A-SIT Technology

E-Government Innovation Center

13 February 2023 Analysis · Transparency · Mobile Security · 19 min read

Runtime Integrity Checks in Android Applications

Florian Draschbacher



Authors	Florian Draschbacher
Version	2023-02-08
Source	https://technology2.a-sit.at/en/publications/runtime-integrity-checks-in-android-applications/
Note	Machine translation.

Key findings

- Of 99 apps examined, only 3 (3%) contained a non-trivial integrity check; 97% were not effectively protected against repackaging.
- Given enough effort, all local repackage-proofing methods can be bypassed.
- Insurmountable protection requires remote attestation: the result goes to the backend server, which decides outside the attacker's control.
- Ibrahim et al. found the SafetyNet Attestation API in 62 of 163,773 apps; 32 of these (52%) evaluated the attestation locally.
- The Play Integrity API also permits local decryption of the token, and thus the same mistakes as the SafetyNet Attestation API.

In order to hamper reverse engineering of applications and thwart repackaging attacks, some particularly security aware Android developers integrate runtime integrity checks into their program code. These checks verify at runtime that the operating system is in an original manufacturer-supplied state (no root possibilities for the end user, original firmware image, unmodified system partition) and the application has not been manipulated between submission by the developer and installation at the user.

In this research project, we provide an overview of different possibilities for integrity checking and proofing. To this end, several proposals from science and practice are discussed. Additionally, we present a study measuring the prevalence of runtime integrity checks in 99 popular Android applications.

1. Introduction

In parallel with the widespread adoption of mobile devices such as smartphones and tablets, parties with malicious intentions also discovered the advantages of the new platforms. The ubiquity of these devices, the permanent connection to the internet and the density of sensitive user information that is processed and stored on them make mobile devices particularly attractive to attackers.

A persistent problem in the app ecosystem is repackaging. Here, a legitimate application is manipulated by an attacker and passed on to unsuspecting users. This approach can be used, for example, to carry out phishing attacks in which sensitive user data (such as login credentials or banking data) can be collected. The attack scenario is also used to execute arbitrary malicious code on the device.

As an alternative to repackaging, existing apps can also be manipulated by having an attacker control the runtime environment or the operating system. Attacks are also conceivable here, for example, in which the control flow of an application is to be changed, for instance in order to unlock functionality that is actually hidden behind a paywall.

To prevent both categories of attacks on Android, various protective measures are available that app developers can implement in order to ensure at runtime that the execution environment as well as the application code itself have not been manipulated. This research project investigates which options are available here, and to what extent they are already used by Android applications in production.

2. Background

This section explains the technologies that are necessary for understanding the later explanations.

2.1. Android

Android is the most popular operating system for mobile devices such as smartphones and tablets. The system is developed within the Android Open Source Project (AOSP) by several device manufacturers under the leadership of Google. From a technical point of view, it is an operating system based on a Linux kernel. However, the userspace components differ substantially from those of any other Linux distribution. They were developed anew specifically for the requirements of mobile devices.

2.2. Android apps

Android allows users to run arbitrary third-party applications. Google provides a Software Development Kit (SDK) with which developers can write applications in the programming languages Java, Kotlin, C and C++. After compilation, programs are packaged into an Android Package (APK) file and can be installed on an Android device. As a rule, APK files are distributed to end users via the Google Play Store. This distribution channel does not necessarily have to be used, however. In the Android system settings, the installation of APK files from arbitrary sources can be enabled.

2.3. APK file

APK files are essentially a zip archive with a clearly defined structure. Central elements within this structure are the application manifest, the DEX files, native libraries and app resources. The manifest is an XML file that describes which operating system resources the application uses and which functionality it makes available to the system or to other applications. Dalvik Executable (DEX) files contain the compiled program code that was compiled from Java or Kotlin. During compilation, this is transformed into Dalvik bytecode. This is an intermediate representation that facilitates the platform-independent execution of the code on the device. It is worth mentioning in this context that crucial high-level information about the original program code is retained in Dalvik bytecode. In contrast to programs that exist as machine code, the implemented functionality can therefore be reconstructed relatively easily here and subsequently modified. Native libraries contain the compiled program code of those parts of the application that were implemented in the native programming languages C or C++. Since they exist as CPU-specific machine code, they are considerably harder to reconstruct and manipulate.

