Revealing Malicious Remote Engineering Attempts on Android Apps with Magic Numbers

Leonidas Vasileiadis
leonidasvas@mail.com
University of Trento
Trento, Italy

Mariano Ceccato
ceccato@fbk.eu
Fondazione Bruno Kessler
Trento, Italy

Davide Corradini
corradini@fbk.eu
Fondazione Bruno Kessler
Trento, Italy

ABSTRACT

Malicious reverse engineering is a prominent activity conducted by attackers to plan their code tampering attacks. Android apps are particularly exposed to malicious reverse engineering, because their code can be easily analyzed and decompiled, or monitored using debugging tools, that were originally meant to be used by developers.

In this paper, we propose a solution to identify attempts of malicious reverse engineering on Android apps. Our approach is based on a series of periodic checks on the execution environment (i.e., Android components) and on the app itself. The check outcome is encoded into a Magic Number and send to a sever for validation. The owner of the app is then supposed to take countermeasures and react, by disconnecting or banning the apps under attack.

Our empirical validation suggests that the execution overhead caused by our periodic checks is acceptable, because its resource consumption is compatible with the resources commonly available in smartphones.

KEYWORDS

Remote attestation, Malicious reverse engineering, Code tampering

ACM Reference Format:
Leonidas Vasileiadis, Mariano Ceccato, and Davide Corradini. 2019. Revealing Malicious Remote Engineering Attempts on Android Apps with Magic Numbers. In *9th Software Security, Protection, and Reverse Engineering Workshop, December 9-10 2019, San Juan, Puerto Rico, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

The extensive diffusion of Android attracted not only the attention of end-users and app providers, but also of attackers interested in applying malicious reverse engineering to tamper with app code[1]. Attackers can be willing to alter the behaviour of apps for many reasons, such as unlocking premium features for free, or removing advertisements. As a matter of fact, many tampered app (also called *repacked* or *piggybacked* apps) can be identified across different app stores [13].

It is notorious the case of a popular music player that adopted a freemium business model. This app offered all its end-users free listening to music, and premium features (i.e., no ads) only to paid accounts. This app attracted the attention of attackers, that distributed tampered versions with unlocked premium features for free. Eventually, these tampered versions were so popular and so easy to find, that app owners started a cleaning campaign to ban all the accounts supposed to use a hacked version[2].

An extensive user study conducted with professional hackers [6, 7] revealed that (malicious) reverse engineering is a prominent task to plan, elaborate and assess tampering attacks. In fact, before figuring out how to change the code, an attacker has to identify where to perform the change. Thus, in order to mitigate tampering attacks, it is of crucial importance to limit the possibility of understanding and analyzing the code.

In this paper, we propose a solution to detect attempts to apply malicious reverse engineering to Android apps. In particular, we propose a series of checks to verify several distinct aspects of the execution environment, to spot different classes of reverse engineering tools, such as emulators, root access, instrumentation frameworks and customized (e.g., tampered) operating system. Subsequently, in case the execution environment is safe, we propose also to check the app itself, to spot possible instrumentation and tampering. The result of all these checks is used to compute a *Magic Number* that is sent over to a remote server, where the check result is verified by the app owner. In case this verification fails, the app might be considered under malicious reverse engineering and, thus, the app owner might react e.g. by disconnecting suspicious app instances [4, 5, 8].

Similarly to other protections, also our solution comes with the cost of increased resource consumption. Thus, in this paper, we additionally present the results of our empirical investigation, conducted to quantify the cost of our solution. Memory, CPU and network overhead has been measured on distinct devices when the checks are more and more frequent. Empirical data seem to indicate that our solution can be largely adopted in a wide set of devices.

The rest of the paper is organized as follows. Section 2 covers the background about some popular tools and approaches to analyze Android apps. Section 3 presents the high-level architecture of our approach, that is later discussed in detail. In particular, Section 4 covers all the checks and Section 5 describes how check results are

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

SSPREW 2019, December 9-10, 2019, San Juan, Puerto Rico, USA

ACM ISBN 978-1-4503-9999-9/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

[2] J. Sommerlad, Spotify Cracks Down on Premium Pirates Streaming for Free. https://www.independent.co.uk/life-style/gadgets-and-tech/news/spotify-premium-piracy-crackdown-apps-bypass-restrictions-accounts-deactivated-music-streaming-a8241936.html

turned into the Magic Number that is reported to the server. Our empirical validation is described in Section 6. After comparing our approach with the state of the art in Section 7, Section 8 closes the paper.

2 BACKGROUND

This section presents the background knowledge about rooted devices, SELinux and Magisk.

2.1 Android Root Permission and Rooted Devices

In Unix-like operating systems (such as Android), we refer to *root* as a user with administrative capabilities. Root is granted full control of all functioning of the operating system with no limitations. Conversely, a regular user has a limited access to the filesystem and devices. In Android, limitations and boundaries 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, who (accidentally or not) might alter or replace system applications and settings [24].

A rooted device is a device where an app or a process can be granted root permissions. When root permissions are granted, system security settings can be changed, that would be otherwise inaccessible to a normal Android user. Further, the entire operating system could be replaced with a custom one (custom ROM) that would enforce no restriction to the user, letting her/him modify any part of it at will.

There are two main ways to obtain root privileges. A vulnerability in the device can be exploited to do privilege escalation, or by executing the `su` binary from within the device, that starts a new process with root privileges. The first one is referred to as *soft root*, while the second one is known as *hard root*. Both methods result in altering system files and properties. These change traces represent valuable clues that could be exploited to reveal root access.

2.2 SELinux and Magisk

SELinux[3], stands for *Security Enhanced Linux*, it is a kernel module meant to enforce limits to all users, including root. Starting from Android 5, SELinux is enabled by default on devices, to enforce mandatory access control over all the processes. Thus, on recent devices, acquiring root privileges is not enough to mount a successful attack, but additionally SELinux policies should be disabled or patched. This strategy is also applied by Magisk[4].

Magisk is a "systemless" root method, which is installed by modifying the boot image file *boot.img*. As soon as the device boots using this patched image, the SELinux policy rules are patched to have an unrestricted context and a Magisk daemon is started in this unrestricted context. After this, the boot continues as in the original device. In this way, the daemon has full capabilities of root within an unrestricted SELinux context.

When an app needs root access, it requests it to the Magisk daemon (by executing the Magisk *su* binary). To hide its traces, root privileges are granted without changing the *UID/GID* of the process, but by opening a root shell that is accessible via a UNIX socket.

Magisk is referred to as *systemless root*, because it tampers with the boot process and not the */system* folder. However, in case a particular attack needs to alter the */system* folder, e.g. by adding executable binaries there, Magisk supports it in a stealthy way. Instead of changing the folder (which could be detected), Magisk exploits bind mounts[5], to dynamically mount and unmount files in folders and by overlaying them, so the *systemless* feature is preserved.

It is worth noting 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[6] using the features offered by Magisk.

3 REFERENCE ARCHITECTURE

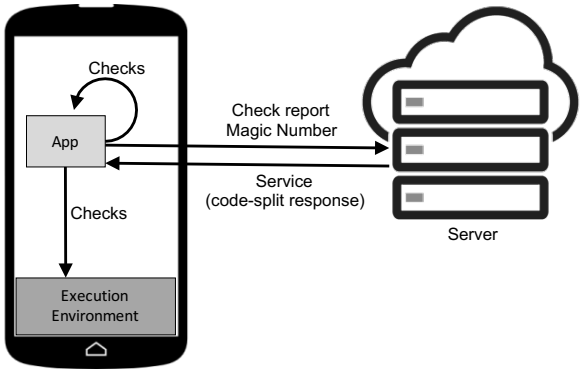

Figure 1: Reference architecture.

