plugin is available then to the developer to be used directly on the android project of the application. All the developer has to do is specify the exact parts of the app that need to be protected, as we are going to see in the presentation of the two basic parts of the tool. Upon compiling the application for release, the plugin would automatically handle all the steps required to harden the security of the app against malicious reverse engineering and tampering.

There are two basic components included in the plugin, the *check component* and the *split component*. They are designed to work together and in collaboration to a server.

Check component: The check component is responsible for testing several aspects of the integrity of the application and of the execution environment. The main purpose is to verify that the application has not been tampered with and to avert such attempts. The result of the testing is sent back to a server where the output is logged and analyzed to verify any inconsistencies.

Split component: The split component is used selectively by the author of the application in order to detach a method from the application‘s code and move it to the server. This would mean that the installed apk would be incomplete and is unable to run on its own since some required parts are moved to the server component. Upon a normal execution of the application, the check component would verify the integrity of the app and a proper environment and then the server would allow the split component to respond to that specific device running the app with the part of the code that split has extracted.

Following we have an analytical overview of how the process works, also depicted in figure 4.1. The application starts in the Android device, and when it reaches a predetermined location the check component will be started. Several aspects of the application and the execution environment will be

checked and reported back to a server. The server will evaluate the response received and identify if the application has been tampered with or if the execution environment indicates any sign of malicious reverse engineering. If the report from the check component triggers no alarm to the server, then the specific device is white-listed for a short amount of time. The distinction between devices is done using the *Settings.Secure.ANDROID_ID*[1], which is a 64-bit key different per app and per user in an Android device. The server stores these values for the white-listed devices, for a short period of time, in order to allow interaction of the specific device with the split part. The application is granted access to the part of the code that is missing due to the split component. This would allow the application to continue normally the execution without any interruption.

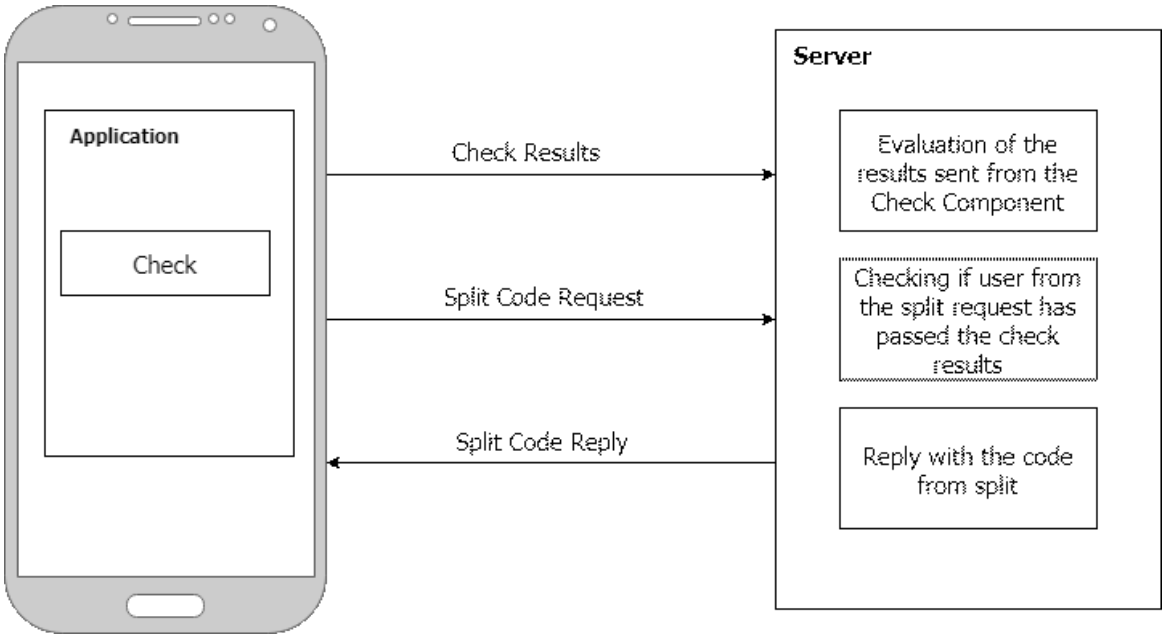

Figure 4.1 Architecture of Communication

As we can see in figure 4.1 the *check component* initiates the communication to the server by sending to it the report from the result. This means that the *check component* can be executed independently from other entities and their status. We mention this to emphasize the difference between other services like Google Play when to initiate a testing of the app the server has to first send a "nonce" value to be used. The communication architecture we chose reduces the time taken for the check to complete and also allows a simpler configuration of the overall architecture that is deployed. In regard to defending against "replay attacks" due to the absence of a "nonce" value, a security mechanism is in place, explained more thoroughly in chapter 5.1.2.

Emphasis is given to why this particular interaction of the tools is chosen. Nowdays that there are several tools and frameworks to reverse engineer an Android application and thus it is possible given enough time to bypass all security checks embedded in an app. Taking as an example the famous music streaming app we have mentioned before, an attacker could reverse engineer the app and tamper it in such a way that all the checks employed by the *check component* would be removed. This would practically mean that there is no way to stop the execution of the app since all the checks are removed. Indeed, in such a scenario the app would not stop its execution upon running in a compromised environment but it would also not send the proper responses to the server in order to be granted access to the *split component* part of the code. This interaction of the two components and the server, ensures that the attacker will have to send data to the server, implying that bypassing this architecture becomes more complicated.

[1]https://developer.android.com/reference/android/provider/Settings.Secure#ANDROID_ID

Usage by the Developer

The easy deployment of a security mechanism is crucial for the adoption rate from the developers. As we already mentioned the security protections we offer come in the form of a Gradle Plugin. The developer includes the plugin to the development environment of Android Studio and from there she/he can call the two components, the check and the split at will. As a first step, the developer should choose an important method of the app, and annotate it properly so the split component would extract this part of the code and move it to the server side. This way we ensure that the application would not work unless the user is granted access to this part of the code, according to the process we described earlier. The annotation for the use of the check component can be used as many times as the developer wishes throughout the app. It is important to note that, at least one check should occur before reaching the required missing code, so the server would already know if the device contacting is white-listed or not.

At which methods the security check should be initiated is something decided by the developer each time based on her/his judgement. A security check that occurs only when a method is accessed might lead to the application being tampered with, successfully by an attacker, as the security check might not be triggered at all if the attacker does not access the method required. The frequency of the security checks along to the choice of the method to trigger them, is a responsibility delegated to the developer. This allows the security checks to be adoptable to the different requirements that different apps will have.

4.2 Remote Attestation Checks

In most cases where an attempt to reverse engineer an application takes place, the attacker has full control over the device he is using. This means that the common security measures that Android uses can not be considered in effect. The attacker will make the environment of the execution fertile for such attacks, by disabling any defence measures Android presents that could detect the application tampering. In other words, the attacker has *root access* to the device allowing him to alter any system permission or property at will. It is understandable now why it is essential for an application not to trust the environment it is being executed on. Having as our guide all the possible modifications to a running system an attacker would do, we design several checks that we assume will increase the confidence of trust (or not) to the execution environment.

Signature Checking

In the scenario that the attacker has managed to bypass all the security checks implemented, the application would be modified and if repackaged would result in having a different signature as we already discussed in chapter 3.1.1. By implementing one additional security check we can verify whether the application has been tampered or not based on the signature it reports back to the server. The way that is used to retrieve the signature of the app is critical in the trust we can have in the reported signature value, as many tools exist that already manage to automatically bypass known methods developers employ.

Emulator

Although it is important to identify whether the execution is occurring in a physical device or an emulator, the distinction between the two is not always straight-forward. There is a variety of emulators for android devices in all operating systems. Each emulator has each own characteristics that set it apart from others of the kind. These fine grained details are what we can use as a tool to distinguish these emulators from actual devices. There are many methods employed for our purpose, it can be a system file that is present or even the name of a running process. We will discuss more details in our approach on the chapter 5.2.6.

Root Access

The next and highly critical check, is the verification of root access to the system. Gaining root access as a user in Android devices on the one hand has a lot of signs that can not hide it, since system files are tampered with, on the other hand since the user is already root it is possible to masquerade any modification. Additionally there are techniques of adding root access without modifying system files (system-less root), which are considered impossible to detect with maximum certainty.

Root detection is a subject that has been in the center of attention for anti-tampering mechanisms since the beginning. A lot of libraries and quite a lot of papers as well have been published on this matter [10] [1] [30] [15]. The issue though remains unchanged and it is mainly because once a method of detection is known the root manager used in the device will implement some mechanisms against that detection.

> "A central design point of the Android security architecture is that no app, by default, has permission to perform any operations that would adversely impact other apps, the operating system, or the user. This includes reading or writing the user's private data (such as contacts or emails), reading or writing another app's files, performing network access, keeping the device awake, and so on. [5]"

Permissions

Suspicious permissions of applications or file-system permissions could indicate a malicious environment as well, besides the fact that the presence of root could be implied. As already mentioned above, the bypassing of the default security protections offered by Android usually results in several system files with different permissions than normal or even additional files that should not exist. That means that, in the case the attacker is already familiar with the root detection mechanisms that are implemented in an application, he could bypass them but he would have to do an extra step in the case there are additional checks for permissions of the file-system or for specific applications. Again in this case if the attacker already knows that these checks are in place or if he guesses what are the most common permissions that are checked he could patch the system so it would respond appropriately and fool the detection mechanism.

SELinux

Selinux[3] as the name states, Security Enhanced Linux was introduced to Android as a way to enforce mandatory access control over all processes. Further, SELinux has played a substantial role in enhancing the security of Android. In Android 5.x and later SELinux is set to enforced by default [3], providing this way a security hardening mechanism. The enforcing status of SELinux limits what can be accessed or modified even by a root user. This means that even if root is present with the status of SELinux being enforcing, modification to system files would not be possible. For this reason often in Android devices the kernel is patched to support a permissive or disabled SELinux. Now SELinux would not interfere with any action of a root user allowing her/him full control. This is a detail we have included in our security checks as it can provide valuable information for the environment.

Android Properties

Continuing in our list of potential signs of a hostile environment we present the android system properties. *Default.prop* is a file which initially it resides in *boot.img*, and upon boot of the device it is copied in the *System* directory. The *Default.prop* file contains all necessary information for the ROM specific build. It can provide us with information whether the *debuggable* flag is enabled or if *adb* can run as root along with many more information. These information are essential in identifying an environment setup for reverse engineering.

Dynamic Binary Instrumentation Frameworks

Extending on the sense of what can we find in the Android System, we will also talk about loaded libraries and open ports, both of which hold a strong argumentative position in convincing us that a *fishy* system is in place. In many frameworks like the *Xposed Framework* we saw in chapter 2, an addition of libraries is required in order to work properly. Depending on the case it might be to our advantage to check whether such a library is loaded to the system or not. The fact that a library of *Xposed* could be detected strongly suggests a malicious actor due to the framework itself exists mainly to help in tampering/modifying Android itself or applications of it. Further in regard to the open ports, debugging and instrumentation frameworks require custom ports to be open in order for the application to be able to contact the attacker and report or receive new content or commands.

Suspicious Apps

Not all intelligence we can gather have the same weight of importance. Some indications give us more certainty than others. This is something we should keep under consideration for all mentioned cases and also most importantly upon the next candidate in our list, the existence of usually installed applications in tampered systems. There are some patterns attackers follow mainly due to habits of famous or most used applications. An example would be a root manager which will manage which application are allowed root access and for how long. The work of the root manager although important is not required in order to achieve root access, consequently a root manager might not exist even if root access is present. Another example is applications offering ad-blocking protection or applications designed to backup any part of the filesystem. Though, as already mentioned, these apps even if detected on a device they do no necessarily indicate the existence of root.

In the following chapter we are presenting all of our implementations, some direct and straightforward in their tasks and some reassembling side-channel attacks. Our goal is to assert a path of high certainty upon deciding the presence of a malicious environment so as to avoid the possible tampering of our application.

Chapter Five
Implementation

In this chapter, we describe exactly the steps we followed in our implementation and in each step we explain the choices we made and why. We split the section into three main parts based on the security issue each implementation tackles. Initially, we present the integrity checks implemented, followed by the environment checks which include root checking, emulator detection and custom Rom detection, while the last part is identifying binary instrumentation frameworks. While it was possible to make all the implementations using Java, we decided for the majority of them to be done using the Java Native Interface (JNI) and write them in C. The advantage this gives us is over reverse engineering techniques, since the Java Bytecode is much more similar to the original code and thus it provided more information to an attacker. On the other hand, the code written in C is harder to be reversed after being compiled. Also we should mention that throughout the implementation whenever hashing of a value was needed we used *SHA256*.

5.1 Integrity Checks

We will see two methods, capable to decide with high confidence whether the application has been tampered with. The first method involves the detection of the key used to sign the application employing a different path from the most common one. The second method is a way to judge if the application uses predetermined hard-coded values, in order to bypass the necessity for response to the server, as described in the chapter 4.1.

5.1.1 Application Signature Detection

As we have seen in chapter 2.1.1, the signature of an apk can give us information about its integrity and whether or not the original author has signed it. Following in chapter 3.1.1 we saw how the official way to retrieve the signature described by the Android Documentation is easy to attack. Now, we are presenting a way that the signature can be retrieved and is not vulnerable to the already known attacks. The idea is based on a StackOverflow post [19]. An overview of the process is shown in figure 5.1. We first acquire the path of the apk, unzip it and reach the *CERT.RSA* file and finally extract all the information required from there.