2.4. App signing

Before an app in APK format can be installed on an Android device, it must be signed by the developer. To do so, the developer generates an asymmetric key pair consisting of a private key and a public certificate. Key and certificate are cryptographically linked in such a way that the certificate can be used to determine that the APK file was signed by the owner of the private key.

As part of the signing process, a digital fingerprint of each file in the APK file is calculated. Finally, a fingerprint of these fingerprints is encrypted with the private key, and the certificate with the public key is stored in the APK file. During the installation of the application, the operating system calculates the same fingerprints and compares the resulting fingerprint with the signature value decrypted using the public key. If they match, it can be cryptographically confirmed that the APK file was signed by the owner of the private key that matches the certificate.

In this scenario, the signature of an APK file is meant to link each application to a developer identity. Although information on the developer's name is often stored in the certificate, from a cryptographic point of view the developer identity consists only of the private key. Because the issuance of key pairs (or certificates) is handled by the developer themselves and is not centrally monitored, arbitrary information can be stored in the certificate. The app signature therefore does not allow any reliable conclusions about the legal identity of the developer.

This fact is a crucial technical foundation for repackaging attacks. Manipulated APK files must be re-signed by the attacker before they can be installed. Because the private key of the original app developer is generally not available to attackers for this purpose, they have to use a new asymmetric key pair. The resulting application can nevertheless be installed without any problems. For an averagely skilled user, it is not apparent that the application has been manipulated or equipped with a certificate that differs from the original.

2.5. Rooting and hooking

Since Android is based on the Linux kernel, there is also a root user here who, from the point of view of Discretionary Access Control (DAC), is equipped with every possible permission with regard to resource access (for example to files). As a rule, however, the Android system is configured in such a way that the end user cannot run arbitrary software with root privileges. On most Android devices, however, it is possible to obtain root access, either deliberately by the end user (“rooting”) or by exploiting vulnerabilities. Among other things, it is thus also possible to replace system components or to manipulate the memory contents of arbitrary processes.

When an Android application is executed, its compiled code is generally not executed directly on the CPU, but within the Android Runtime (ART). This approach allows Android applications to be executed on devices with any CPU architecture. However, it also enables attackers to manipulate the execution of every installed Android application by replacing a few system components. If the execution of a particular function is intercepted in a targeted way, this is called “hooking”.

Because rooting and hooking make it possible to manipulate the control flow of Android applications to a similar extent as repackaging, some developers of particularly security-sensitive applications integrate runtime integrity checks. If a manipulated runtime environment is detected (for example the presence of rooting tools or hooking solutions), the application refuses to operate.

3. Local solutions

This section discusses how repackaging can be detected and prevented without the help of external parties.

3.1. Centralised repackage detection

The simplest way to detect repackaging is static solutions that, by comparing a known original application A and a package B of unclear provenance, can determine whether B was created by modifying A. Such a solution is well suited, for example, for integration into app stores, where it can be ensured for every app newly submitted by developers that it is their own work. Detected cases of repackaging can thus be rejected at upload.

However, this technology cannot prevent legitimate original applications from being downloaded from these app stores, modified, re-signed and passed on via other channels. As mentioned above, apps that were downloaded, for example, from arbitrary websites can also be installed on Android. There are also websites, for example, whose explicit purpose is the illegal exchange of applications that originally require payment.

3.2. Dynamic repackage proofing

To prevent the distribution of modified applications via channels outside the established app stores as well, some developers rely on dynamic repackage detection, which they can integrate directly into their program code. The goal of this dynamic check is generally to refuse the functionality of the app when a modification is detected. This is called repackage proofing, because the app is to be protected against repackaging.

The Application Programming Interface (API) that Android developers can access offers, in the `PackageManager` class with the method `getPackageInfo()`, a way to obtain information (among other things) about the APK file of the currently running application. As part of this information, the signing certificate can also be read out, which was already verified by the system at installation. A simple way of performing a runtime integrity check therefore consists of comparing the value delivered by this interface with the statically stored correct signing certificate. While this method works in principle, in practice it does not protect effectively against repackaging. As part of their modifications, attackers can simply remove the checking code entirely, intercept the method call and return the certificate of the original APK, or replace the stored original reference certificate.

Because of the known weaknesses of this approach, but the conceptual lack of alternatives for local solutions in some deployment scenarios, a whole series of scientific publications has investigated how the check can be secured. The goal here is to make the check harder to locate, manipulate or remove without destroying the existing functionality of the application.