Our approach is composed as shown in Figure 1. When malicious reverse engineering is a problem, the app performs a two-fold check:

Execution environment check: First of all, the underlying Android environment is checked to spot traces and clues of reverse engineering tools, analysis frameworks and inspection attempts. These checks include verifying if the app is executed within an emulator, or in a device with *root* access, if the file-system has been altered to support dynamic analysis, if the standard Android environment has changed in an inconsistent way, if traces can be found of dynamic instrumentation tools and what kernel the underling linux layer is using.

App self-check: Even if no traces of malicious reverse engineering can be found on the underlying Android execution environment, we still need to check the integrity of the app, because an attacker might try to tamper with the app code and instrument it by injecting tracing instructions, in order to perform dynamic analysis by inspecting execution traces. The verification of the app integrity makes sense only when the checks on the Android execution environment pass, so we are confident that the app self-checks are not subverted by the underlying Android infrastructure.

The detail of all these checks is presented in Section 4.

[3]https://source.android.com/security/selinux
[4]https://github.com/topjohnwu/Magisk
[5]https://github.com/topjohnwu/Magisk/blob/master/docs/details.md
[6]https://magiskroot.net/bypass-safetynet-issue-cts/ "Bypass SafetyNet Issue: CTS Profile Mismatch Errors"

The results of these checks are included in the security report and also encoded as part of a *Magic Number*. More details about the check report and the magic number are included in Section 5. The Magic Number is computed by the app in a way that:

- The Magic Number is different for each execution and can not be replayed. I.e., the attacker can not just intercept this value for a valid app and reuse it in a tampered app; and
- It is difficult for an attacker to guess and predict what is the right value to use.

The algorithm used to compute the Magic Number is described in Section 5.

Based on the received Magic Number, the server is assigned the responsibility to make the final decision about whether the app is working as expected or if it is under malicious reverse engineering.

Server-side reaction: In case an attack is detected, countermeasure might be taken to react and block the app under malicious reverse engineering. A reaction by the app code would have limited effectiveness, because an attacker might tamper with app code and skip the reaction, by simply replacing the decision making code with the constant decision of never reacting. We suggest to make a more secure decision at the server side. In case the server notices malicious reverse engineering on a client app, it would immediately block that app, for instance by stopping delivering content.

A possible approach to block apps controlled by malicious users or by attackers is represented by *client-server code splitting* [2, 3, 22]. According to this approach, a part of the app is *sliced* away, such that (i) the *slice* is very small with respect to the size of the whole app; and (ii) the *slice* is crucial for the correct functioning of the app because, without that part, the app does not work properly.

The *slice* is never delivered to the end-user, and it is installed on the server, where it is remotely executed every time this is needed by the sliced app. In this way, to work properly, the sliced app always requires a server to execute its missing code. The protection consists in refusing to remotely execute the *slice* for those client apps where malicious reverse engineering has been detected.

Client-server code splitting is not a novel contribution of this paper. A detailed description of it can be found on those articles that proposed and assessed this reaction technique [2, 3, 22]. Reaction is in general out of the scope of this paper

4 DETECTION OF MALICIOUS REVERSE ENGINEERING

Malicious reverse engineering can be deployed in many different ways. Based on the results of user studies involving professional hackers [6, 7], we collected a list of attack strategies and attack tools that are relevant for program understanding, malicious reverse engineering and code tampering. Additionally, we surveyed practitioner and technical informal literatures, including forums (such as stackoverflow and slashdot), guidelines (by Google, Facebook and Owasp) and Android security blogs (such as http://www.vantagepoint.sg/blog/).

Based on our survey of recommended checks, we implemented the following approaches.

4.1 Emulator Detection

To reverse engineer an app, it can be quite useful to run it in an emulator, i.e. a simulated smartphone that runs on the developer computer and controlled by the development environment. Our approach to emulator detection is based on identifying Android properties and directories known to be used by emulator engines. A similar method based on Android Properties is also adopted by Facebook in their *react-native framework*[7] for detecting emulators. Table 1 contains some values of Android properties that we use to this aim.

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

Table 1: Android properties used for emulator detection.

Considering that reading Android properties might be hooked or circumvented by an expert attacker, we complement it with an additional lightweight implementation. Manually analyzing the Genymotion emulator and the Bluestacks, we listed some file-system paths and files that are unique to these emulators and, thus, can be used to identify these emulators. Table 2 includes a small part of the paths that we are checking.

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

Table 2: Paths used for emulator detection.

4.2 Root Detection

Attacks can be mounted by acquiring *root* privileges. In fact, an app with root privileges (i.e., the system administrator in linux), not only can access all the hardware components (network interface, I/O) but it can also break the Android sandboxing principle and read/write other apps private data.

Our approach for root detection is based on looking for the binaries used for gaining root privileges, or for specific files known to be used by root managers.

First of all, we use the linux command "*which su*", to reveal the presence of *su* binary in the system, that can be used to run a process with root privileges.

Then, we check for the presence of some files, that could be a clear indication of the presence of root in the system. The first one

[7] https://stackoverflow.com/a/21505193/9378427

is *SuperSu*, a commonly used root manager. It modifies the *system* files and installs its APK in the path */system/app/Superuser.apk*.

Other paths we check are */data/adb/magisk.db*, */data/magisk/resetprop*, */cache/magisk.log* and */sbin/.magisk/mirror/data/adb/magisk.db* with the aim of detecting files that the *Magisk* manager uses. Additionally, we check several other locations known to store binaries related to root like *busybox* and *su*, and other properties specific to Magisk.

Finally, we check if other root managers are installed on the device, or apps that require root privilege to work. Some of these apps is listed in Table 3.

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

Table 3: Apps indicating root privileges.

4.3 File System Read/Write Mode Check

Suspicious permissions granted to apps or file-system permissions can be a clue that a malicious reverse engineering environment has been setup, or that root privileges could have been acquired. As already mentioned above, bypassing Android sandboxing usually results in several system files with changed permissions, or additional files that should not exist.

For security and stability reasons, normally, specific parts of the Android filesystem are read-only. Yet, when an attacker acquires root privileges, specific directories are mounted in *read-write* mode, in order to enable the modification of system files and complete specific attack tasks. Our approach consists of checking the read/write mode of specific paths in the file-system, some of them coming from those involved in RootBeer[8], a prominent library for Android root detection. These are */system*, */system/bin*, *system/sbin*, *system/xbin*, */vendor/bin*, */etc* and */sbin*.

4.4 SELinux Status Check

SELinux stands for *Security Enhanced Linux* and it was introduced in Android to enforce mandatory access control over all the processes. Since Android 5, SELinux is set to `enforced` by default[9], to limit what can be accessed or modified, even by root. This means that, even if root is present, when SELinux in is enforcing mode, modifications to system files are blocked at kernel level. Thus, the kernel on attacker devices is often patched to support a permissive or disabled SELinux, to let an attacker fully control the device.

In our implementation, we employ two ways to detect the state of SELinux, at increasing level of complexity. The first approach is still based on Android properties, in particular the properties *ro.build.selinux* and *ro.boot.selinux*. The first one reports the SELinux status from the boot image *boot.img*, while the second one checks if the Kernel is currently configured in the permissive mode.