Figure 5.1 Process to retrieve Signature of APK

The process is straightforward and most importantly it does not require any Android Library that could be *hooked* by an attacker. It uses the minizip library [27] to unzip the contents of the apk and the pkcs7 library [21] to parse the *CERT.RSA* file which are included.

Initially, we get the package name of the application by using the */proc/self/cmdline*. Then we read */proc/self/maps*, which is a file containing the currently mapped memory regions and it includes

the pathname if the region in question is mapped from a file. In our case since the region is mapped using the *apk* the pathname would be the location of the apk. Next we examine if the package name is part of the path so we can find from all the paths returned the proper one we seek. Lastly after we have retrieved the path, we verify that it indeed includes an file with an *.apk* extension.

Finally, after we are confident of the path of the apk, we are using the *minizip library* to extract the content of the apk and more specifically we are insterested in the *META-INF/CERT.RSA* file. Using the *pkcs7 library*, we parse the certificate and extract the public key, which eventually after hashing it (SHA256) we add it to the *check report* that is to be sent to the server.

5.1.2 Magic Numbers

Due to our decision to keep the communication of the check one-way, we had to include a measure against replay attacks or hard-coded values a reverse engineer could add to the app in order to bypass some checks. For this reason, we propose the "Magic Numbers". This prevention measure is based on the simple logic of generating a number that follows specific rules. Then, the server verifies if the number is indeed generated with the same logic, and if it is, then the number would be accepted. Additionally, every execution of the check would produce a new, different number, thus any repetition of the same number multiple times would be recognisable from the server.

The implementation is part of the native code of the application, making it harder to reverse engineer. This simple measure has multifold benefits, such as:

- Simplifying the communication between server and device
- Less bandwidth consumption
- Faster completion of the check part since we do not have to wait for a server to send a nonce
- Security hardening against replay attacks
- Security hardening against hard coded values in the checks

The first three items in the list above, are compared to the alternative of having the server provide a nonce and the check would have to return that nonce. Since in our implementation we do not have any steps required before the check process begins, that results in a more simplified communication between server and device, less bandwidth and a faster completion of the security check for each run.

The security against replay attacks comes from the fact that a replay attack would reuse the same value and thus the server side would detect this. Finally, what we consider the most important item, is the detection of hard coded values. In more detail, let us assume an attacker is unwilling to spend time on reverse engineering the native part of the application given that it takes a significant longer time than reversing the java part. Then all the checks that are implemented in this native part should be replaced with values that the attacker knows that would be accepted by the server. That would require to also hard code the numbers generated as well. Eventually, the repetition of the same value would lead to the server detecting such an attempt. Of course if the native part is reversed then the attacker would be impossible to be traced.

In such a case, an alternative solution can be employed. In our case the implementations are to be used by applications that are presumably going to participate to Google Play Store. Therefore, it is required to follow the terms of service imposed by Google.

> "Apps or SDKs that download executable code, such as dex files or native code, from a source other than Google Play."

The above is a quote from what is considered malicious behavior by Google, and the developers of applications that intend to enter Google Play should avoid [14]. But, if we are not interested in entering Google Play Store, then we could adjust the native library loaded to the apk, as so it would be downloaded dynamically every time the check was required. This would allow to introduce new rules in creating the numbers and thus render useless any attempt to reverse engineer any previous version.

5.2 Environment Reconnaissance

Here we are presenting all of the methods and techniques employed to detect the execution environment. We will see a wide variety of techniques employed ranging from detecting values of Android Properties to dynamically assessing the permissions of paths and files.

5.2.1 Detecting Keys

One of the most well known ways to attempt detection of root in an Android device is by referring to the *ro.build.TAGS*. This is an Android property which contains tags describing the current build. These tags may provide information related to the keys used to sign the kernel when it was compiled like *unsigned* or *debug*. We specifically search for the existence of three strings, namely *test-keys*, *dev-keys* and *release-keys*. The first two strings we seek if found means that the kernel when was compiled was **not** signed by an official developer but from some third party. This instantly means that the device is not considered a safe environment due to any system app may be hijacked. On the other hand we expect to find the third string, the *release-keys* as it indicates that an official developer signed the build and we can trust it. We note that this technique has been reported to show some false positive results [17].

5.2.2 Wrong Permissions

All android devices have specified some parts of the filesystem that for security reasons remain only *read-only*, in order to avoid any accidental tampering which could have serious implications to the system itself. Yet, when a user desires to get root access it mounts specific directories with *read-write* permissions, in order to be able to modify any system file that may be required for a particular task. Which directories‘ permissions have been modified is not something stable, and it depends on what we are trying to accomplish each time. A part of the paths we used are originating from the famous library for root detection in Android, called RootBeer [1], and we included some additional paths based on our tests.

/system
/system/bin
system/sbin
system/xbin
/vendor/bin
/etc
/sbin

Table 5.1 Reliable Wrong Permissions Paths

In order to assert the validity of all the paths and verify their percentage of success, an extensive experiment including a high number of physical devices is required. In our tests several directories that were found with suspicious writable permissions on a rooted device we used for testing are not included in the table above as we have not verified their reliability. We should also keep in mind that currently the *trend* of gaining root access is shifted to systemless root like Magisk [38]. In systemless root the system directory remains untouched and access to the su binary is granted using bind mounts.

5.2.3 Unusual Android Properties

Using the Android properties again as we did with the detection of specific keys, we can determine if the system is allowing debugging or *adb*. We currently test for exactly three specific properties which we analyse below and explain why we chose them.

On Android, debuggers can communicate with the virtual machine using the JDWP, which is a protocol that allows other application development platforms to support debugging Android applications. A dedicated thread is created in the virtual machine waiting for JDWP traffic. The way the

VM knows whether to create that thread or not is through a flag known as *ro.debuggable* [2]. This flag alerts the VM that the application is under development and that regardless of a production device or not, the debugging protocol should be activated. This would allow an attacker to attempt a dynamic analysis on the application therefore it is something we need to avoid.

If we combine the above flag with the flag *ro.secure* and the flag *service.adb.root*, we have an indication not only about whether the debugging is in effect but also that it is allowed to run as root. This grants all privileges to the attacker connecting for example through *adb*, and allows her/him to manipulate at will every aspect of the device. This is common tactics used to remount the *system* directory and grant root access to the device, or when Dynamic Binary Instrumentation tools are used [9].

5.2.4 Selinux

SELinux works by white-listing explicitly what is allowed, everything else is denied. It works in three different modes, *Enforcing*, where the the permissions are enforced and logged, *Permissive*, where permissions are only logged and *Disabled*, where as you can guess, it is disabled. It is important because all android devices (since Android 5.0 [3]) have SELinux enabled and set as enforcing. Thus any finding that has SELinux in any other state indicated a modified environment.

In our implementation, we employ three different ways that allow us to detect the state of Selinux. Android properties come again to the rescue by providing us information through *ro.build.selinux* and *ro.boot.selinux*. The first one reports the status from the *boot.img* about Selinux on the current build, while the second one is to see if the Kernel uses the permissive parameter.

For the third way, we propose to detect Selinux by identifying the contents of the "enforce" file, which is a file holding the current state of SELinux. Based on our research, although this method is known for linux systems, we did not find any relevant detection mechanism implemented in Android. We start by reading */proc/filesystems*, which includes the filesystems that were compiled into the kernel or whose kernel modules are currently loaded. This way upon reading which filesystems are compiled into the kernel, we can attempt to detect if the *selinuxfs* is present, which would imply that SELinux was loaded in the kernel. This method has a twofold importance, first, it matters whether we can read at all the */proc/filesystems* and secondly if we find the *selinuxfs*. According to our reseach and testing after Android 8, the Selinux blocks reading several */proc/* locations, one of them being the *filesystems*[1].

In the case we are able to read */proc/filesystems* we know that Selinux is probably disabled or permissive. Alternatively, for older versions of Android, after we read the contents of */proc/filesystems*, we try to detect the value *selinuxfs*, and if we find it we know SELinux is running on the device. Though, we do not know in which state SELinux is at this point. In order to determine that we read the */proc/mounts* and track *selinuxfs*, so we can extract the absolute path, which we use to find a file of interest called *enforce*. As a last step we read this file and in the case the content is "1" the SELinux is in *enforcing* state and if it is "0" the state is *permissive*. This method proves to be stable and more reliable than the first two according to the results we see in figure 7.3.

5.2.5 Root Managers & Binaries

In this section, we attempt to detect directly the existence of root. The ideas described here involve the detection of binaries used for root or even specific files known to be used by root managers. This practice appears to be common amongst root detection techniques and thus it could be bypassed by some advanced managers [10]. Following we describe all the implementations made.

The first check we propose is to test what is the output of the command "*which su*". This command would reveal the existence of *su* binary and thus we would know that it is installed in the system.

As a next step we have a series of checks for files, that the existence of, is a clear indication of the presence of root in the system. One of the most famous root managers is the *SuperSu* manager. It is working by modifying the *system* files and it installs its apk in the path */system/app/Syperuser.apk*.

[1]https://github.com/CypherpunkArmory/UserLAnd/issues/59

This is a clear path we can check to see if it exists. Other paths we check are */data/adb/magisk.db*, */data/magisk/resetprop*, */cache/magisk.log* and */sbin/.magisk/mirror/data/adb/magisk.db* which obviously attempt to detect files that the *Magisk* manager uses. Additionally we check to see several known locations for the existence of binaries related to root like *busybox* and *su* and other properties specific to Magisk.

Finally, the presence of root managers and applications, that require root access to work, in the installed packages is also part of what we check. We are checking all the installed packages to find applications like the one presented in table 5.2.

com.saurik.substrate
com.topjohnwu.magisk
eu.chainfire.supersu
kingoroot.supersu
com.yellowes.su
com.chelpus.luckypatcher

Table 5.2 Applications Indicating Root Access

Some of the applications may offer root cloaking services, other offer a way to cheat in games and others are just managers of the root access. The common thing amongst them is that they all require root access in order to work, and thus we can deduct that root is present if such an application is found on the system.

5.2.6 Emulator Check

An important step towards identifying a hostile environment is to realize whether the application is running on an emulator or not. It is a common practice upon attempting to reverse engineer an application to run it on an emulator. Our approach on emulator detection involves identifying specific values in the Android Properties and known used directories of emulator engines. Each emulator engine reports some unique values that could easily help in identifying them. This method is also adopted by Facebook in the *react-native framework* [33] for detecting emulators through the Android Properties. Table 5.3 contains a sample of Android Properties that are being checked along with their corresponding values.

Android Property	**Value**
Build.FINGERPRINT	generic
Build.HARDWARE	goldfish
Build.MODEL	Emulator
Build.MANUFACTURER	Genymotion
Build.PRODUCT	google_sdk

Table 5.3 Emulator Detection From Android Properties

Though we knew that a detection through the Android properties would not be enough against an advanced attacker, so an additional light weight implementation is in place. Upon analyzing the Genymotion emulator and the Bluestacks we extracted paths and files that are unique to these emulators and can be used to identify their existence. The following table 5.4 includes an example of the paths we are checking against.

We believe that an extended analysis on more emulators could create a library of paths and files that can be evaluated on runtime to verify whether an emulator is in place or not. Such a library would introduce high credibility to the emulator detection technique as each file or path would be derived from a specific emulator engine.

/dev/vboxguest
/dev/vboxuser
/fstab.vbox86
/init.vbox86.rc
/data/bluestacks.prop
/data/data/com.bluestacks.appsettings
com.bluestacks.settings.apk

Table 5.4 Emulator Detection Paths

5.2.7 Device Profiling

After all the checks that we already implemented, we realized from our research that a significant part of trust is embedded on the Android Properties. We propose to take advantage of that by creating a database of the most common devices in such a way that the greater the number of devices that share some common properties the better the trust we could have. We picked 4 specific Android properties that are identifying a device and a model, so we can separate different devices of the same manufacturer for example, and also that are usually modified upon using a custom ROM. The properties chosen along with an example output from a device with custom ROM can be seen in the next table:

DEVICE	vince
MANUFACTURER	Xiaomi
PRODUCT	havoc_vince
FINGERPRINT	xiaomi/vince/vince:8.1.0/OPM1.171019.019/V10.0.4.0.OEGMIFH:user/release-keys

Table 5.5 Example of Android Properties in Device Profiling

As we can see in the table 5.5, the custom ROM of the device is available in the *Product* property, though that is not always the case. The *Fingerprint* property is designed to identify a specific build for a specific device and it is required by the Google Play Store in order for it to be able to show to the user only the applications that are capable of running to the device. This serves our purpose well since it provides information unique to the device and in combination with the other three properties which originally should be included in the *Fingerprint* we can differentiate the same device having the stock ROM with one that has a custom one. In our implementation we gather these properties and hash them together, sending the hash to the server along with the rest of the details. This feature upon having previous security checks of the same device allows the server to evaluate how many successful attempts are already registered for these kind of devices and therefore if the rate is high, have a higher sense of trust to the device. Further, this feature in case a suspicious result would come up for a device that has mostly successful security checks would allow a manual investigation as to which security flag was triggered.

5.3 DBI Frameworks

Dynamic Binary Instrumentation frameworks are the usual suspects upon reverse engineering an application. An attacker will use this kind of frameworks to bypass possible checks employed by the application in order to trick it and run on an unsafe environment. What we are trying to do is, analyse properties of two popular Frameworks and implement checks to identify them. The two frameworks we chose are, the Xposed Framework and Frida. We described in chapter 3.2.1, how an attacker would use the Xposed Framework to attack an application. Both of the frameworks offer the capability of hooking into methods the applications use and alter their behavior, with Frida, as a state of the art tool, which is constantly being updated, having many more capabilities than Xposed.