One way to make locating and manipulating more difficult is the use of native code. The essential difficulty for attackers arises here from the fact that, after compilation, native code exists only as machine code, which contains considerably less high-level information than Dalvik bytecode, making the original code harder to follow. The use of native code is often combined with further obfuscation mechanisms. Tanner et al. [1] combine the approach of multi-layered encryption of Dalvik bytecode and native code. Protsenko et al. [2] also follow a similar approach.

Encryption of bytecode is also used by some other researchers. Zeng et al. [3] use the technique to conceal logic bombs that are woven into the core functionality of the app and perform a signature check only while the app is being used by real end users. The key can only be derived dynamically at runtime if no modifications are present. Ren et al. [4] combine self-decrypting code with watermarking.

It should be noted for all of these solutions that their protection against attackers who want to remove the signature checks lies merely in obfuscation. Attackers are thus made to work harder to understand the implemented functionality from the compiled code. However, because the signature checks generally have to use certain Android APIs, for example to locate the APK file in the file system or to read out the signature, in many cases the exact mode of operation of protection mechanisms does not even need to be known in order to circumvent them. Ma et al. [5] show that it is often sufficient to manipulate the runtime environment of Android apps in a targeted way in order to intercept crucial Android API methods and thus simulate an unmanipulated environment to the repackaged app. Intercepting these methods can be implemented either via system-wide hooking using root privileges, or by injecting a native library into the app that manipulates the ART runtime at app start.

In general, it can be stated that, with sufficient effort, all local repackage-proofing methods can be circumvented. It can be assumed that for every obfuscation approach a corresponding static or dynamic analysis tool can be built that can reverse the obfuscation mechanisms and thus also remove the signature checks.

3.3. Remote code attestation

To achieve insurmountable protection against repackaging, an independent external entity is needed alongside the application itself, whose functionality is not implemented in the target package itself and which is executed separately.

On many current Android devices, such an independent external entity can even be implemented locally if use is made of the Trusted Execution Environment (TEE; e.g. ARM TrustZone). Here, the CPU provides a secure execution environment whose state is completely decoupled from the main execution environment. Only the secure execution environment can determine to what extent a data exchange takes place when required. Demesie et al. [6] present an approach that implements protection against repackaging by means of a TEE.