For the second way, we propose to indirectly detect SELinux by identifying the contents of the `enforce` file, which is a file holding the current mode of SELinux. To this aim, we read the file */proc/filesystems*, which is supposed to list those filesystems that were compiled into the kernel, or whose kernel modules are currently loaded. Thus, we detect if the *selinuxfs* file system is present, which would imply that SELinux is currently loaded in the kernel.

Since Android 8, reading this file should be blocked by SELinux[10]. Thus, a successful read in Android 8 is already a clue of malicious reverse engineering, because it means that SELinux is disabled or in permissive mode.

Alternatively, for previous versions of Android, we try to detect the value *selinuxfs* in */proc/filesystems*. In case we find it, we know that SELinux is running, but we still do not know in what mode it is running. In order to determine the mode, we search for *selinuxfs* in */proc/mounts* and, in particular, for a file named *enforce*. If the content of this file is "1" the SELinux is in *enforcing* mode, otherwise if the content is "0" the state is *permissive*.

4.5 Android Properties Check

System properties can be used to spot malicious reverse engineering. Upon boot, the file *Default.prop* is copied from the boot image *boot.img* to the *System* directory. This file contains all information about the specific build in execution, such as whether the *debuggable* flag is enabled, if *adb* can run as root, and many other information. These information are essential in detecting a setup that could be used for malicious reverse engineering.

A debugger is commonly used to test an app under development. In Android, the debugger communicates with the Dalvik virtual machine (that runs the app code) using Java Debug Wire Protocol (JDWP), a protocol that allows an app development platform to read/write the state of a running app, to inspect its memory and to do step-wise execution. To this aim, a dedicated thread is created when the Dalvik virtual machine starts an app, that waits for a debugger to attach.

The Dalvik virual machine decides whether to create this thread or not based on the *ro.debuggable* flag[11]. This flag means that the debugging protocol should be activated for the app under development. This would allow an attacker to attempt dynamic analysis.

If also flag *ro.secure* and flag *service.adb.root* are detected, we can be quite confident not only that debugging is activated, but also that it is allowed to run as root. This grants all privileges to the attacker connecting through *adb*, and allows her/him to manipulate the device at will. This is a common strategy used to remount the *system* directory and to grant root access to the device, or when Dynamic Binary Instrumentation tools are used[12].

4.6 Dynamic Binary Instrumentation Detection

An effective practice to reverse engineering Android apps is to use *hooking* frameworks and dynamic instrumentation tools. There are

[8] https://github.com/scottyab/rootbeer/
[9] https://source.android.com/security/selinux
[10] https://github.com/CypherpunkArmory/UserLAnd/issues/59
[11] https://developer.android.com/studio/debug
[12] https://android.stackexchange.com/questions/28653/obtaining-root-by-modifying-default-propro-secure

many of these tools available, including Frida[13], Artist[14], Whale[15] and Xposed[16].

Xposed can be used to hook into method calls. The hooking capabilities allow an attacker to inject new code right before or right after a hooked method call, and thus modifying the behavior of the app at specific places. This is achieved by Xposed by replacing the file */system/bin/app_process* with a custom one that contains the Xposed runtime, because this file is used by Android to spawn new app processes. A succesfull attack can be scripted as an Xposed module and then shared. Modules have the form of Android apps that, once installed on a device, will be recognized by the Xposed framework and automatically loaded and applied to mount an attack.

Our approach aims at detecting indirect clues of the presence of Xposed and Frida. In fact, Xposed and Frida require some libraries to be mapped into the memory of the app, in order to let an attacker inject code. We resort to */proc/self/maps*, that describes a region of continuous virtual memory in the app. To detect Xposed, we look for libraries *app_process32_xposed* and *XposedBridge.jar* which we know are loaded by default. For Frida, we specifically look for the library *frida-agent-32.so* which is loaded upon hooking. To avoid being detected and hooked, instead of implementing this detection in Java, we implemented it in C using the *fopen* primitive to read */proc/self/maps*.

These clues might change in future versions of the dynamic instrumentation frameworks, for example because of library renaming. Thus, in order for our detection to stay accurate, we need to monitor updates and the evolution of these frameworks, and update our detection clues accordingly.

4.7 Kernel Signature Check

An alternative approach to detect a rooted Android device is by detecting a custom build with the *ro.build.TAGS* property. This is the Android property that contains the tags that describe 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 three values, namely *"test-keys"*, *"dev-keys"* and *"release-keys"*.

The first two values mean that the kernel was compiled but **not** signed by an official developer. A third party, instead, signed the kernel, for instance an attacker built and signed it, after applying custom changes to enable malicious reverse engineering. This is a hint that this device can not be considered a safe environment, because system components could have been hijacked.

In case the value *"release-keys"* is found, it indicates that an official developer signed the build, so this kernel is more trustful.

4.8 App Signature Check

In case an attacker managed to bypass all the security checks implemented, the tampered app is repackaged and signed, of course using a different singing key than the secret key owned by the original developer. By implementing one additional security check, we can verify whether the app has been repackaged, based on its signature. The way we retrieve the signature is critical, because tools exist that automatically bypass common methods for reading the signature. In fact, an attacker might make tampered apps report the original signature, thus bypassing signature checks.

Signature spoofing can be done in several ways:

- **Patching the system:** this consists in tampering with the underlying Android framework, to make it return the original signature instead of the actual one, when the app requests it. This can be achieved using an Xposed Module[17], by patching the system with a tool like *Tingle*[18], or using a custom ROM; or
- **Patching the app:** Directly tampering with the app, and dynamically divert the call to `getPackageInfo` of the PackageMangaer, to some custom code that ignores the invalid signature provided by Android and just returns a local copy of the original signature expected by the app check.

Our protection is inspired by a StackOverflow post[19], i.e. avoiding calls to Android libraries, because they could be *hooked* by an attacker to circumvent our check.

Initially, we get the package name of the app by inspecting */proc/self/cmdline*. Then, we read */proc/self/maps*, which is a file containing the currently mapped memory regions and it includes also 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 file. Then, we resort to the *minizip library*[20] to read the content of the APK file and, in particular, the *META-INF/*.RSA* signature file, where the star * matches any signature file name. Using the *pkcs7 library*[21], we verify that the file we chose is actually a signature file, and then we parse the certificate and we extract the public key, which is it hashed (using SHA256) and added to the *check report* that will be sent to the server.

4.9 Security Report

When the security checks are completed, a report is filled with the check results. This report includes the app signature as *SHA256* hash. Then, the report continues with the detailed results of all the checks, encoded as a 10-bit value as shown in Table 4.

Position	Security Check
1-2	Emulator Detection
3	Root Detection
4	Root Detection (Apps)
5	FS Read/Write Mode
6	SELinux Status
7	Android Properties
8-9	Dynamic Binary Instrumentation
10	Kernel Signature

Table 4: 10-bit value as report of the security checks.