As a first step we are targeting specific files these frameworks require in order to operate. In order to run Frida on a device one of the possible ways, which is also the one mostly used is to run the *frida-*

server on a rooted device. We do this by using *adb push* and transferring the file inside the device, from where we also run it. We use this knowledge to search for the existence of the *frida-server* file in the device. Respectively, Xposed Framework has the *libxposed_art.so* library file, which is usually located in specific directories. This method of detecting specific files that the framework uses, is efficient and reliable though we have to keep in mind that it is easy to bypass, with only renaming the files. This check although fragile against advanced users is the first line of defence against inexperienced users, and the reason we included it, is due to it is highly reliable.

As a more advanced scenario we are using a similar technique to the one we saw in the *Application Signature Detection* section above. Both Xposed and Frida require some libraries to be mapped into the memory of the application in order to inject code. The concept is to use the */proc/self/maps*, which describes a region of continuous virtual memory in the application and attempt to detect any sign of the frameworks. What signs of the frameworks we are looking can be changed over time and it directly related to the latest changes of each framework. In order to be accurate we need to monitor any recent updates to the framework, analyze it and judge which would be a stable library or "gadget" that we may attempt to detect.

For the Xposed Framework we are looking for libraries like *app_process32_xposed* or *XposedBridge.jar* which we know are loaded by default. For Frida we specifically looking for the library *frida-agent-32.so* which is loaded upon hooking into an application.

This method employed although it goes deeper in the Frameworks, it is not without a vulnerable point. In our code we used *fopen* in order to read the */proc/self/maps* which would lead an experienced user in hooking into these calls. That means that we need further custom implementations to replace the libc library functions. This would require a significantly greater amount of time invested into reversing and of course a novice user would be discouraged against it. Although this is a very interesting step towards hardening the security and we have partially implemented it, we did not include it in the code used for the evaluation since more tests on the stability need to be completed first.
Finally, in our implementations we keep in mind to create checks which are as light as possible as we would not want a *"heavy"* load caused by a check which may run multiple times during the runtime of the application. This averted us from ideas like, running a full port scan to attempt to identify the port used for the framework to listen on.

Chapter Six
Empirical Validation of Performance

In this chapter, we evaluate the resource consumption our security checks add to the application we wish to safeguard. We run our tests on 5 different physical devices, through the *Firebase* platform from where we can get analytics on the CPU, RAM and Network consumption. The fact that the tests run on physical devices rather than emulators allow us to deduct more accurate and reliable results with lower margins for errors. Besides the analysis provided in this chapter, an extensive report from the *Firebase* results can be found in the Appendix **??**, where we show for each device the individual results for each test we made.

6.1 Context

Research Questions

We wish to evaluate the consumption impact the security checks are imposing on the device. In order to have a clear answer on whether our implementation is indeed lightweight we expect a negligible increase in the percentages of CPU and RAM consumption. We define the following research questions to be answered as a verification of the above statement.

- Will our security checks have a negative impact on CPU consumption in an application bearing them?
- Will our security checks consume a considerable amount of the available RAM of the device?
- Will a high frequency of running the checks have a greater impact on CPU/RAM consumption?
- Is there a significant network stress caused by the security checks?

Devices

We chose devices that are commonly used and that in their majority have different Android API one from another. The list of the devices chosen is:

- Pixel 2, API Level 28
- OnePlus 5, API Level 26
- Huawei Mate 9, API Level 24
- Xperia XZ1 Compact, API Level 26
- Nokia 1 (Android GO), API Level 27

An important notice is the device Nokia 1 (Android GO). We chose that device due to it belongs to the low-end spectrum of Android devices. We considered it would be an interesting scenario as well to see the impact of our implementation in the low-end devices where RAM and CPU power are limited.

What we measure

We are interested in CPU, RAM and Network load. We measure the CPU consumption in percentage of the overall consumption of the device. This means that if the device has available more than one cores, our results is the average between these cores for any given moment. For the RAM we measure the overall consumption of the device in *kilobytes*. Finally the traffic reported to us is the traffic related to the execution of our application. Any data sent or received to and from our app will be reported back to us.

The Application

As a testing application we used a simple calculator app. This application does only basic arithmetic operations, and we chose it due to its simplicity and the low footprint on resource usage. This way it will be clear the impact on the resource consumption that the security checks are imposing on the application. We added the security checks into the application and adjusted them in such a way that we can regulate the interval between two consecutive executions. A new thread is created for the checks to run, as not to stress the main thread of the app. We created five different test *apks* with different intervals between running the security checks. We chose to run the tests with intervals of 0.5, 1, 2, 5 and 10 seconds. Although running a security check every half a second may seem a bit exaggerated we have added it in an attempt to ascertain the results even in an extreme case.

The Tests

The recorded test includes 21 arithmetic operations between numbers, like *5 + 2 = 7*, where overall it lasts approximately 40 seconds, depending on the device it is being run on. The test is recorded using a Roboscript [1] in Google Firebase. Due to the lack of export capabilities for the metrics in the Firebase dashboard we created a script which parses the results after we manually extract them from the dashboard. We evaluate three parameters on resource consumption, the CPU, the RAM and the network traffic. For the network traffic we should note that the security checks are sending a post request with a json containing all the results and receive as an answer a verification from a server on whether the post was processed properly or not.

6.2 Results

Before analyzing further the three metrics described previously, we are going to show the highest and lowest interval along with the results of the tests without any security check, for two distinct devices. The two devices are not picked randomly but they are representative of the behavior of the rest of the devices as well. The complete list of the results containing all the intervals tested for each of the five devices can be found in Appendix A.

The figure 6.1 depicts the results from the application that does not bare any security checks. The CPU values indicate the average of the consumption of all the available CPUs the device has for any given moment. We notice a spike in the beginning in the CPU which is something noticed across all tests of all devices. We believe this is caused from the installation of the application to the device for the testing to begin. The calculations performed during the tests last from the sixth second until a bit after the 30th second. We can see that the CPU consumption clearly does not go over a 10% at all times of this test, the RAM is almost stable at 50 Kilobytes and the network traffic is non existent as our app involves no security checking and thus there is no need to send or receive anything.

If we compare with figure 6.2 and later on with figure 6.3, we can automatically see that there is no significant increase in RAM and CPU usage. This is a good initial sign as it means that even at the rate of two security checks per second the resource usage of RAM and CPU is remaining at minimum levels. What stands out immediately is the traffic generated by the security checks due to the sending and receiving data to and from the server.

[1] https://firebase.google.com/

Figure 6.1 clean App - Pixel 2

Figure 6.2 10 seconds - Pixel 2

Figure 6.3 0.5 seconds - Pixel 2

From the five devices we chose to conduct the experiment all four devices except the Nokia 1 belong to mid or high end range. For these four devices the results are similar to the ones we saw for Pixel 2,

they do not seem to be affected neither in CPU nor in RAM by running even as much as two security checks every second. As a first glance this is exceptional for our case as our implementations are proven light weight.

We see the results for the device Nokia1, for the clean version without any security check and the high and low intervals at figures 6.4, 6.5 and 6.6 respectively. Nokia1 belongs to the low end devices that is why we see it reaching even over 20% CPU consumption when running the clean Apk. We notice how much of a difference there is in the CPU consumption between the clean version and the two versions running the security checks. As a second observation we see that the RAM consumption is not affected as much as CPU and that it remains almost stable in the two versions with the security checks. The traffic from the network has very similar behavior to the Pixel2 device, which is something expected. Lastly one important notice is the fact that the test took more than 85 seconds to complete in this device in comparison to the 34 seconds it took for the Pixel2.

Figure 6.4 clean App - Nokia 1

Figure 6.5 10 seconds - Nokia 1

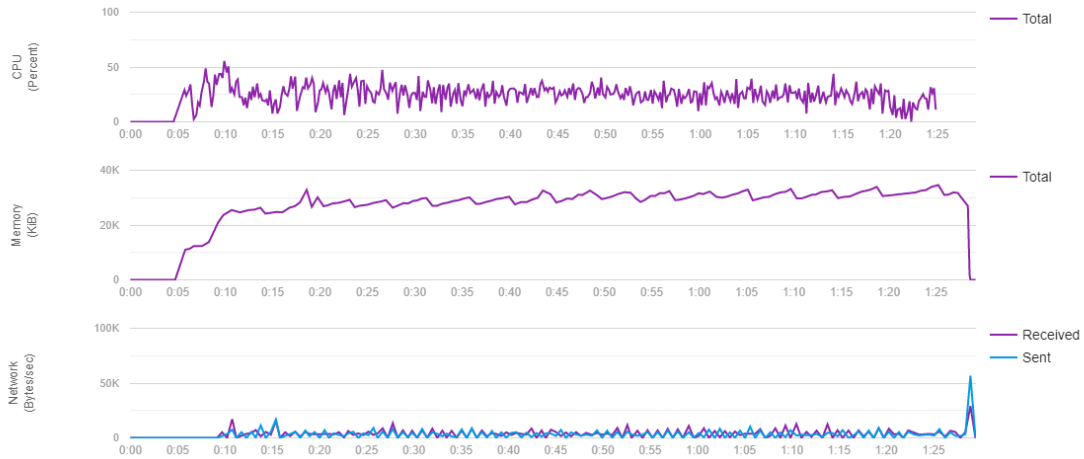

Figure 6.6 0.5 seconds - Nokia 1

In the following sections we are going to analyze in depth the results for each metric seperately and explain our discoveries.

6.2.1 CPU Consumption

In figure 6.7 we can see the average of the CPU consumption in percentage, for the five different intervals of running the security checks. On the axis-X the first values we can see refer to *"clean"*, this is the application without any security checks, as a measure of comparison.

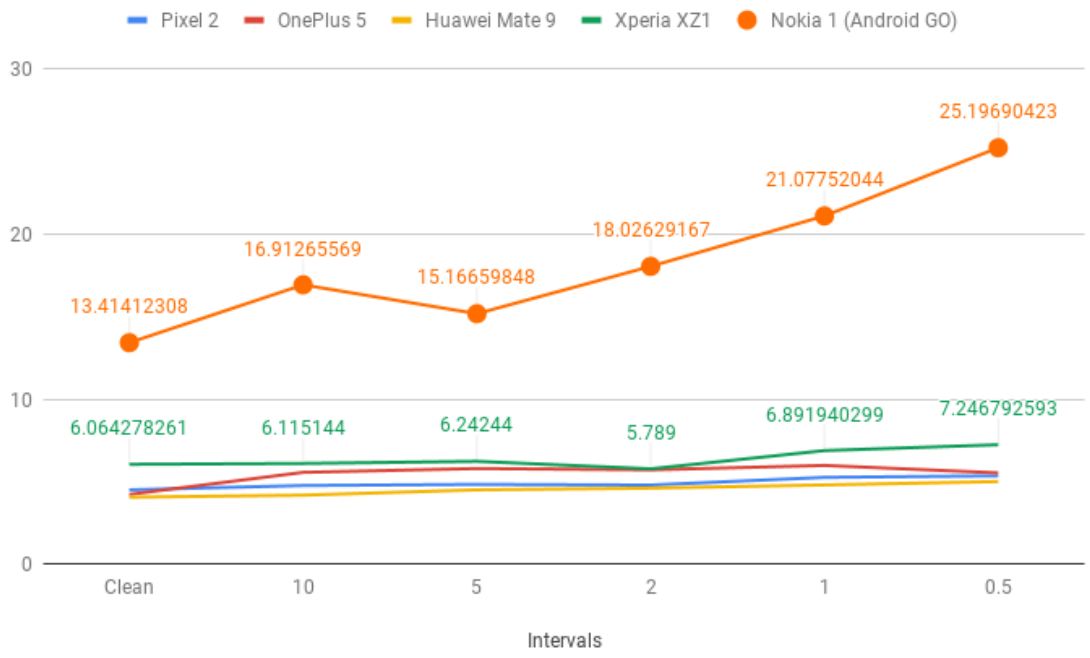

Figure 6.7 Average CPU Consumption Per Device Per Test

We can immediately notice that for all devices except Nokia 1(Android GO), the resulting resource consumption the security checks added seems insignificant. On the one end of the chart we see the app without any security checks while on the other the security checks are running twice for every second. Upon comparing the average consumption for each device between these two mentioned cases as shown in table 6.1, we see that besides the Nokia 1 device which is a low-end device the rest of the devices report an increased consumption of at most 1.3%. This translates that running the security checks

twice every second which is an exaggeration, will have a really light impact of the CPU consumption of a mid or high-end device. For the Nokia 1 device, we see that it reaches almost 12% increase when the interval of the checks is the minimum. Yet it still has an increase of a reasonable 2.4% when comparing the clean apk with the interval of the 5 seconds.