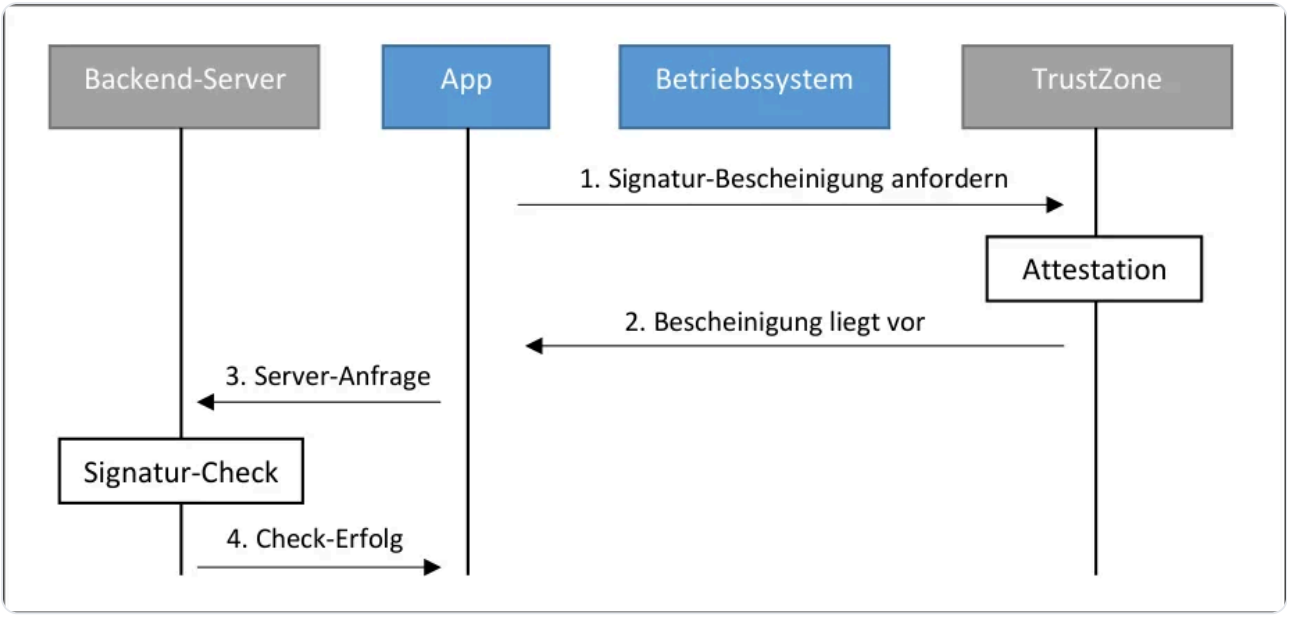


Figure 1: Schematic representation of simple remote attestation

While the verification of the integrity of the runtime environment and the application can be implemented locally and securely by means of a TEE, insurmountable protection against repackaging is nevertheless only possible for applications that have to communicate with a server backend for their core functionality. If the app alone decides locally how to proceed in the event of a negative integrity check, an attacker can manipulate this behaviour even with an external integrity check. If, however, the result of the integrity check is transmitted as part of each request to the backend server, the backend can decide, outside the attacker's sphere of control, whether the request is to be answered or the communication is to be terminated.

Based on this conceptual structure, a few further considerations are crucial for the actual security of the implementation. The signature attestation needs to be protected against manipulation by the potentially attacked app while it is being passed on to the backend server. Asymmetric cryptography can be used for this. The attestation is signed or encrypted by the TEE component and verified or decrypted at the backend server. The signed (or encrypted) attestation is now also called attestation, and the entire process remote attestation. To prevent replay attacks, a nonce, that is, a unique value, must be included in the attestation.