[13]https://github.com/frida/frida
[14]https://github.com/Lukas-Dresel/ARTIST
[15]https://github.com/asLody/whale
[16]https://repo.xposed.info/
[17]https://repo.xposed.info/
[18]https://github.com/ale5000-git/tingle
[19]https://stackoverflow.com/a/50976883/9378427
[20]https://github.com/nmoinvaz/minizip
[21]https://github.com/liwugang/pkcs7

The first two bits are for the result of emulator detection, indicating that an emulator was found. Two bits are needed because two different checks are performed, each of them having a separate indicator and each could provide useful information either on its own or in combination with the other one. The bit in position three and four represent the result of root detection, respectively at Android level and at the app level. For the Android level we employ techniques to identify existence of root like detecting the root binary, and for the app level we check whether any of the installed apps is known to require root. The subsequent bits report the checks on the file system (fifth bit), the status of SELinux (sixth bit) and Android properties (seventh bit). Then, two bits (eighth and ninth bit) are for the result of Dynamic Binary Instrumentation detection, one of them reports the result for finding files related to the frameworks, while the other one indicates if a library belonging to a framework is already loaded. This information is useful on the server side, because if the first bit is not flipped but the second one is, it would mean that an attacker managed to bypass the first security check while failed for the second one which is much harder. In that case, a stricter policy could block the app more urgently, or ban it for a longer period of time. The last bit (i.e., the tenth bit) is related to the check of the kernel signature.

5 CHECK REPORT AS MAGIC NUMBER

When all the verifications and checks described in the last section are complete, their outcome should be sent to the server for validation. In order to defend against replay attacks or precomputed report values, we propose to use a *Magic Number*. This approach is based on an algorithm that takes into account check results to compute a (magic) numeric value. Then, the server is supposed to verify this value using the same algorithm. If the expected and received values do not match, it means that a malicious reverse engineering attack might be in place, so the client that sent the wrong Magic Number should be disconnected.

The Magic Number is composed of two main parts. The first part is a random unique id, and the second part is a security report checksum.

5.1 First Part

The first part of the Magic Number is computed using the pseudo code shown in Algorithm 1. `InitRandomNumber` is assigned a random value that is `n` digits long. This same value is used to compute other two values, i.e. `sec` and `third` in a deterministic way, using arithmetic expressions, based on some constant values X_1, Y_1, X_2, Y_2 that are decided at protection time (either configured by the developer or randomly generated otherwise when compiling the code). Eventually, each of these three values is truncated to keep only its first `n` characters, then these values are all concatenated and returned.

The concatenated value will be validated at the server side. First of all, the server will check that this value has not been observed earlier in other messages. Second, the arithmetic operations are checked, following the exact same process as the client used initially to calculate the number, to verify that the number is consistent.

The length `n` of the initial random number and the length of the concatenation (i.e., `3*n`) can be configured by the developer, to guarantee enough variability to support a large number distinct check messages. A longer number and more complex arithmetic operations could be used, to enforce a more secure scheme against attackers.

Algorithm 1 Magic Number - Part 1

```
1: procedure
2:     InitRandomNumber ← random n-digits number
3:     sec ← InitRandomNumber * X1 + Y1
4:     sec ← sec(0 : n)
5:     third ← InitRandomNumber * X2 + Y2
6:     third ← third(0 : n)
7:     return ← concatenate(InitRandomNumber, sec, third)
```

5.2 Second Part

The second part in the magic number represents a checksum of the report. The pseudo code in Algorithm 2 shows how this second part is computed. It uses a portion of the app signature, a portion of the Android user-ID that is running the app, a portion of the first part of the Magic Number already computed and a portion of the 10-bit number report (see Section 4.9).

Algorithm 2 Magic Number - Part 2

```
1: procedure
2:     SignSlice ← signature(S1 : S2)
3:     UIDSlice ← userid(U1 : U2)
4:     FSlice ← firstMagicNumber(F1 : F2)
5:     IndicatorsCnt ← Count occurences of 1
6:     concatenate:
7:     s ← concatenate(SignSlice, UIDSlice, FSlice, IndicatorsCnt)
8:     List = []
9:     for character in s do
10:        List ← abs(Unicode(character)) + X3
11:    return ← String(List)
```

Some characters are taken from the app signature (from the S_1-th to the S_2-th character) and some characters from the Android user-id (those with positions in the interval [U_1,U_2]). Then, a portion of the first part of the Magic Number is also taken, in particular the characters from the F_1-th to the F_2-th are considered. The last value (i.e., *IndicatorsCnt*) is computed by counting how many occurrences of "1" are found in the 10-bit number included in the report.

Here, we only include the number of occurrences of "1", but not specific positions of them, because any non-zero value indicates a security alert. An attacker would try and hide an attack by removing a "1" from the security report. However, the attacker would not be able to predict what other changes are required to the magic number in order to make it consistent with the tampered report.

These four values are concatenated in the string `s`, that is later converted character-by-character into a string composed only of digits, by using the Unicode representations of each character, plus the constant value X_3. This last string is then interpreted as a numeric value and it represents the second part of the Magic Number.

The constant values S_1, S_2, U_1, U_1, F_1, F_1 and X_3 are decided at protection time, possibly by the developer who applies our approach.

5.3 Server Side Validation

When the report is received at the server side, the server checks this Magic Number against the complete check report. This verification follows the same algorithms used to compute this value at client side. This includes not only checking that the first and second part of the Magic Number are correct, but also that they are consistent to each other, by verifying the portion of the first part that is used to compute the second part.

This approach meets keys security requirements in order to block these attacks:

Replay attack: Every execution of the above algorithms would produce a new, different value for the Magic Number, because the first part is based on a random value generated on each execution. Thus, a valid Magic Number intercepted during a valid execution of the app can not be reused multiple times. In fact, reusing a value would be easily detected at server side, when the same value is observed twice.

The randomness of the first part of the Magic Number is based on the `srand()` function in `C`, seeded with the time in seconds. This allows to have a different random number after one second. This means that when security checks are required faster than once per second, the same random number is reused. This is not a problem, because our server side implementation is compatible with this scenario. In fact, the server allows the same magic number to be reused by the same user within one second. Additionally, a collision in the first part of the Magic Number among two different users is allowed, because their second part (which includes the User-IDs) would be different and, therefore, distinguishable at server side. Conversely if the same Magic Number is repeated across different users, it means that both parts of the Magic Number are spoofed and, therefore, we assume an attack is attempted that should be blocked by the server.

Precomputation attack: An attacker might try to compute a correct value offline, and then hardcode it in a tampered app, to send a valid Magic Number to the server, even if the app is under malicious reverse engineering. This attack is very hard to mount, first of all because a single valid value can not be reused multiple times. Then, it is quite hard to reverse engineer the algorithm used to compute the Magic Number, in fact its implementation is native code (compiler C code should be harder to analyze than Java) and it is obfuscated. Further, arithmetic operations in Algorithm 1 and constant values used in Algorithm 1 and in Algorithm 2 can be arbitrarily changed, virtually enabling unlimited new versions of the app for frequent updates [4, 5, 8], thus making potential precomputation attacks expire soon.

6 EMPIRICAL VALIDATION

In this section, we evaluate the resource consumption of our approach for detecting malicious reverse engineering.

6.1 Research Questions

To conduct our empirical validation we formulated the following research questions:

- **RQ**$_{CPU}$: What is the computational load overhead caused by our approach?
- **RQ**$_{Mem}$:What is the memory overhead caused by our approach?
- **RQ**$_{Net}$:What is the network overhead caused by our approach?

The approach proposed by us to detect malicious reverse engineering comes at cost of some performance overhead. In particular, the first research question is meant to investigate the CPU consumption of our checks. The second research question is still about the overhead of our approach, but on the memory consumption. Eventually, the third research question measures the network usage, needed to communicate the check results to the server. By answering these research questions, we will be able to quantify if the cost of the proposed approach is affordable for smartphones.