Devices	CPU Dif
Pixel 2	0.8500851911
OnePlus 5	1.29938228
Huawei Mate	0.9465011186
Xperia XZ1	1.182514332
Nokia 1	11.78278115

Table 6.1 Average CPU Maximum differences per Device

6.2.2 RAM Consumption

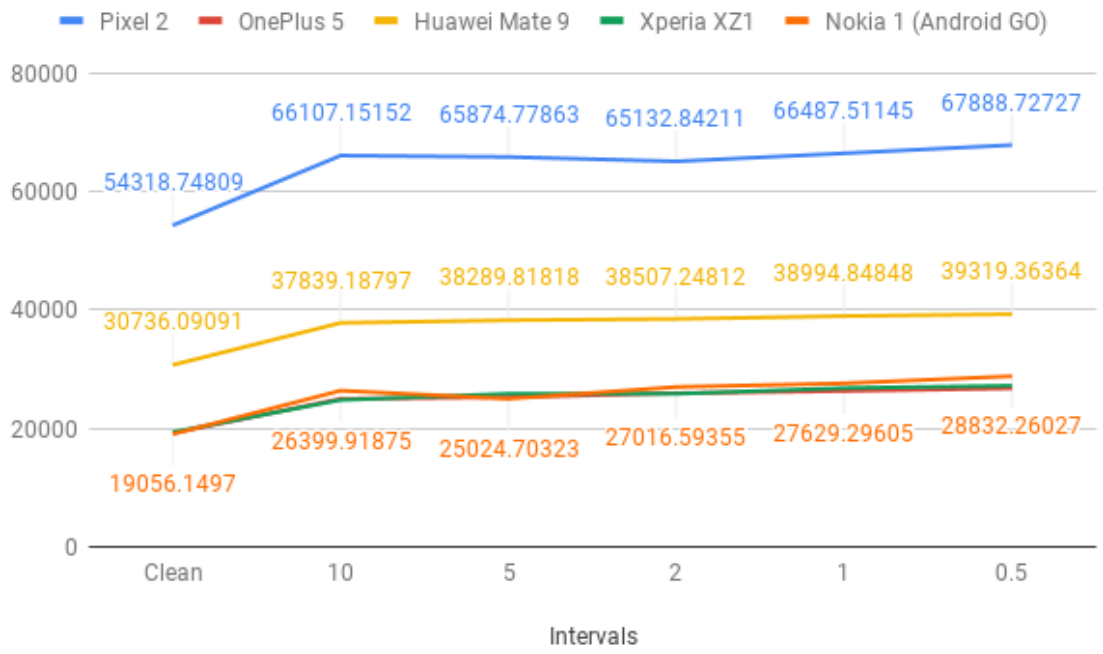

Figure 6.8 Average Ram Consumption Per Device Per Test

In the figure 6.8 we can see the RAM consumption measured in *kilobytes*. Again first is the apk without any security checks and then the rest of the intervals. What we notice at first glance of this chart is that the increase in the required RAM by the application is not dependant on the frequency the security checks are being executed. This is a notable observation as we can roughly estimate the amount of RAM that is required for the security checks to be executed overall. The average of the maximum differences of RAM consumption for all devices is less than 10 megabytes. We see that the footprint of the security checks is relatively lightweight, even for the Nokia 1 device, where its RAM is 1GB. There is a slight increase in the gradient in the measurements when the frequency of the checks increase. The biggest difference is on Pixel 2 and it reaches 13.5 Mb which again does not seem to be a considerable amount towards the rest.

6.2.3 Traffic

Following we present two figures showing the the traffic monitored in the devices throughout the tests. There is a chart for the traffic received and one for the traffic sent. The charts are showing the sum of

the traffic for each device for each interval starting with the apk without any protection where naturally we observe that there is no traffic there. We would expect the traffic amongst the different devices for the same interval to be approximately equal, since the security checks are sending or receiving similar data in each case. Though we observe not such thing, and on the contrary the gap between some devices like Nokia 1 and Xperia XZ1 is vast. This is justified by how the Robo test is being executed. In the Robo test we have specified 21 different calculations to be executed, and this results in different time of execution based on the capabilities of the device.

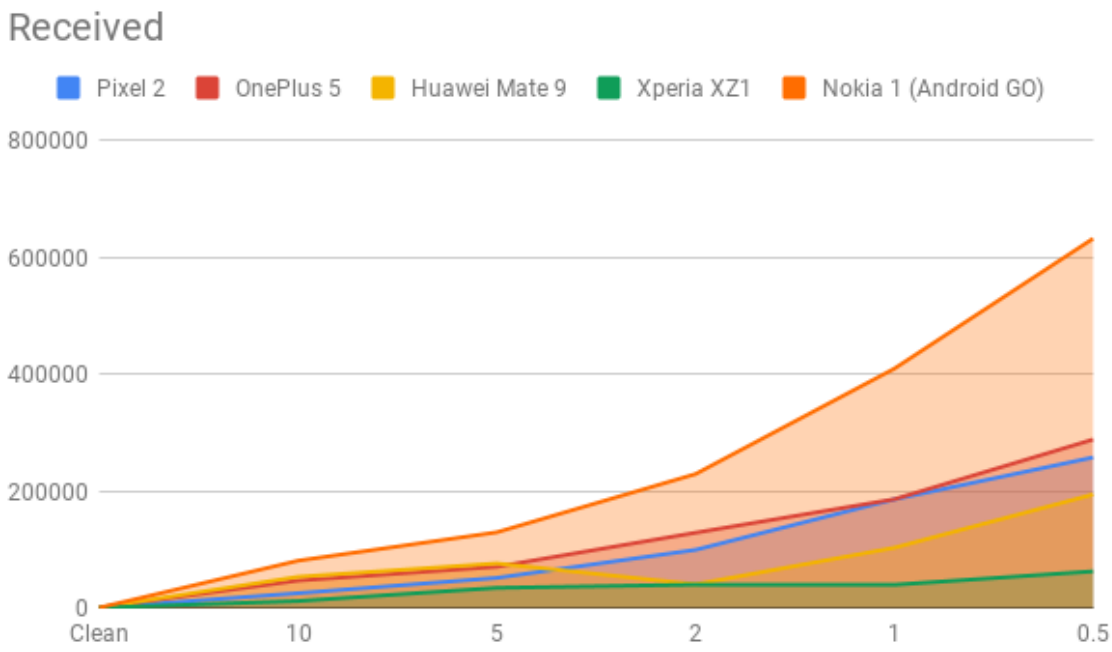

Figure 6.9 Sum of Received Traffic Per Device Per Test

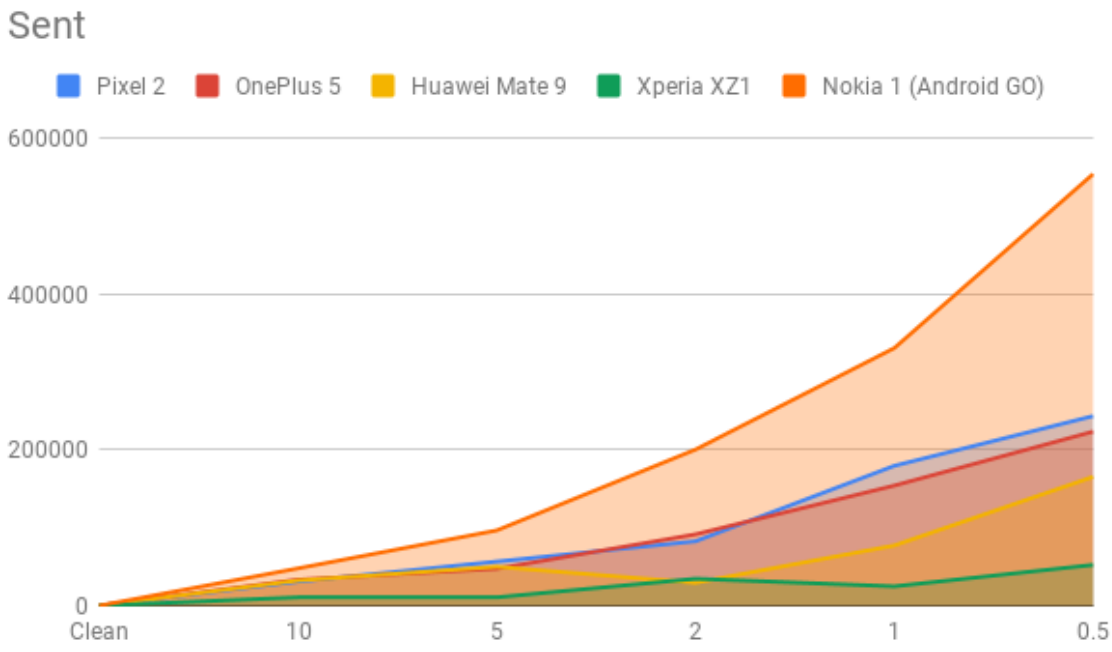

Figure 6.10 Sum of Sent Traffic Per Device Per Test

The Nokia 1 being a low-end device takes the most of the time to get the test completed, approximately 80 seconds, while the Xperia XZ1 required less than 30 seconds. Due to the security checks being implemented in these tests to run on steady intervals many more security checks eventually are executed on the Nokia 1 device and thus result in higher amount of traffic generated. The same logic applies to the other devices as well, the difference in the time required to finish the testing is translated in more traffic being sent or received from and to the devices.

In order to avoid any misconception based on the issue above, we decided to include one more test. This time we adjusted the app in order for the security checks to be executed upon clicking on the *equal sign (=)* in each of the 21 operations the Robo script is running. This way independently of how much time it takes for the device to complete the calculations the same amount of traffic should be reported in the measurements. Indeed this is the result as it can be seen also in the table 6.2. All the devices report approximately the same number of bytes being received/sent during the test run.

Device	Received (B)	Sent (B)
Pixel 2	17244	41624
OnePlus 5	17372	40455
Huawei Mate	17299	39354
Xperia XZ1	17482	40170
Nokia 1	18416	41710

Table 6.2 Sum of traffic for the same number of security checks

6.3 Answers to the research questions

> 1 - Will our security checks have a negative impact on CPU consumption in an application bearing them?

We saw that when the security checks are executed twice for every second, which would results in the highest load they could produce, the mid and high-end devices reported an average of less than 1% increase in their CPU load. For the low-end device, Nokia 1, although the case where running two checks every second leads to an increase of 12%, an acceptable for a developer execution of one security check every 5 seconds leads to only 2.4% increase in the CPU load. Based on the above we can safely assume that our security implementations are light-weight on most devices and will not cause any negative impact CPU-wise to any application bearing them.

> 2 - Will our security checks consume a considerable amount of the available RAM of the device?

We see that the biggest difference is noted on the device Pixel 2, where the RAM consumption reached 13.5Mb. For a device having 4GB of Ram, we understand that the consumption reported due to our security checks is negligible. The low-end device Nokia 1, with only 1GB of RAM reports less than 10MB of RAM consumed, and yet again the number seems negligible in the overall performance. We consider our implementations to have a very low footprint on RAM consumption.

> 3 - Will a high frequency of running the checks have a greater impact on CPU/RAM consumption?

In figure 6.7 we clearly see that a higher frequency in the security checks leads to an increased CPU load. This impacts mostly the low-end device, Nokia 1, where the higher the frequency the more increased load we detect. For the rest of the devices the difference in the CPU load as we move to higher frequency is not affecting them as much, reporting approximately 1% in all of them. The RAM consumption appears to be irrelevant to the frequency of the security checks, maintaining an approximate of 10Mb for all test cases.

We deduct that our implementation can safely be considered light-weight in even as much as two execution per second in mid and high-end devices. For low-end devices a five second interval also proves to have a very light impact on CPU and RAM.

4 - Is there a significant network stress caused by the security checks?

As we see in table 6.2, at most, 40kB of data are being sent and almost 20kB received. Taking under consideration that for 2018 the average download speed worldwide is 21.35 Mbps and the average upload speed is 8.73 Mbps [32] we see that the time taken to send all of the results of the 21 checks is almost zero. Based on that we understand how negligible is the amount of network load caused due to a single check. We verify therefore that the network stress caused by the security checks is insignificant.

Chapter Seven

Empirical Validation of Detection Capabilities

In this chapter, we test the validity and accuracy of our implementations against known attacks. We will see how the security check for reporting the signature of an apk goes against attempting to bypass it, we will also test to see how the security checks handle the Magisk Hide capability of Magisk Manager and in general how the root detection mechanism behaves in different scenarios. We will attempt to run Xposed Framework and Frida against a protected apk, and we will report the time and persistence required in order to bypass the protections. Throughout this section when we refer to signatures we imply the SHA256 hashed version of them.

Research Questions

In our attempt to verify the accuracy of our proposed approaches we define the following research questions.

- How easy is for an attacker to bypass our signature checking implementation by spoofing the signature?
- Is there any emulator environment from the ones available to us that is capable of evading detection?
- Is there any device emulated or not, that has root access, capable of evading the detection of the fact that it is tampered (excluding the checks for emulation) ?
- Do we have any false positives? Meaning, is there any not tampered device that any check on that device reports something suspicious?
- How easy it appears to be the fact to instrument the code of the app in order to bypass the checks?

Metrics

Our metrics for verifying the results on the above questions are split into two categories. The first one being a direct recognition of whether our result is contained in the report that the device has sent to the server. For example, if we are checking to verify the results of emulation, the 10-digit binary report from the device would reveal what we need to know. The second category, refers about how we define the term "how easy" something is. This part we measure it in relation to the skill and time it would require for someone to achieve the given goal. An example is, if we need to define how easy is to bypass the signature checking, then we can first see if already known tools are able to bypass it, if some custom work is needed then what level of knowledge does it require and how much time.

7.1 Signature Spoofing Validation

In chapter 4.1 we mentioned that the check tool sends a report back to the server which contains the results from all the checks made. For this particular testing we have adjusted that report to also

contain besides the signature our implementation derives, the signature from the package manager's *GET_SIGNATURES* which is mentioned in chapter 3. This way we would know what our method retrieves but also we can compare with the suggested way to retrieve the signature that Android offers.

We will run two tests, the first includes signature spoofing which is supported by the OS used in the device and the second a specially crafted application as described in Chapter 3.1.1. For the device specific scenario we use a device *Xiaomi Redmi Plus 5* running *HavocOS* which supports signature spoofing. We create a dummy application which utilizes the permission *android:name="android.permission .FAKE_PACKAGE_SIGNATURE"/>* and reports a fake signature every time it is requested. We add our security checks in the application and run it.