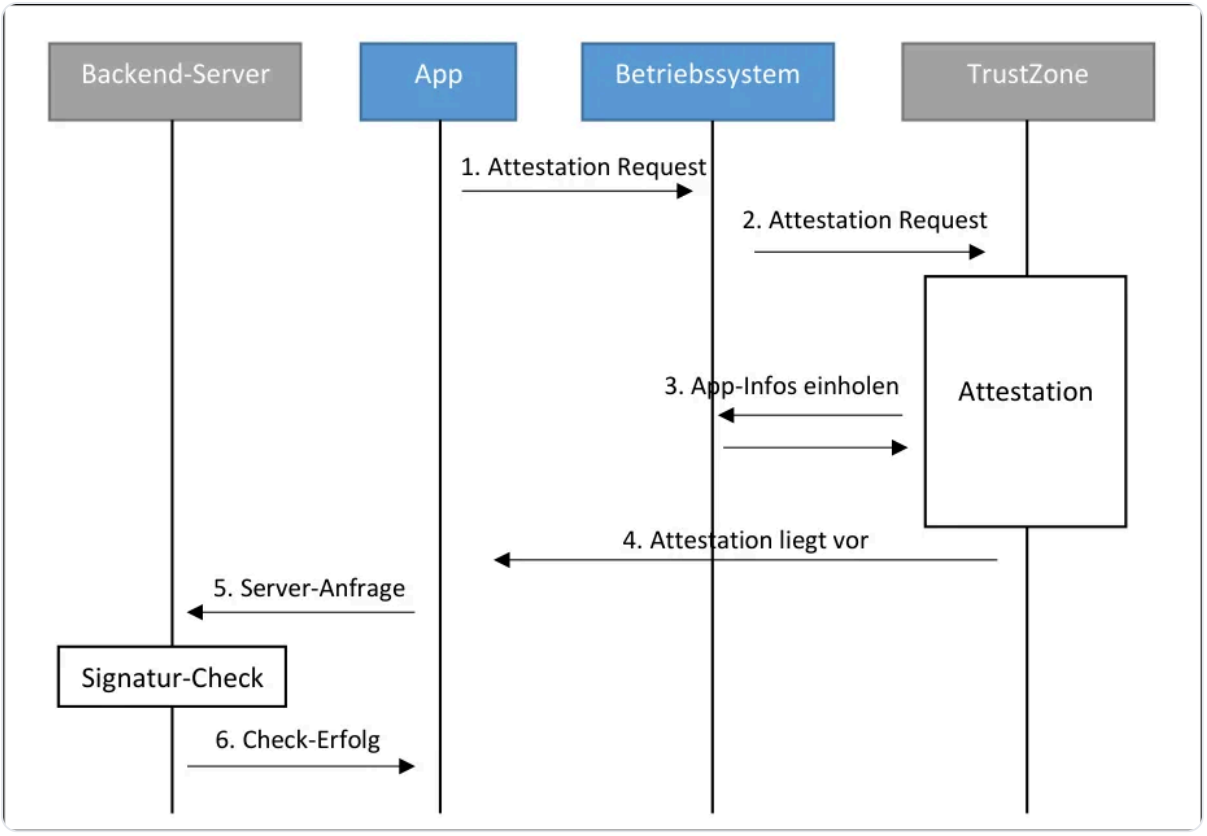


Figure 2: Remote attestation including operating system interaction

Finally, the role of the operating system also deserves closer consideration. As shown in Figure 2, the TEE component relies on the operating system to obtain app information such as the package name of the running application and the matching signature information. If an attacker is able to influence the communication between the operating system and the TEE component, the entire mechanism can be undermined. To prevent this, information on the integrity of the operating system must also be obtained as part of the attestation. This makes it apparent why the integrity checking of applications cannot be considered separately from the integrity checking of the operating system. Prünster et al. [7] were the first to show how the continuous integrity of bootloader, system and application can be guaranteed with Key Attestation, which has been available since Android 8.0.



shows the basic flow that such an implementation follows. At app start or before critical processing steps (for example before submitting a high score), the app requests a signature attestation from a corresponding component in the Trusted Execution Environment. The result is then transmitted to the backend server together with the request, where the attestation is evaluated. The request is only answered if the attested signature certificate matches the original signature certificate.

3.4. Google SafetyNet

To provide app developers with an effective tool against repackaging and similar attacks on application integrity, Google has for several years offered a solution called the SafetyNet Attestation API[1]. This works essentially like the solution proposed by Prünster et al [7]. To issue the attestation, the application contacts the Google Play Services, which in turn involve a corresponding Google server for this purpose that signs the attestation. Google Play Services then forward the response to the app, which transmits it to its backend server. The detailed flow can be seen in Figure 3.

The exact role of the Attestation API backend is not documented by Google. It is unclear why the attestation mechanism of Key Attestation, whose signature can be issued locally (in the TEE), is not used here as proposed by Prünster et al. [7].

Figure 3 Google SafetyNet Remote Attestation (Source: Android Developers Documentation)[2]

Ibrahim et al. [8] found that of 163773 examined applications, only 62 (0.04 %) used the SafetyNet Attestation API. Of these 62 applications, however, 32 (52 %) evaluated the attestation locally (that is, did not forward it to a backend server), as a result of which the complete attestation process could easily be removed from the application by an attacker.

Presumably in response to the frequent incorrect use of the SafetyNet Attestation API, Google introduced its successor in the form of the Play Integrity APIs in June 2022. In contrast to the previous solution, the result of the API is now an encrypted token that, in the standard use case, is transmitted by the backend server to a Google server for decryption. The flow is shown in Figure 4. Although according to the official documentation validating the attestation locally on the device is no longer possible, this does not correspond to the technical facts. It is still technically possible to call the Google server for decrypting the token from within the app.