6.2 Experimental Environment

We run the experiment on *Firebase Test Lab*[22], a paid service provided by Google to test apps. This service also provide analytics on the CPU, RAM and Network consumption on the app under test. This service allows to run tests on physical devices and on emulators.

6.3 Metrics

To answer research questions, we adopted the following metrics:

- *CPU Load:* We measure the CPU consumption using the inbuilt tools provided by Firebase Test Lab. The tools report an average of the overall CPU consumption of the device.
- *Memory:* The amount of memory used by the device, measured in MB, as reported by Firebase Test Lab.
- *Net:* The amount of data in KB exchanged via the network interface, either *Received* or *Sent*. They are measured by Firebase Test Lab platform tools.
- *Interval:* The amount of time (measured in seconds) between two consecutive security checks. This interval is configured by us and hard-coded in the app under analysis.

6.4 Devices, App and Scenario

We chose devices that are commonly used and that have different Android versions. The list of devices used in our empirical validation is shown in Table 5. The first column reports the model of the device, the second column is the *API* level supported by the device, which maps directly to a version of Android. Then, the table reports the number of cores available in the device (third column) and their speed in GHz (fourth column). The last column reports the amount of RAM memory in GB.

While the majority of the devices have many cores and at least 4 GB of memory, Nokia 1 is a peculiar case. We chose that device because it belongs to the low-end spectrum of Android smartphones. We considered it an interesting scenario to observe the impact of our

[22]https://firebase.google.com/products/test-lab/

Device	API	Cores	CPU Freq	RAM
Pixel 2	28	8	4x 2.35 + 4x 1.9	4 GB
One Plus 5	26	8	4x 2.35 + 4x 1.9	6 GB
Huawei Mate 9	24	8	4x 2.4 + 4x 1.8	4 GB
Xperia XZ1	26	8	4x 2.45 + 4x 1.9	4 GB
Nokia 1	27	4	4x 1.1	1 GB

Table 5: Hardware specifications of devices used in the empirical validation.

implementation in low-end devices, where RAM and CPU resources are limited.

As a testing app, we used a simple calculator. This app does only basic arithmetic operations, and we chose it due to its simplicity and the low footprint on resource usage. So, we expect the impact on the resource consumption due to security checks to be evident. We created five different versions of this app, with different intervals between running the security checks. We chose to run the tests with intervals of 10, 5, 2, 1 and 0.5 seconds. Although running a security check every half a second may seem extreme, we included this configuration in our experimental setting to collect results also in particularly adverse case, towards a worst case analysis.

To run this app in Firebase Test Lab, we defined a use case that resemble a normal execution scenario by the end-user, because we are interested in quantifying the overhead that an end-user would experience in her/his daily usage. This scenario consists in 21 arithmetic operations between numbers, like *5 + 2 = 7*, where overall it lasts approximately 40 seconds. The test is recorded using a Roboscript a capture-and-replay tool available in Firebase Test Lab.

6.5 RQ$_{CPU}$: Results of Computation Overhead

Figure 2 shows the average of CPU consumption for the five different check intervals (x-axis). The first value on the left-hand side represents the original *clean* app with no security checks, as a baseline reference for comparison. Different devices are represented with lines with different colors.

We can notice that for almost all devices (except Nokia 1), the CPU consumption is almost constant for any check interval. Instead, for Nokia 1, CPU consumption increases linearly as the check frequency increases.

Table 6 reports the result of a numerical comparison between the clear and the protected app, each column represents the increase when a different check interval is experimented. Different devices are represented in different rows.

We see that for the shortest interval, most of the devices report an increase in CPU consumption of at most 1.3%. For the Nokia 1, instead, we see that it reaches almost 12% increase, when checks are performed each 0.5 seconds. However, CPU overhead is a reasonable 1.75% when we check every 5 seconds.

Considering that Nokia 1 is a low-end device, these results suggest that too frequent security checks (such as twice per second) are not appropriate. We observe an acceptable impact on CPU consumption only on mid and high-end devices.

Devices	10 sec	5 sec	2 sec	1 sec	0.5 sec
Pixel 2	0.26	0.33	0.28	0.75	0.85
OnePlus 5	1.33	1.56	1.48	1.75	1.29
Huawei Mate	0.12	0.44	0.54	0.74	0.94
Xperia XZ1	0.05	0.17	-0.27	0.82	1.18
Nokia 1	3.49	1.75	4.61	7.66	11.78

Table 6: Average CPU load increase depending on the interval between checks.

6.6 RQ$_{Mem}$: Results of Memory Overhead

Figure 3 shows the RAM consumption (in MB) on different devices for different check intervals, with the original app with no check on the left-hand side. The difference in memory consumption between the original and the protected app is shown in Table 7. As we can see, the RAM consumption increase does not depend on the check interval. Thus, the amount of RAM required for the security checks is independent of how often the security check is occurring. The increase of RAM consumption reaches at most, 13 MB for the Pixel 2 device, while for the rest of the devices it remains lower than 10MB.

We can conclude that the footprint of the security checks is relatively lightweight, even for the Nokia 1 device, where the available RAM is limited to 1 GB. There is a slight increase in the gradient in the measurements when the frequency of the checks increase.

Devices	10 sec	5 sec	2 sec	1 sec	0.5 sec
Pixel 2	11.8	11.6	10.8	12.2	13.6
OnePlus 5	5.8	6.1	6.8	7.1	7.6
Huawei Mate	7.1	7.6	7.8	8.3	8.6
Xperia XZ1	5.4	6.0	6.5	7.4	7.9
Nokia 1	7.3	6.0	8.0	8.6	9.8

Table 7: Average RAM consumption increase (measured in MB) with decreasing interval between checks.

6.7 RQ$_{Net}$: Results of Network Overhead

Figure 4 and Figure 5 show the network resources consumed during the experiment, respectively for the incoming traffic and for the outgoing traffic. The app with no check is displayed on the left-hand side, followed by apps with checks at increasing frequency (decreasing check interval).

As expected, the network overhead increases when checks are more frequent and more check reports are sent. Additionally, we also observe different network usages for the different devices. In particular, the gap between Nokia 1 and Xperia XZ1 is quite significant. This is explained by considering how the execution scenario is run. Each scenario includes the same interactions (i.e., 21 different arithmetic calculations) to be performed, but this scenario takes shorter on faster devices (30 seconds Xperia XZ1) on and it takes longer on slower devices (80 seconds Nokia 1). Since the checks are performed at a strict pace, different duration of the execution scenario means different number of checks performed and thus, different number of check reports sent to the server.