The real hashed signature of the original app is

92bc5c274032f9b3b62b8a141b3aeda6c044e9235a706a615661e2f25f08354c

and the hashed spoofed signature we have set is

aa5c0802caede68509a1fea433b50ffa3a534771966c20795133e7c66df50711

As mentioned before we have adjusted the reporting back to the server to fetch the signatures with both methods. Here are the results:

Our implementation yields

92bc5c274032f9b3b62b8a141b3aeda6c044e9235a706a615661e2f25f08354c

while the package manager's *GET_SIGNATURES* feature yields

aa5c0802caede68509a1fea433b50ffa3a534771966c20795133e7c66df50711

We can see that although the *GET_SIGNATURES* method was tricked and returned the spoofed signature our implementation bypassed the spoofing and reported the actual signature of the app.

A similar test was conducted for the application specific spoofing scenario, using the ApkSignatureKiller [1] tool which directly hooks the *getPackageInfo* method of the *PackageManager* and returns a pre-specified signature every time it is requested. In this tool we need to supply the apk that we wish to attack and the apk from which we need to copy the signature, then the tool automatically patches the apk and outputs the attacked apk. In our tests we use this "attacked" apk and we run it against the set of five devices we used in the chapter of the Performance Monitoring.

We expect that since the tool is making changes to the application and eventually resigns it with a prespecified key that, the signature will be different than the original one of the untouched apk which is the

96a35810c2426dfcfdd6e5f421371ace386339b575da6b12db01222e0dd17222

This is due to how the signing process in Android works as we described in 2.1.1.

All the devices report back the same result which is, for our implementation

2c61800cd343d4aecf3a42adaf711f917ede4d73197f1606a133b8b5fcd003e2

and for the package manager's *GET_SIGNATURES*

96a35810c2426dfcfdd6e5f421371ace386339b575da6b12db01222e0dd17222

We can see once again that our method prevailed against this attack and managed to report the actual signature of the "attacked" apk, rather than the one of the untouched apk. The code used to extract the signature with the package manager's *GET_SIGNATURES* is shown in the listing 7.1.

[1] https://github.com/L-JINBIN/ApkSignatureKiller

```
private String getPublicKey() {
    StringBuffer result = new StringBuffer();
    try {
        PackageInfo packageInfo = context.getPackageManager()
                .getPackageInfo(context.getPackageName(),
                        PackageManager.GET_SIGNATURES);
        for (Signature signature : packageInfo.signatures) {
            byte[] signatureBytes = signature.toByteArray();
            InputStream certStream = new ByteArrayInputStream(signatureBytes);
                CertificateFactory certFactory = null;
            try {
                certFactory = CertificateFactory.getInstance("X509");
            } catch (CertificateException e) {
                e.printStackTrace();
            }
            Certificate x509Cert = certFactory.generateCertificate(certStream);
            byte[] encodedKey = x509Cert.getPublicKey().getEncoded();
            MessageDigest md = MessageDigest.getInstance("SHA256");
            md.update(encodedKey);
            final String digest = bytesToString(md.digest());
            result.append(digest);
        }
    }
    catch (CertificateException | NoSuchAlgorithmException |
    PackageManager.NameNotFoundException e) {
        return e.getClass().getName() + ": " + e.getMessage();
    }
    return result.toString();
}
```

Listing 7.1 Java Code to retrieve public key from Signature

7.2 Device Profiling

Throughout our detection validation the server part of our architecture was logging all the reports made from the devices running the security checks. This server besides helping us gather the responses from sources out of our reach, like Firebase also contributed in gathering details about each device as mentioned in 5.2.7. A list of devices was formed this way, that can help identifying instantly for example the devices coming from Firebase and thus allow a higher sense of trust against them.

The table 7.1 contains the collection of the device profiles as they appear in the server side. In the first three columns we see identifying details about the device. In the first column there is the device "code-name", in the second the manufacturer of the device and in the third the product name. In the fourth one we see a rate of successful check reports over the overall ones found in our database. The fourth column provides us with two details. The first is that it informs us how many times we have seen this device in our database. This number can be used as an approximate estimation of the popularity of the device amongst the users using the application. Secondly, based on the fourth column we know the success rate a particular device has. Overall, this list is capable of providing us with information about the popularity of the device, a high level identification of it and details about an overall of the previous reports found. This means that any security checks run previously on such devices are evaluated on server side and the list of devices is automatically updated to include any new reports.

We see that most of the Firebase physical devices since they are untampered, they have an excellent rate of trust while the devices that run on emulators and/or they have root access they are detected as such and they fail passing the security check. We see some exceptions in the Firebase physical devices, like the Huawei device which reports that its SELinux property is either permissive or disabled. This happens also for the Sony device G8142 while the Sony G8441 correctly reports as untampered. Due to the fact though that we have no access to the devices that report SELinux altered, we can not test further whether this is a false positive.

Device	Manufacturer	Product	Trust Ratio
walleye	Google	walleye	12/12|100.0%
FRT	HMD Global	Frontier_00WW	15/15|100.0%
HWMHA	HUAWEI	MHA-L29	0/12|0.0%
mlv1	LG Electronics	mlv1_global_com	0/13|0.0%
lucye	LGE	lucye_nao_us	0/12|0.0%
A0001	ONEPLUS	bacon	0/12|0.0%
OnePlus5	OnePlus	OnePlus5	12/12|100.0%
G8142	Sony	G8142	0/12|0.0%
G8441	Sony	G8441	12/12|100.0%
vince	Xiaomi	vince	0/2|0.0%
potter	motorola	potter	0/12|0.0%
a5y17lte	samsung	a5y17ltexx	12/12|100.0%
hero2lte	samsung	hero2ltexx	12/12|100.0%
j1acevelte	samsung	j1acevelteub	0/15|0.0%
j7xelte	samsung	j7xelteub	12/12|100.0%
star2qlteue	samsung	star2qlteue	12/12|100.0%
gce_x86	unknown	gce_x86_phone	0/24|0.0%
gce_x86	unknown	gce_x86_phone	0/48|0.0%
gce_x86	unknown	gce_x86_phone	0/57|0.0%
gce_x86	unknown	gce_x86_phone	0/37|0.0%
gce_x86	unknown	gce_x86_phone	0/12|0.0%
gce_x86	unknown	gce_x86_phone	0/12|0.0%

Table 7.1 Device Profiling List

7.3 Environment Reconnaissance Validation

For the validation of this part we will explain in what form the server receives the security report for the environment reconnaissance. All the methods implemented in chapters 5.2 and 5.3 besides the Device Profiling are returned to the server in a binary form. Meaning that if the method was successful in finding something it returns "1", otherwise "0". The result is a string of ten digits, as shown in figure 7.1.

We plan to run the calculator application that was used also in the chapter 6 and run it into different environments. For every test we will explain the structure of the environment and then we will analyze the results of the security checks.

7.3.1 Simple Case Scenario

For the following listed tests, we use the same devices where each one has specific configurations and in order to avoid referring to them separately in every different testing we are listing them below along with their specifications. The devices can be seen in table 7.2.

- Emulator Detection
- Proper Keys Detection
- Wrong Permissions
- Unusual Android Properties
- Selinux
- Root Managers & Binaries
- Packages Detection

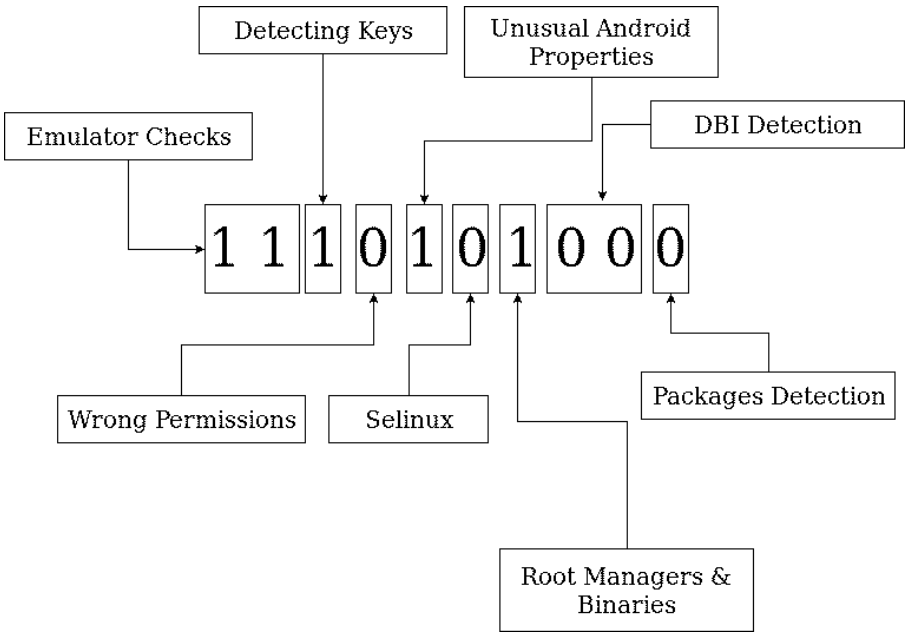

Figure 7.1 Reconnaissance Server Report

Source	Device	API	Specifications	Security Report
AVD	Pixel 2	28	-	1100000000
AVD	Nexus 6	28	Root	1110101000
AVD	Nexus S	25	Root	1110111000
Genymotion	Samsung S8	24	Root, Xposed	1110111110
Genymotion	Google Nexus 9	24	Root, Frida	1110111010
Genymotion	Google Nexus 10	21	Root	1110111000
Bluestacks	Google Pixel XL	22	-	1001010000
Firebase	Huawei Mate 9	24	-	0000010000
Firebase	All Physical Devices	26-28	-	0000000000
Author	Xiaomi Redmi Plus 5	28	Root	0001011001

Table 7.2 Devices used for Detection Validation

Emulator Detection

We started by testing against known emulator engines and more specifically we used the AVD manager of Android Studio, the virtual devices offered by Firebase, Genymotion and Bluestacks. The tests include gathering the results using different devices from each platform and also different Android versions. We evaluate only the first two digits from the reported result of each security check, since only these are related to directly detecting emulators. The results are presented on the table 7.3 and as you can see our implementation was able to detect all four emulator environments it was tested against and distinguish the Firebase (P) which refers to the physical devices.

Furthermore, in table 7.2 we verify that also for the device Xiaomi which is a physical device as well the report showed that it was recognised properly as such.

Proper Keys Detection

For our next evaluation, which is the "Detecting Keys", we use the fingerprints of the devices. Fingerprints are strings that identify the device and the current build, which we also mention and you can see as an example in table 5.5. We gathered the fingerprints from all the tests we have with the four emulators plus the available physical devices. Whether the device is signed with development keys or test keys or actual release keys is mentioned in the fingerprint. The results confirmed that the AVD devices with root access and the Genymotion environments are not signed with release keys

Emulator	Detection
AVD	11
Firebase (V)	11
Firebase (P)	00
Genymotion	11
Bluestacks	10

Table 7.3 Emulator Detection Testing

and that only Bluestacks , the AVD device with Play Store support and the physical devices provided either from Firebase or from the author of this thesis, were signed properly. We can see that by this check we can immediately distinguish the Genymotion and AVD devices. The Bluestacks reports back release-keys for the build it is using and thus manages to not trigger this alarm.

Wrong Permissions

The "Wrong Permissions" indicator in the report is the fourth digit, and consulting the table 7.2 we see that it appears tampered in the Bluestacks and on the Xiaomi device. In order to verify that the results are indeed correct we manually checked the permissions for the paths specified in the table 5.1 for all of the devices. Although in the Bluestacks device we did not gain root access and we neither modify it in some way it appears to have the */system* directory as writable and it was identified as suspicious. For the rest of the applications that root access is attended the checked paths are not writable. We see here that although the Bluestacks evaded the previous check with the "Proper Keys" it now got detected due to the permissions applied in its */system* directory.

Unusual Android Properties

If we take a look at the section 5.2.3 for the list of suspicious android properties the fifth bit is informing us about, we will see that it refers to the *ro.debuggable* flag and the possibility to run *adb*[2] as root. In our devices the Pixel 2, the Firebase devices, the Bluestacks and the Xiaomi device all are devices signed with release-keys as we mentioned previously and as so they have the *ro.debuggable* flag set to "0" and for the rest except the Xiaomi device we do not have root access. The Xiaomi device manages to bypass this check although there is root, due to the fact that, the root access through the adb, is managed by the internal magisk manager and therefore there is no reason to set any android property for it. For the Genymotion devices and the two Nexus from AVD, we expect them to raise an alarm on this security check as they are devices used during development and all the flags are set to allow debugging and root access so it is convenient for the developer to debug applications. Again we see that our checks manage to detect precisely which devices are considered "fertile" to malicous reverse engineering.

SELinux

The SELinux security check is expected to be triggered in devices where root access is granted. It is required to have root access to the device in order to change the SELinux status, but having root privileges does not automatically mean that SELinux will be in permissive mode or disabled. For our results in order to verify the state of SELinux we connected through *adb* and run the command *getenforce* which returns the current state of SELinux. For the remote devices of Firebase, we know that are physical unmodified devices, set for release and thus the SELinux is enabled by default.

We see that the Nexus S device, all the Genymotion devices, the Bluestacks, the Xiaomi device and also mysteriously the Huawei from Firebase, all have the bit of SELinux flipped to "1". More specifically for the Nexus S we have set the SELinux to run permissive as it was serving our testing purposes and the Genymotion devices are by default running with SELinux disabled. For the Bluestacks

[2]Android Debug Bridge

device we verified that indeed SELinux is set to permissive by default. What surprised us is the result for the Xiaomi device as upon installing the ROM in use and using Magisk to gain root access, the SELinux status was tested and it reported to be enforcing. After some investigation we found the culprit, apparently the *ro.build.selinux* property was set to "0" although the status of SELinux was indeed enforcing. This means that the status of SELinux was changed to enforcing after that ROM was installed in the device. This is something that we should take under consideration when working with Android Properties as their value might not always represent the actual status of the device. Finally for the Huawei device, since it uses the default configuration we do not expect it to have SELinux disabled, though due to we have no access to the device no further experimentation is possible in order to verify if this is a false positive indeed.