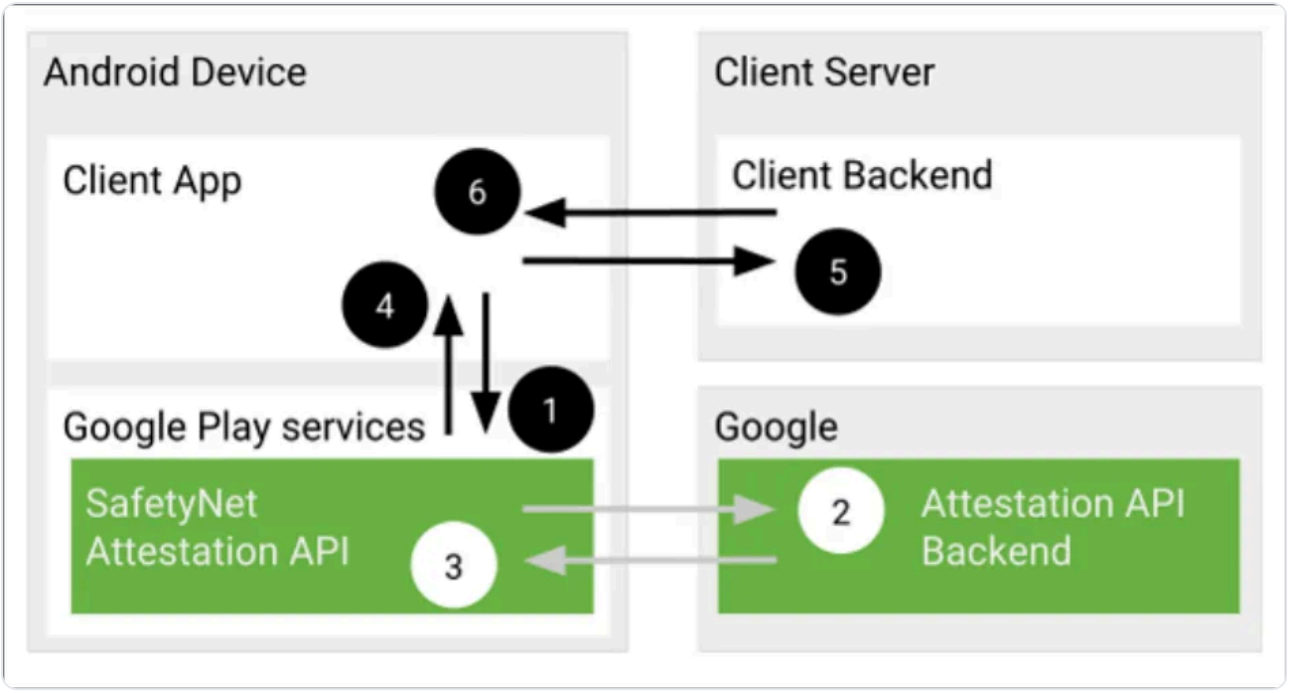


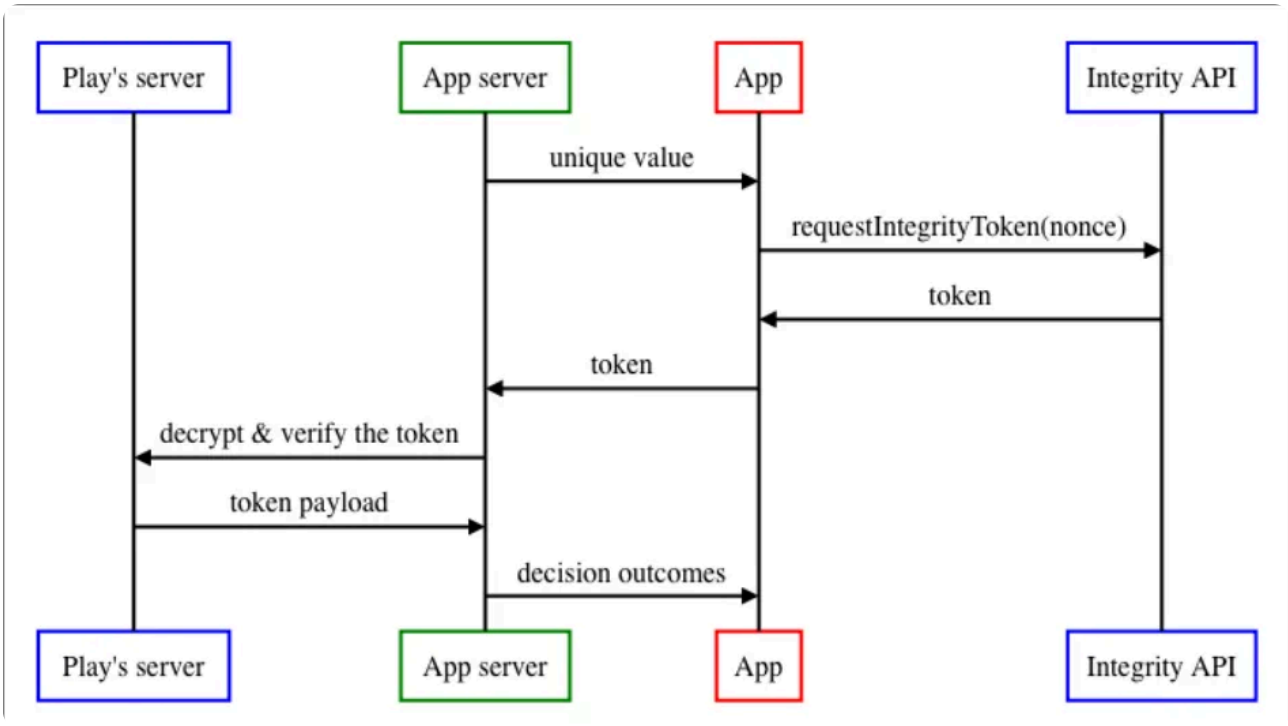
Figure 4: Google Play Integrity API Flow (Source: Android Developers Documentation)[^4]