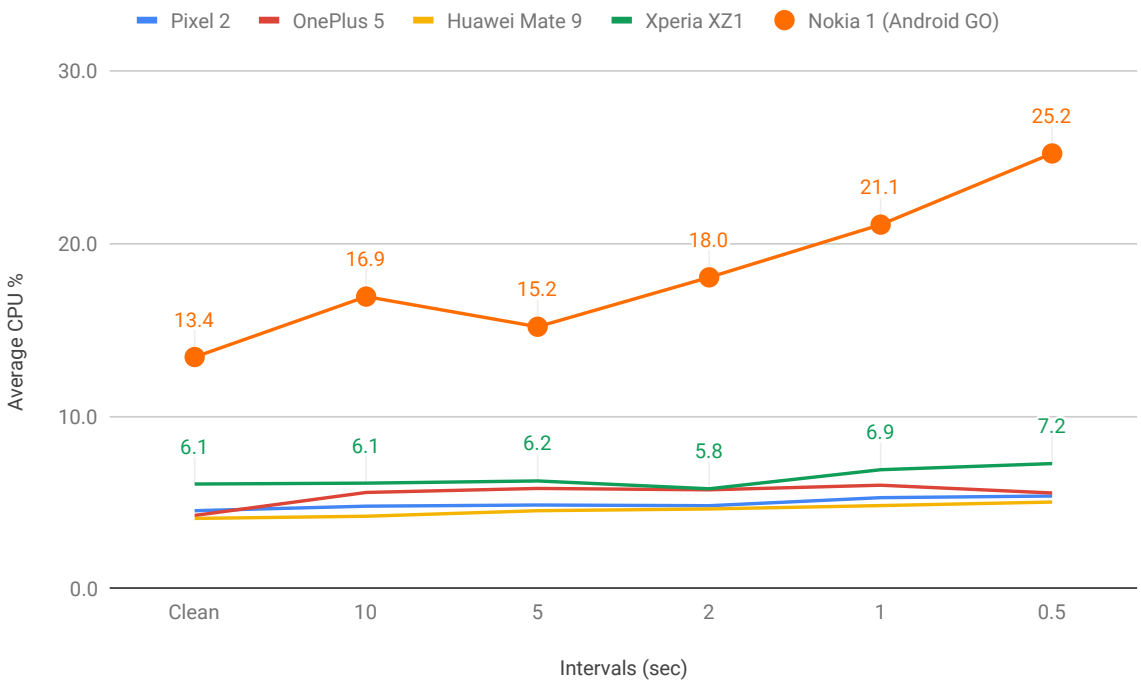

Figure 2: CPU consumption for protected apps with decreasing interval between checks.

In order to avoid this problem, we decided to change the experiment to measure network overhead. Instead of performing the checks at a constant pace, they are executed after each arithmetic computation, when the equal button of the calculator is pressed. In this way, the number of checks is independent from the device speed and the network traffic is comparable across devices. Table 8 reports the results of this last experiment. All the devices report approximately the same amount of data being received/sent during the test run, i.e. around 40 KB sent and 18 KB received.

Device	Incoming	Outgoing
Pixel 2	17.2	41.6
OnePlus 5	17.4	40.5
Huawei Mate	17.3	39.4
Xperia XZ1	17.5	40.2
Nokia 1	18.4	41.7

Table 8: Incoming/outgoing network traffic (in KB) for protected apps.

6.8 Considerations

When the security checks are performed twice per second, which means the highest possible load, the mid and high-end devices reported an increase in CPU load of less than 1%. For the low-end device, like Nokia 1, running two checks per second is not quite acceptable (CPU load increase of 12%). However, performing a check every 5 seconds leads to only 1.75% increase in the CPU load, which is acceptable. Based on these results, we can safely assume that our implementation is light-weight on most devices and it is expected not to cause a negative impact.

Considering the memory overhead due to our checks, we noticed that the worst case is for the device *Pixel 2*, where the RAM needed for the checks reached 13.5 Mb. Considering that this device has 4 GB of Ram, we conclude that the memory overhead caused by our checks is negligible. For the low-end device Nokia 1, with only 1 GB of RAM, we record less than 10 MB of memory overhead. This can be also considered a negligible and acceptable cost. So, we conclude that our implementation has a low footprint in terms of RAM consumption.

Concerning network overhead, we observed 40 KB of data being sent and 20 KB been received. Considering that in 2018 the average download speed worldwide was 21.35 Mbps and the average upload speed was 8.73 Mbps[23], the network cost of our checks is almost negligible.

All in all, we can conclude that the overall cost (including CPU, memory and network overhead) of the checks approach that we propose to detect malicious reverse engineering is small and appropriate for modern smartphones.

7 RELATED WORK

The first approaches to check for the integrity of the code under execution were based on remote attestation. One of the first and most

[23]https://www.speedtest.net/insights/blog/2018-internet-speeds-global/

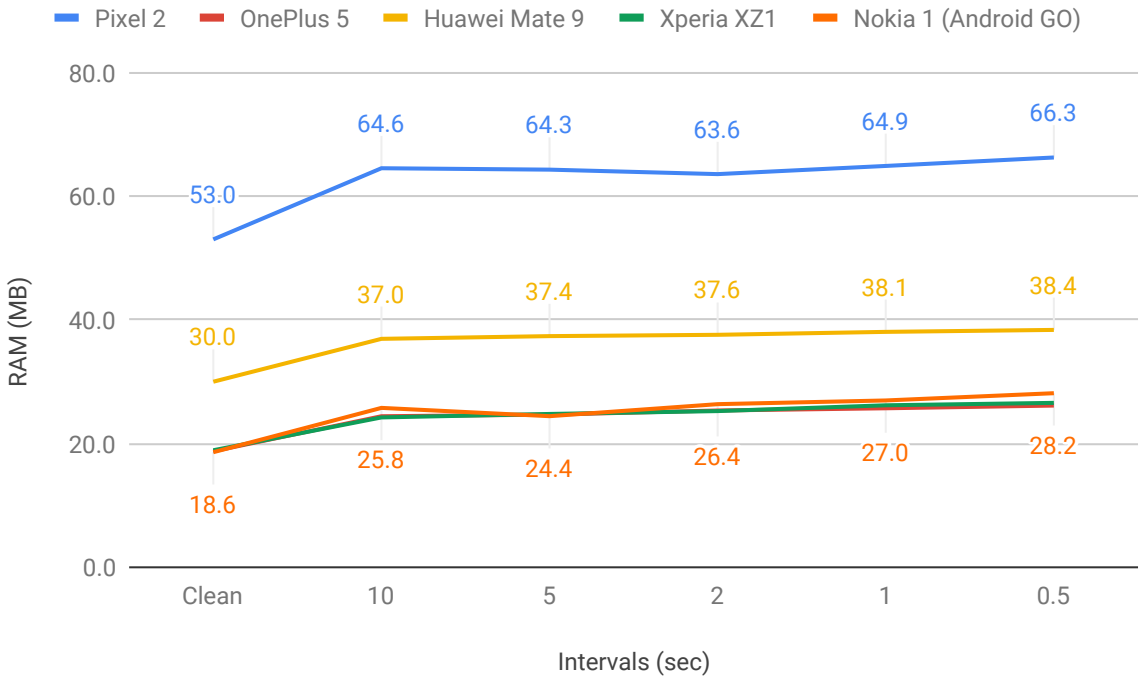

Figure 3: RAM consumption (in KB) for protected apps with decreasing interval between checks.