Root Managers & Binaries

Our next test involves the results for "Root managers & Binaries". As we already have explained in chapter 5.2.5, we are employing several different tactics into detecting the existence of root access directly. At first glance the results are excellent as in all the cases where root is present the seventh bit is flipped to "1". In order to have more detail we run the tests using a logging system which is in place to allow us to gain more information on which exactly checks were triggered. For the two Nexus devices of AVD the superuser binary got detected, for the Genymotion devices we have multiple detections of the Busybox and the Superuser binary but also the check to run the command "which su" was successful. Finally for the Xiaomi device the superuser binary and the command "which su" were both successful, along with the detection of specific Magisk files, which in combination with the last flipped bit, it also found the magisk manager installed and other root specific applications.

7.3.2 Advanced Case Scenario

In the advanced scenario we attempt to run the security checks on environments where we attempt to conceal the presence of root or other properties of the system. We do this by using the almighty Magisk Hide along with randomizing the Magisk Manager app name. There was an attempt to use some Xposed Modules for this task and also some applications claiming that hide root but from our experience they did not manage to hide even the basic things. We compare the results from Magisk hide with the ones we have from the simple scenario and elaborate as to why they have changed.

Hide	Detection
Off	0001011001
On	0000011000

Table 7.4 Magisk Hide Results

In table 7.4, we notice that two of the properties are changing. The bit that corresponds to the "SELinux" detection as we mentioned earlier is false due to reasons related to the ROM in use, so we are ignoring it. We notice that the bit corresponding to the "Wrong Permissions" and the bit related to the "Packages Detection" are not flipped and showing that nothing suspicious is found. The reader may find it interesting to read the part of the Magisk Hide code responsible for flipping the "Wrong Permissions" bit, found here[3]. The bit regarding the detection of suspicious packages was flipped to "0" due to the Magisk Manager‘s app name is now randomized in the advanced scenario and thus unable to be detected. The bit capable of detecting still the existence of root is the one related to "Root managers & Binaries". We mentioned in the beginning of this paragraph that we see two properties changing after we have Magisk Hide "on". This is not entirely correct and we will explain why. Actually three properties do change but due to the fact that the seventh bit has multiple checks behind it, we can not see that, in the advanced scenario only a few of the checks manage to find the existence of root in contrary to when Magisk Hide is off where many checks report that root is present. Further for the superuser binary detection and the result from running the command "which su", that

[3]https://github.com/topjohnwu/Magisk/blob/master/native/jni/magiskhide/hide_policy.cpp

we know previously revealed the existence of root in the device, now the result comes back as "0". This happens as the Magisk unmounts */sbin* and the other bind mounts in order to hide itself [39], thus the superuser binary now vanishes and therefore the "which su" command shows no result. We see, that although Magisk Hide manages to bypass some of the general checks we deploy in the app, it is not capable bypassing the specific checks we make for it in the "Root managers & Binaries".

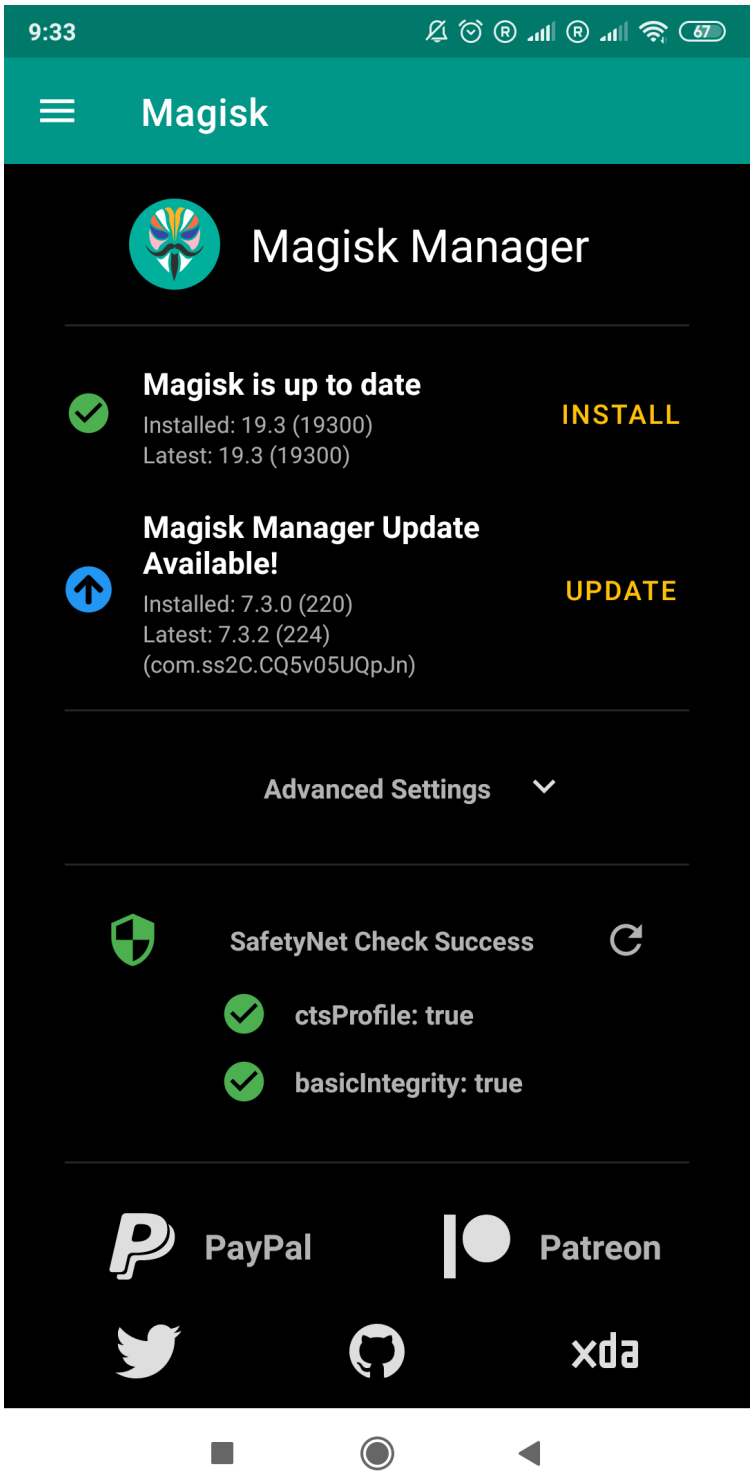

Figure 7.2 Magisk Manager bypassing SafetyNet

As a final step we compare our proposed approach with the Google's service, SafetyNet[4] against the Magisk Hide. SafetyNet is used widely from developers who wish to ensure their application is running

[4]https://developer.android.com/training/safetynet

only on safe environments. We used Magisk's protective measures against SafetyNet and the result can be seen in figure 7.2. We can see that SafetyNet is successfully bypassed reporting "ctsProfileMatch": true and "basicIntegrity": true. This means that any application uses SafetyNet to verify the safety of the execution environment of a device will have false negative results against devices gaining root using Magisk. It is a common practice for users who wish to bypass SafetyNet protections to resort to Magisk [24]. Many gaming and banking apps who use Magisk, like ING banking app, has been reported to be trivial to bypass [13] due to the majority of the checks are based on SafetyNet and the rest that are not, are already covered by the hiding mechanisms of Magisk.

7.4 DBI Detection

Simple Scenario

As stated in table 7.2 both devices that have Xposed Framework and Frida got detected. The Samsung S8 was picked up with the Xposed framework for having specific files that we attempt to identify and also due to a loaded library related to the framework. In order to verify the result we are manually running the command *cat /proc/self/maps* where *self* is the *pid* of the app, after we connect through *adb* with the device. The result is presented bellow:

```
vbox86p:/ # cat /proc/1590/maps | grep pose
5fd60000-5fd6f000 r-xp 00000000 08:06 1279 /system/bin/app\_process32\_xposed
5fd6f000-5fd71000 r--p 0000e000 08:06 1279 /system/bin/app\_process32\_xposed
f174c000-f174d000 r--s 00019000 08:06 1122 /system/framework/XposedBridge.jar
f1762000-f176c000 r-xp 00000000 08:06 2203 /system/lib/libxposed_art.so
f176d000-f176e000 r--p 0000a000 08:06 2203 /system/lib/libxposed_art.so
f176e000-f176f000 rw-p 0000b000 08:06 2203 /system/lib/libxposed_art.so
```

Listing 7.2 Loaded Xposed Libraries

For the consistency of the document we have omitted the majority of the output, yet we clearly identify three different entities we can have as indicator for the presence of the Framework.

The Google Nexus 9 is the device where the *frida-server* is available. We see that the file itself was found and triggered the detection but no library was found loaded. This is what we were expecting after all, as we have not hooked the application so the *frida-agent* to be found loaded.

Using Frida we inject a harmless code into the calculator app we have been using, in order to see whether the the Frida library will be detected or not.

Before	01
After	11

Table 7.5 Frida library Detection

As we can see in table 7.5 where the first bit refers to the library detection while the second one to the file of the server being found, the library was detected after we hooked into the application. As we can see in the listing 7.4 the *frida-agent* library is loaded, and it is a clear indication that Frida is interfering with our application.

```
vbox86p:/ # cat /proc/1521/maps | grep frida
d68c7000-d6a4a000 r-xp 00000000 08:13 627094 /data/local/tmp/re.frida.server/frida-agent-32.so
d6a4a000-d6a4b000 rwxp 00183000 08:13 627094 /data/local/tmp/re.frida.server/frida-agent-32.so
d6a4b000-d7d5c000 r-xp 00184000 08:13 627094 /data/local/tmp/re.frida.server/frida-agent-32.so
d7d5d000-d7da4000 r--p 01495000 08:13 627094 /data/local/tmp/re.frida.server/frida-agent-32.so
d7da4000-d7dec000 rw-p 014dc000 08:13 627094 /data/local/tmp/re.frida.server/frida-agent-32.so
```

Listing 7.3 Loaded Frida Libraries

Advanced Scenario

To our knowledge and research there is not an available way to hide the fact that a device is using the Xposed Framework and additionally it is a project relatively outdated thus we will not test further into it. We consider that the methods we propose for identifying Xposed Framework are adequate.

We continue an attempt to bypass the detection using Frida. There are multiple ways in achieving this, especially given the fact that we already know in depth exactly what each implementation is checking. These attempts will only focus on hiding Frida itself from the detection and not bypassing all the security checks.

As a first step we are moving the *frida-server* to a different not common location and after rename it we execute it from there. For a fact we know that it will not be found by our checks in this case for two reasons. One, since we renamed the file, it is not possible to be predicted by the check what the new name is and two, since we execute the frida server as root, we can place this file at a location where a simple app will not have access. As expected the result we get in the report shown at table 7.5, we see that Frida is not detected at least before we any of its libraries are loaded in the app.

But still if we try to load the frida libraries in order to inject any code, the application will detect the library. So our second step is to bypass that as well. We know that the check we implemented utilizes *fopen* and *fclose*, so the obvious solution is to use frida to hook *fopen* and make it return *null* every time, in order for the check of the loaded libraries to fail. Though such an attempt would be easily detected by using *fopen* to read a file that we already know it should exist and have some content. If that file returned *null* we would be suspicious of the environment and thus we could stop the execution and alert that some form of injection is in place. We see the results on table 7.6, where before moving the server file and attempt to hide the library both bits are set to "1", while after we change the location of the server file it no longer get detected. The loaded library or a tampering to the *fopen* is detected during our tests at all times.

Before	11
After	10

Table 7.6 Frida library Detection

One idea we thought in order to bypass the check for the loaded library is, if we use a feature of the Frida Framework in order to hook "early" the function that does the check for the loaded libraries. By hooking "early" we mean to be able to inject code to the application before the app starts. This resulted in the application crashing although we have not yet implemented any feature against this kind of attacks.
We see that the aforementioned Dynamic Binary Instrumentation security checks prove to be hard to bypass yet reliable for their results. It requires an attacker who is familiar with dynamic binary instrumentation frameworks and enough persistence to do static analysis on the C library code where the checks for the DBI are.

7.5 Answers to the research questions

> 1 - How easy is for an attacker to bypass our signature checking implementation by spoofing the signature?

We showed that in order for an attacker to bypass the proposed method for retrieving the signature of the apk, it would require manual work and a proper understanding of the Android signature scheme. I would also require reversing the C library in which our implementation resides, something that on its own is considered moderately difficult, especially when obfuscation is applied. Therefore we consider the proposed method to require a skilled attacker and a moderate amount of time.

> 2 - Is there any emulator environment from the ones available to us that is capable of evading detection?

Based on our results, it is evident that there is no emulation engine from the ones tested that could evade detection of the fact that the environment is an emulated Android device. We consider a complete success the detection of emulators.

3 - Is there any device emulated or not, that has root access, capable of evading the detection of the fact that it is tampered (excluding the checks for emulation) ?

None of the devices, even upon applying techniques like Magisk Hide, was able to completely hide the fact that the device was rooted. All of the devices that were evaluated based on the security checks, except emulation and DBI detection, were successfully detected from our proposed methods.

4 - Do we have any false positives? Meaning, is there any not tampered device that any check on that device reports something suspicious?

We show two cases where we had unexpected results. Both cases are related to the SELinux security check, the one coming rom the Xiaomi device, where we explained why it occurred and the other from the Huawei device where since it is provided by Firebase we are unable to further examine it. We consider the case from Xiaomi a mis-configuration blamed on the creator of the custom ROM installed in the device and thus not a false positive originating from our implementation. We consider therefore all the test to have an excellent percentage of success without any false positives. Yet, we emphasize that in order to have stronger foundations in that statement a larger set of devices needs to be tested.

5 - How easy it appears to be to instrument the code of the app in order to bypass the checks?

Testing against Xposed Framework and Frida showed that we were able to detect their existence in the system at all times, even when we attempted to hide the use of Frida. We consider it hard to instrument code using the frameworks mentioned, as it would require enough time for static analysis in order to know exactly which function to instrument and also it requires deep knowledge of the frameworks in order to use advanced features that could bypass our checks.

Chapter Eight
Conclusion and Future Work

8.1 Conclusion

Attacking the integrity of an application by repackaging attacks and Dynamic Binary Instrumentation frameworks having root access, is a process that can be done even without the knowledge of an expert. Application tampering prevention is essential in safeguarding the interests of the developer along with the data of the Android user. For this reason we propose a high accuracy, low consumption solution against tampering and malicious reverse engineering.

From our performance validation, we proved that our goal of a light weight security solution that would not cause any cluttering in the device or the application, is achieved with a minimal CPU consumption of 1% and 10 Mb of RAM required regardless of the frequency of the checks. Further, our detection results showed a complete success against devices with or without protections against anti-tampering and anti-debugging mechanisms.

We implemented a list of more than 20 different checks covering a wide range of Android security. We split them into three main categories, identifying tampering of the application itself, active reconnaissance of the environment running the application and detection of dynamic binary instrumentation tools. We implemented novel detection methods and we extended existing ones in order to acquire higher accuracy results. Additionally, where necessary, we showed why some anti-tampering and anti-debugging techniques are failing against state of the art attacks and we reasoned as to why our implementation is able to evade such attacks.

The coverage of multiple aspects in our security checks guarantees to detect a modified environment as at least one of the checks will be triggered. This additionally requires an attacker to go in greater depths in order to bypass all the checks and tamper the application. In conclusion, our proposed approach offers high accuracy results along with low resource consumption and an effortless integration to an Android app.

8.2 Future Work

A new branch of our implementations is now possible to be created, which would increase the anti-tampering and anti-debugging difficulty substantially. We are talking about the dynamic loading of the native library at runtime and the only disadvantage of it is that Google Play Store considers this as a malicious behavior and it is not allowed. Though, for application who seek maximum security and are not interested in the Google Play Store this solution may be ideal. Enabling dynamic loading of the security checks means that what and how we check for tampering and malicious reverse engineering can be updated at any given time without the need to update the app itself. This automatically increases the difficulty of bypassing the security checks as depending on the security level we wish to achieve, we can update the library daily, even by just switching the order the security checks are performed and also adjust the server side to expect the results according to our modifications. This would require by an attacker to restart the process of reverse engineering the app in order to figure the changes we applied.

Extending the concept with the dynamic loading of the library, we can take advantage of the fact that our application utilizes the *Split Component* as explained in chapter 4.1, meaning that some part of the app is located on the server and not the app itself. Forcing this way the app to contact the

server and pass the security test. As we have explained the server already knows what to expect from the application and evaluates what it receives based on specific rules. For example for the "Magic Numbers" integrity test a specific set of rules needs to be followed in order for the server to deem that this number is indeed correct. Imagine now that we update the application's library with a new set of rules for creating this numbers and of course updating accordingly our server. This would force all devices to use the new library on their next tests and any device using the old library automatically would not pass the check as the server would evaluate it to false. This allows us to update the library without the need to update the app itself. It is a major concept as it allows not only to update any of the security checks but also render useless any work any malicious user may have accomplished by that time.

We can see the endless possibilities of this, but we also realize why Google is banning this behavior. The possibility to allow dynamic loading opens a security hole in untrusted publishers who could compromise the security and privacy of all the users using the application. This method strongly requires the user to trust the publisher of the application.

An additional point for future work, is the evaluation of the reliability of each method employed for different manufacturers and devices. This would allow a ranking on the methods as to how many false positives are reported for each. Then, weights could be assigned on each method, in order to provide one solution for all devices which although it could contain false positives it would be evaluated differently with the intention to recognise them and disregard them. Such an endeavour would require an extensive set of devices with manual testing and recording of all findings. We believe that following this idea we could establish a single solution that could adapt to the different devices and provide high accuracy.

Finally, based on the feature of device profiling, upon running an application on a significant number of devices, we would create the corresponding profiles for each device. A future goal therefore would be creating a large database with a high number of devices, which on its own can consist a module of "trust" that can be used in other security application as well. A large database of devices along with the idea of assigning weights in each method, which was described just before, would set an efficient and accurate way to decide upon trusting or not a single device.

REFERENCES

[1] Scott Alexander-Bown. *RootBeer Library*. 2014. URL: https://github.com/scottyab/rootbeer/.

[2] Android. *Debug your app*. 2019. URL: https://developer.android.com/studio/debug.

[3] Android. *Security-Enhanced Linux in Android*. 2019. URL: https://source.android.com/security/selinux.

[4] Google Android. *Application Signing*. 2019. URL: https://source.android.com/security/apksigning.

[5] Google Android. *Permissions overview*. 2019. URL: https://developer.android.com/guide/topics/permissions/overview.

[6] asLody. *Whale*. 2019. URL: https://github.com/asLody/whale.

[7] Ingo Bente et al. "On remote attestation for google chrome os". In: *2012 15th International Conference on Network-Based Information Systems*. IEEE. 2012, pp. 376–383.

[8] Haehyun Cho et al. "Anti-debugging scheme for protecting mobile apps on android platform". In: *The Journal of Supercomputing* 72.1 (2016), pp. 232–246.

[9] Stackexchange Community. *Obtaining root by modifying default.prop(ro.secure)?* 2012. URL: https://android.stackexchange.com/questions/28653/obtaining-root-by-modifying-default-propro-secure.

[10] Stackoverflow Community. *Determine if running on a rooted device*. 2019. URL: https://stackoverflow.com/questions/1101380/determine-if-running-on-a-rooted-device.

[11] Lukas Dresel. *ARTIST*. 2016. URL: https://github.com/Lukas-Dresel/ARTIST.

[12] Frida. *Frida*. 2019. URL: https://github.com/frida/frida.

[13] Github. *Ing Bank app might have busted MagiskHide*. 2019. URL: https://github.com/topjohnwu/Magisk/issues/985.

[14] Google. *Malicious Behavior*. 2019. URL: https://play.google.com/about/privacy-security-deception/malicious-behavior/.

[15] Zhiyun Qian Hang Zhang Dongdong She. "Android Root and its Providers: A Double-Edged Sword". In: *22nd ACM SIGSAC* 2015 (2015), pp. 1093–1104.

[16] Vincent Haupert et al. "Honey, I Shrunk Your App Security: The State of Android App Hardening". In: (2018).

[17] Gustavo Jiménez. *False Positives on the build Keys*. 2017. URL: https://github.com/GantMan/jail-monkey/issues/18.

[18] Jin-Hyuk Jung et al. "Repackaging attack on android banking applications and its countermeasures". In: *Wireless Personal Communications* 73.4 (2013), pp. 1421–1437.

[19] Dima Kozhevin. *APK Signature*. 2018. URL: https://stackoverflow.com/a/50976883/9378427.

[20] larma. *microG Signature Spoofing and its Security Implications*. 2016. URL: https://blogs.fsfe.org/larma/2016/microg-signature-spoofing-security/.

[21] liwugang. *pkcs7*. 2016. URL: https://github.com/liwugang/pkcs7.

[22] L-JINBIN. *ApkSignatureKiller*. 2017. URL: https://github.com/L-JINBIN/ApkSignatureKiller.

[23] Lannan Luo et al. "Repackage-proofing android apps". In: *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE. 2016, pp. 550–561.

[24] magiskroot.net. *Bypass SafetyNet Issue: CTS Profile Mismatch Errors*. 2019. URL: https://magiskroot.net/bypass-safetynet-issue-cts/.

[25] MalwareBytes. *2019 State of Malware*. 2019. URL: https://resources.malwarebytes.com/files/2019/01/Malwarebytes-Labs-2019-State-of-Malware-Report-2.pdf.

[26] microG. *microG*. 2019. URL: https://microg.org.

[27] Nathan Moinvaziri. *Minizip Library*. 2019. URL: https://github.com/nmoinvaz/minizip.

[28] Zhongyuan Qin et al. "Detecting repackaged android applications". In: *Computer Engineering and Networking*. Springer, 2014, pp. 1099–1107.

[29] rovo89. *Xposed Framework*. 2017. URL: https://repo.xposed.info/.

[30] Konstantin Beznosov San-Tsai Sun Andrea Cuadros. "Android Rooting: Methods, Detection, and Evasion". In: *5th Annual ACM CCS* 2015 (2015), pp. 3–14.

[31] Speedtest.net. *Reverse engineering and Java*. 2018. URL: https://www.speedtest.net/insights/blog/2018-internet-speeds-global/.

[32] Stackexchange. *THE WORLD'S INTERNET IN 2018*. 2013. URL: https://security.stackexchange.com/questions/29866/reverse-engineering-and-java.

[33] Stackoverflow. *Detect Emulator on Android*. 2016. URL: https://stackoverflow.com/a/21505193/9378427.

[34] San-Tsai Sun, Andrea Cuadros, and Konstantin Beznosov. "Android rooting: methods, detection, and evasion". In: *Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices*. ACM. 2015, pp. 3–14.

[35] techradar.com. *Google blocked a million apps from the Play Store in 2018 due to security issues*. 2019. URL: https://www.techradar.com/news/google-blocked-a-million-apps-from-the-play-store-in-2018-due-to-security-issues.

[36] Tuo Wang et al. "Towards Android Application Protection via Kernel Extension". In: *International Conference on Algorithms and Architectures for Parallel Processing*. Springer. 2018, pp. 131–137.

[37] Wikipedia. *Rooting (Android)*. 2019. URL: https://en.wikipedia.org/wiki/Rooting_(Android).

[38] John Wu. *Magisk*. 2019. URL: https://forum.xda-developers.com/apps/magisk.

[39] John Wu. *Magisk Details*. 2019. URL: https://github.com/topjohnwu/Magisk/blob/master/docs/details.md.

[40] Wu Zhou, Xinwen Zhang, and Xuxian Jiang. "AppInk: watermarking android apps for repackaging deterrence". In: *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*. ACM. 2013, pp. 1–12.

[41] Wu Zhou et al. "Detecting repackaged smartphone applications in third-party android marketplaces". In: *Proceedings of the second ACM conference on Data and Application Security and Privacy*. ACM. 2012, pp. 317–326.

APPENDIX

Appendix A

In the following pages you will find all the reports from Firebase with the consumption in CPU, RAM and Traffic for each of the devices. There are six figures for each devices containing the different intervals plus the report for the clean apk, without any security detection mechanism. For an analysis on the results please see chapter 6.