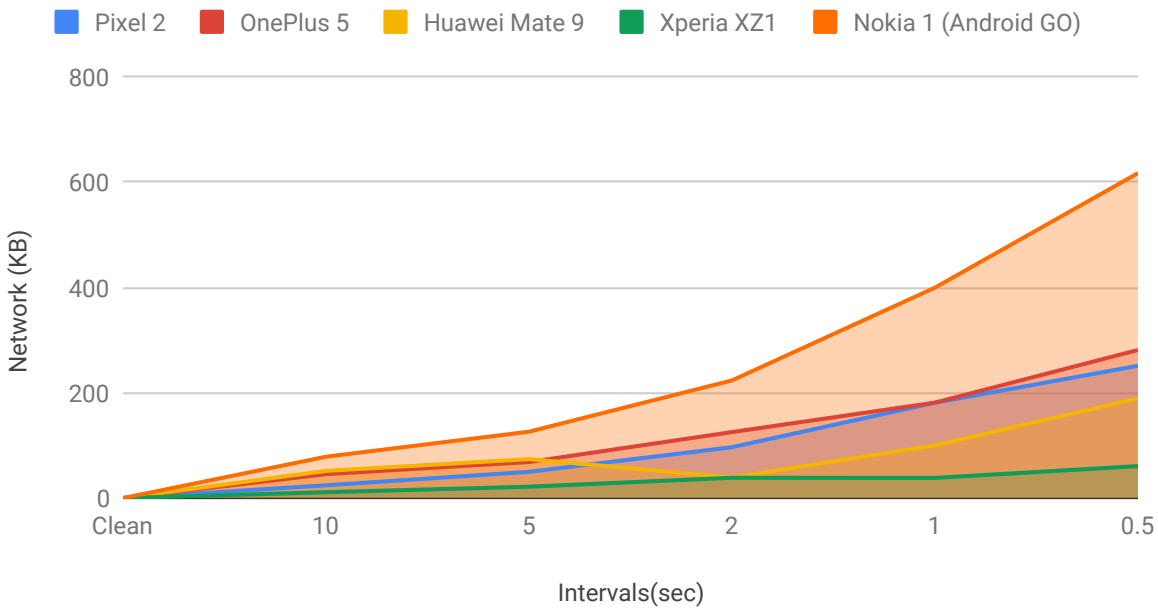

Figure 4: Incoming network traffic (measured in KB) for protected apps with decreasing interval between checks.

common applications of it is the *Integrity Measurement Architecture (IMA)* proposed by the Trusted Computing Group (TCG) [9, 17].

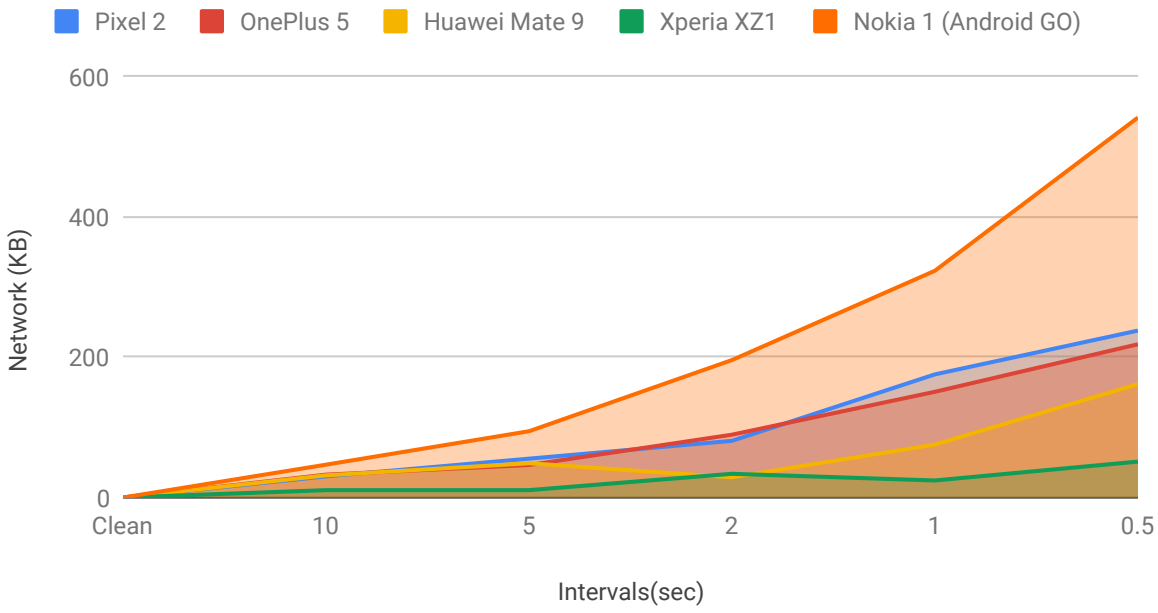

Figure 5: Outgoing network traffic (measured in KB) for protected apps with decreasing interval between checks

However, TCG realisation is based on a hardware module, the *Trusted Platform Module (TPM)*, which acts as the self-protecting trust base. As the technology evolved, the TCG approach evolved as well, thus TCG has extended the hardware-based solution to virtual and cloud-based environment [15, 18]. In general, hardware-based solutions could be used as tamper detection, but they barely apply to the mobile context where constraints on special purpose hardware can not be imposed.

Software-only mechanisms have been proposed. Armknecht et al. [1] have coined the definition of software attestation, to distinguish software-only approaches from the ones that follow the TCG approach. Software-only approaches are divided in three categories, depending on the properties used to compute and verify attestations: time-based attestation, where being able to compute an answer in time is indeed the integrity evidence; static attestation, where static properties of the application are considered, like binaries and read-only memory properties; and embryonal works that use dynamic properties to infer execution correctness.

Time-based approaches estimate a time limit within which the evidence must be produced and sent to the verifier, if the time exceeds this estimation the evidence is not accepted. Seshadri et al. [19] realized their prototype, namely Pioneer, based on time-based attestation. Integrity, with their solution, is assessed through the precise estimation of the execution time of precise code fragments executed as attestation proofs.

The earliest proposal of static attestation is the Spinellis' software reflection, which proposed the hash of random parts of the memory [21]. Similar solutions are SWATT, proposed by Seshadri et al. [20], a software-based solution that monitors target code memory regions, and MobileGuards, proposed by Grimen et al. [10], short-lived attestation agents downloaded from a trusted server. Kennel et al. [12] proposed a set of genuinity tests based on static information.

App repackaging has been identified as a serious issue. A large set of apps from multiple marketplaces have been compared to identify repackaging based on code similarity [16, 25]. This approach works when considering a large number of app for comparison and confirmed the problem of app repacking. However, this approach does fit the problem of deciding if the current app under execution has been tampered.

Countermeasures against app repacking have been proposed, based on the analysis and prof-of-concept repackaging attacks on a number of banking apps [11]. Countermeasures includes (i) self-signing restriction (requiring marketplace signature in order to publish an app) which violates the Android open policy stating that anyone may publish an app; (ii) adoption of code obfuscation to substantially increase the effort needed by an attacker to reverse engineer the app; and (iii) usage of hardware-based security solutions based on TPM, that could be embedded into a smartphone, with the overhead of additional hardware cost.

Signature checking has been also integrated with code obfuscation [14] to mitigate tampering attacks. Moreover, the app signature is verified multiple times, with different approaches and in different locations in the code, with the aim of making is very hard for an attacker to identify and defeat them. Additionally, instead of triggering a failure immediately when detecting a repackaging attack, they transmit the detection result to another node and delay the failure, thus making it even harder for an attacker to pinpoint the origin of the failure. However, they acquire the public key from

the app using the *Package Manager*, which as shown in Section 3 is quite easy to hook.

A modified kernel is proposed by Wang et al. [23], where a kernel-level Android app protection scheme is presented. This approach works as long as we assume that an attacker is not tampering with the kernel, which is not the case when tools link Magisk are deployed. All in all, this solution can not be deployed by an app developer, who can not request end-user to install specific kernel modules before downloading her/his app.

8 CONCLUSION

Malicious reverse engineering is a prominent prerequisite for attacks based on code tampering. This paper presents an approach based on local checks to identify attempts of malicious reverse engineering on Android apps. Our approach is based on (i) accurate checks on the Android execution environment to spot traces of unauthorized analysis; and (ii) consistency checks on the app code. It is important to check both of these perspectives at the same time, because each one could be exploited by an attacker to hide an ongoing analysis on the other one.