Figure A.1 10 seconds - Pixel 2

Figure A.2 5 seconds - Pixel 2

Figure A.3 2 seconds - Pixel 2

Figure A.4 1 seconds - Pixel 2

Figure A.5 0.5 seconds - Pixel 2

Figure A.6 clean App - Pixel 2

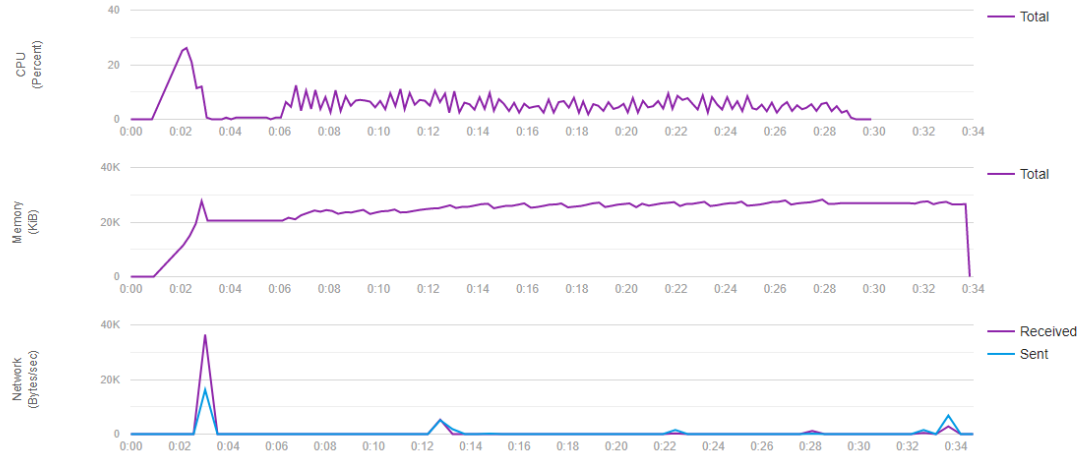

Figure A.7 10 seconds - One Plus 5

Figure A.8 5 seconds - One Plus 5

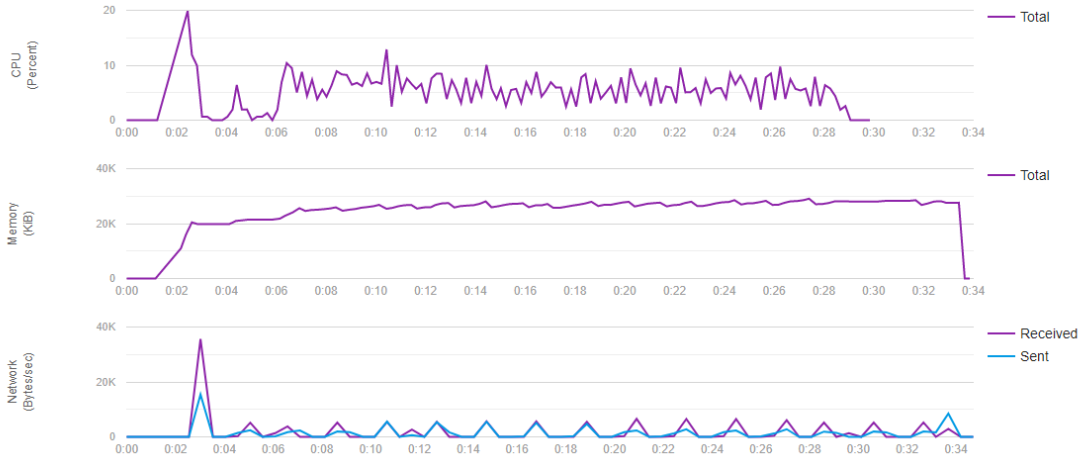

Figure A.9 2 seconds - One Plus 5

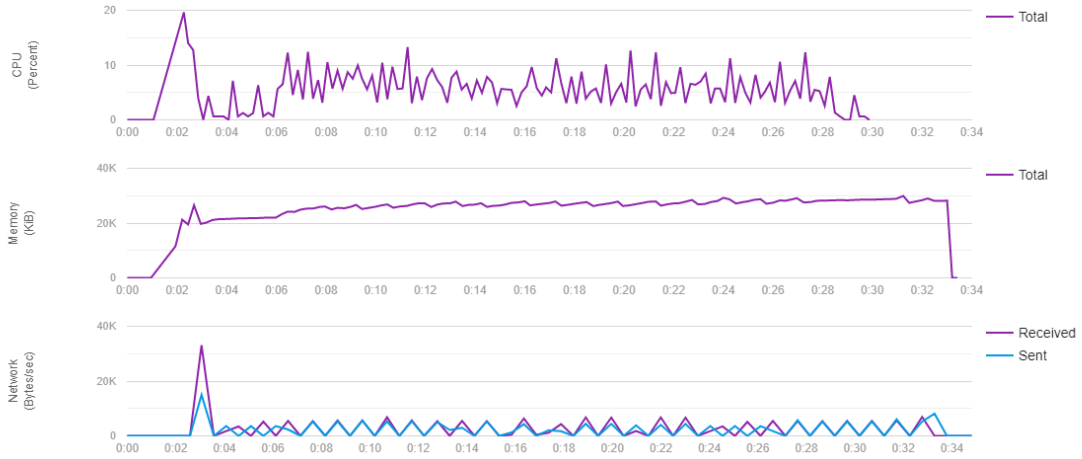

Figure A.10 1 seconds - One Plus 5

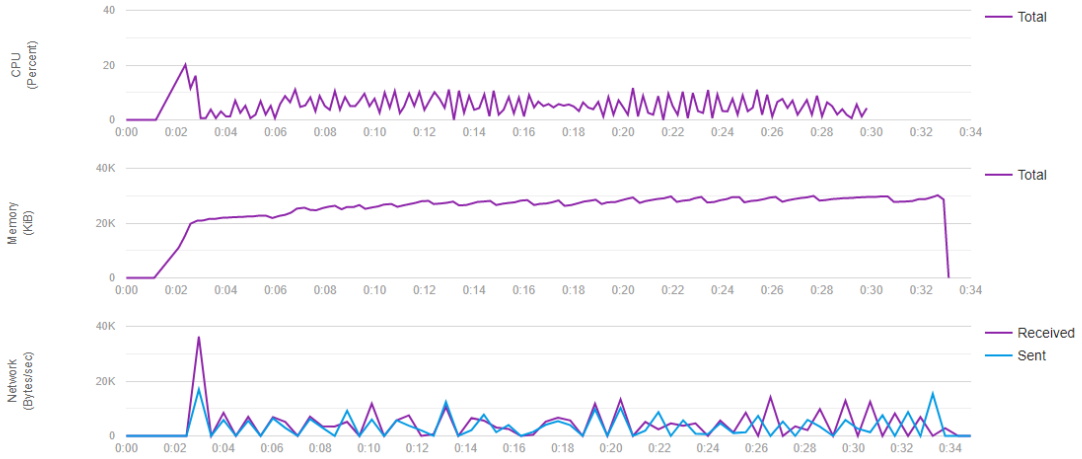

Figure A.11 0.5 seconds - One Plus 5

Figure A.12 clean App - One Plus 5

Figure A.13 10 seconds - Huawei Mate 9

Figure A.14 5 seconds - Huawei Mate 9

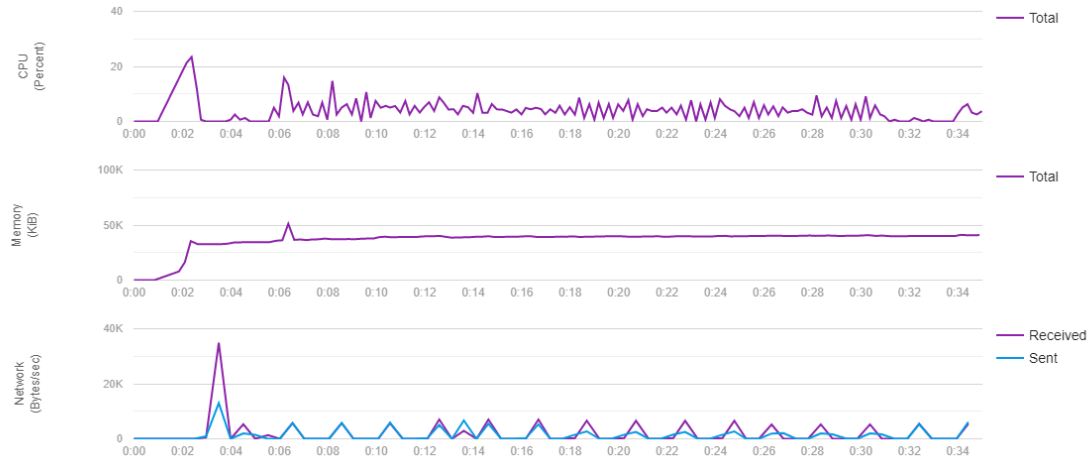

Figure A.15 2 seconds - Huawei Mate 9

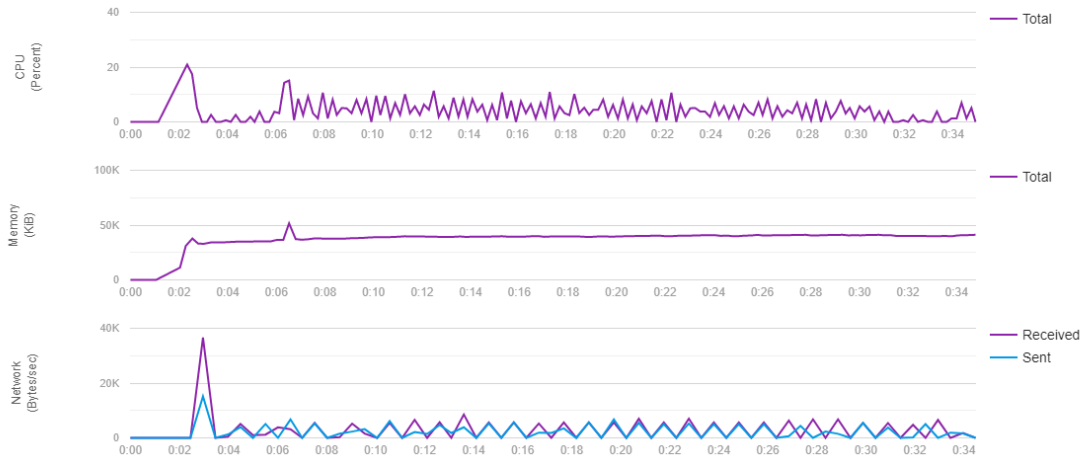

Figure A.16 1 seconds - Huawei Mate 9

Figure A.17 0.5 seconds - Huawei Mate 9

Figure A.18 clean App - Huawei Mate 9

Figure A.19 10 seconds - Xperia XZ1 Compact

Figure A.20 5 seconds - Xperia XZ1 Compact

Figure A.21 2 seconds - Xperia XZ1 Compact

Figure A.22 1 seconds - Xperia XZ1 Compact

Figure A.23 0.5 seconds - Xperia XZ1 Compact

Figure A.24 clean App - Xperia XZ1 Compact

Figure A.25 10 seconds - Nokia 1

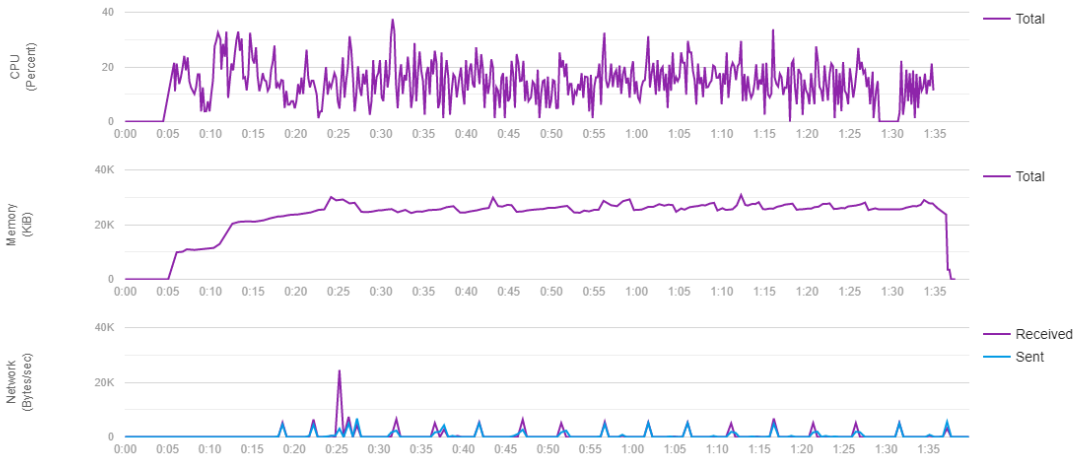

Figure A.26 5 seconds - Nokia 1

Figure A.27 2 seconds - Nokia 1

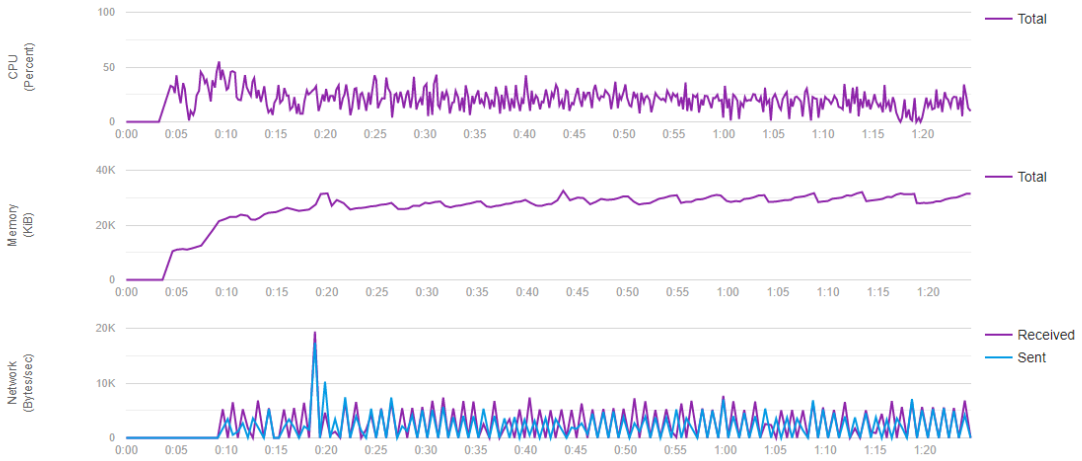

Figure A.28 1 seconds - Nokia 1

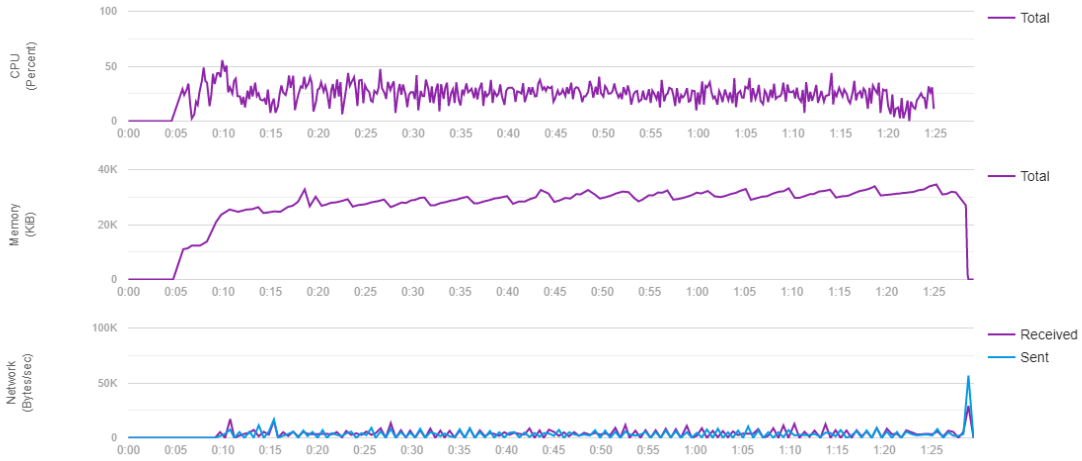

Figure A.29 0.5 seconds - Nokia 1

Figure A.30 clean App - Nokia 1