Our empirical validation suggests that our approach causes acceptable execution overhead, and that it is applicable for a wide variety of devices. The interval between two consecutive checks can be tuned to support also low-end devices with limited computational power and memory.

In our ongoing research agenda, we plan to continue investigating on this topic, first of all by accurately validating the detection capability of our approach. In particular, we plan to test it on the most advanced tools for automated analysis of Android apps. Moreover, we plan to involve expert hackers and ask them to try and subvert or workaround our checks. Based on this user study, new and more effective checks might be elaborated, the overcome the limitations of our current implementation.

ACKNOWLEDGMENTS

This work has been partially supported by activities “API Assistant/STAnD” and “Teîchos” of the action lines *Digital Infrastructure* and *Digital Finance* of the EIT Digital.

REFERENCES

[1] Frederik Armknecht, Ahmad-Reza Sadeghi, Steffen Schulz, and Christian Wachsmann. 2013. A security framework for the analysis and design of software attestation. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 1–12.

[2] Mariano Ceccato, Mila Dalla Preda, Jasvir Nagra, Christian Collberg, and Paolo Tonella. 2007. Barrier slicing for remote software trusting. In *Seventh IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2007)*. IEEE, 27–36.

[3] Mariano Ceccato, Mila Dalla Preda, Jasvir Nagra, Christian Collberg, and Paolo Tonella. 2009. Trading-off security and performance in barrier slicing for remote software entrusting. *Automated Software Engineering* 16, 2 (2009), 235–261.

[4] Mariano Ceccato, Paolo Falcarin, Alessandro Cabutto, Yosief Weldezghi Frezghi, and Cristian-Alexandru Staicu. 2016. Search based clustering for protecting software with diversified updates. In *International Symposium on Search Based Software Engineering*. Springer, 159–175.

[5] Mariano Ceccato and Paolo Tonella. 2010. Codebender: Remote software protection using orthogonal replacement. *IEEE software* 28, 2 (2010), 28–34.

[6] Mariano Ceccato, Paolo Tonella, Cataldo Basile, Bart Coppens, Bjorn De Sutter, Paolo Falcarin, and Marco Torchiano. 2017. How professional hackers understand protected code while performing attack tasks. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*. IEEE, 154–164.

[7] Mariano Ceccato, Paolo Tonella, Cataldo Basile, Paolo Falcarin, Marco Torchiano, Bart Coppens, and Bjorn De Sutter. 2019. Understanding the behaviour of hackers while performing attack tasks in a professional setting and in a public challenge. *Empirical Software Engineering* 24, 1 (01 Feb 2019), 240–286. https://doi.org/10.1007/s10664-018-9625-6

[8] Mariano Ceccato, Paolo Tonella, Mila Dalla Preda, and Anirban Majumdar. 2009. Remote software protection by orthogonal client replacement. In *Proceedings of the 2009 ACM symposium on Applied Computing*. ACM, 448–455.

[9] TT Committee et al. 2002. *Trusted computing platform alliance (TCPA) main specification v1*. Technical Report. 1b. Technical report, TCPA Alliance.

[10] Gisle Grimen, Christian Mönch, and Roger Midtstraum. 2006. Tamper Protection of Online Clients through Random Checksum Algorithms. In *Information Systems Technology and its Applications, 5th International Conference ISTA'2006, May 30-31, 2006, Klagenfurt, Austria*. 67–79. http://subs.emis.de/LNI/Proceedings/Proceedings84/article4304.html

[11] Jin-Hyuk Jung, Ju Young Kim, Hyeong-Chan Lee, and Jeong Hyun Yi. 2013. Repackaging attack on android banking applications and its countermeasures. *Wireless Personal Communications* 73, 4 (2013), 1421–1437.

[12] Rick Kennell and Leah H. Jamieson. 2003. Establishing the Genuinity of Remote Computer Systems. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12 (SSYM'03)*. USENIX Association, Berkeley, CA, USA, 21–21. http://dl.acm.org/citation.cfm?id=1251353.1251374

[13] Li Li, TegawendÃĪ BissyandÃĪ, and Jacques Klein. 2019. Rebooting Research on Detecting Repackaged Android Apps: Literature Review and Benchmark. *IEEE Transactions on Software Engineering* PP (02 2019), 1–1. https://doi.org/10.1109/TSE.2019.2901679

[14] Lannan Luo, Yu Fu, Dinghao Wu, Sencun Zhu, and Peng Liu. 2016. Repackage-proofing android apps. In *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 550–561.

[15] Ronald Perez, Reiner Sailer, Leendert van Doorn, et al. 2006. vTPM: virtualizing the trusted platform module. In *Proc. 15th Conf. on USENIX Security Symposium*. 305–320.

[16] Zhongyuan Qin, Zhongyun Yang, Yuxing Di, Qunfang Zhang, Xinshuai Zhang, and Zhiwei Zhang. 2014. Detecting repackaged android applications. In *Computer Engineering and Networking*. Springer, 1099–1107.

[17] Reiner Sailer, Xiaolan Zhang, Trent Jaeger, and Leendert Van Doorn. 2004. Design and Implementation of a TCG-based Integrity Measurement Architecture.. In *USENIX Security Symposium*, Vol. 13. 223–238.

[18] Nuno Santos, Krishna P Gummadi, and Rodrigo Rodrigues. 2009. Towards Trusted Cloud Computing. *HotCloud* 9 (2009), 3–3.

[19] Arvind Seshadri, Mark Luk, Elaine Shi, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. 2005. Pioneer: Verifying Code Integrity and Enforcing Untampered Code Execution on Legacy Systems. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles (SOSP '05)*. ACM, New York, NY, USA, 1–16. https://doi.org/10.1145/1095810.1095812

[20] Arvind Seshadri, Adrian Perrig, Leendert Van Doorn, and Pradeep Khosla. 2004. SWATT: Software-based attestation for embedded devices. In *Security and Privacy, 2004. Proceedings. 2004 IEEE Symposium on*. IEEE, 272–282.

[21] Diomidis Spinellis. 2000. Reflection As a Mechanism for Software Integrity Verification. *ACM Trans. Inf. Syst. Secur.* 3, 1 (Feb. 2000), 51–62. https://doi.org/10.1145/353323.353383

[22] Alessio Viticchié, Leonardo Regano, Cataldo Basile, Marco Torchiano, Mariano Ceccato, and Paolo Tonella. [n.d.]. Empirical assessment of the effort needed to attack programs protected with client/server code splitting. *Empirical Software Engineering* ([n. d.]), 1–48. https://doi.org/10.1007/s10664-019-09738-1

[23] Tuo Wang, Lu Liu, Chongzhi Gao, Jingjing Hu, and Jingyu Liu. 2018. Towards Android Application Protection via Kernel Extension. In *International Conference on Algorithms and Architectures for Parallel Processing*. Springer, 131–137.

[24] Wikipedia contributors. 2019. Rooting (Android) — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Rooting_(Android)&oldid=909818053 [Online; accessed 23-August-2019].

[25] Wu Zhou, Yajin Zhou, Xuxian Jiang, and Peng Ning. 2012. Detecting repackaged smartphone applications in third-party android marketplaces. In *Proceedings of the second ACM conference on Data and Application Security and Privacy*. ACM, 317–326.