In addition, developers can request the AES key for token decoding from Google, so that tokens can also be decrypted locally. Developers are thus able to decrypt and validate the token directly in the application code as well. Although less apparent from the documentation, it is therefore still possible to make the same mistakes as with the older SafetyNet Attestation API. In fact, example projects that use the insecure local decryption and validation can already be found on the internet[4].

4. Prevalence of runtime integrity checks

In the recent past, several scientific studies have investigated the prevalence of runtime integrity checks in Android applications. Berlato et al. [9] concluded in 2019 that, of the 23610 most popular applications in the Play Store, 88.8 % implemented signature checks. According to Berlato et al., about 11 % of the applications used SafetyNet Attestation. It must be mentioned here, however, that the static fingerprinting used for the analysis has weaknesses. For example, it was not analysed whether an implemented measure is actually executed at runtime. For the analysis of SafetyNet usage, no distinction was made as to whether only the device environment or also the application itself was checked. For signature checks, it was not determined whether repackage proofing was implemented (whether a negative result leads to a refusal of functionality).

To obtain additional data from practice, a study of its own on the prevalence of runtime integrity checks in popular Android applications was carried out as part of this project.



4.1. Dataset

For the analysis, the 3 most popular free applications each were taken from 33 categories in the Google Play Store. If an application required a paid or personalised registration, the next application in the ranking was used instead.

4.2. Test setup and methodology

For the analysis, each application was subjected to a repackaging attack. It was decompiled so that accesses to the PackageManager API for certificate queries were answered with the original signing certificate. This ensured that only apps that contain no trivial integrity checks were captured. The recompiled APK was then signed with a newly generated certificate and installed on a Google Pixel 3 running Android 11 (build RQ3A.210605.005). A human user then navigated through the user interface of the app for 3 minutes, following the realistic user flow of an actual user. If the user noticed an error message relating to the integrity of the app, it was assumed that a non-trivial signature check had been implemented.

4.3. Result

Of the 99 examined applications, a non-trivial integrity check was identified in only 3 applications (3 %). It can be inferred that 97 % of the tested applications were not effectively protected against repackaging. No statement can be made about the actual level of security of the 3 applications with a non-trivial integrity check.

5. Summary

In this report, we examined the problem of app repackaging in the Android ecosystem. In addition to a description of the technical starting position, various options were explained with which apps can check the integrity of the runtime and of their program code. After a conceptual discussion of an insurmountable implementation for remote attestation, a study on the prevalence of repackage proofing in current popular Android applications was presented.

References

[1] S. Tanner, I. Vogels und R. Wattenhofer, „Protecting Android Apps from Repackaging Using Native Code," in FPS 2019: Foundations and Practice of Security, 2019.

[2] M. Protsenko, S. Kreuter und T. Müller, „Dynamic Self-Protection and Tamperproofing for Android Apps Using Native Code," in International Conference on Availability, Reliability and Security, ARES, 2015.

[3] Q. Zeng, L. Luo, X. Du, Z. Li, C.-T. Huang und C. Farkas, „Resilient User-Side Android Application Repackaging and Tampering Detection Using Cryptographically Obfuscated Logic Bombs," IEEE Transactions on Dependable and Secure Computing, Bd. 18, Nr. 6, pp. 2582 - 2600, 2019.

[4] C. Ren, K. Chen und P. Liu, „Droidmarking: resilient software watermarking for impeding android application repackaging," in Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering, Vasteras, 2014.

[5] H. Ma, S. Li, D. Gao, D. Wu, Q. Jia und C. Jia, „Active Warden Attack: On the (In)Effectiveness of Android App Repackage-Proofing," IEEE Transactions on Dependable and Secure Computing, Bd. 19, Nr. 5, pp. 3508 - 3520, 2021.

[6] S. D. Yalew, P. Mendonca, G. Maguire, S. Haridi und M. Correira, „TruApp: A TrustZone-based authenticity detection service for mobile apps," in IEEE International Conference on Wireless and Mobile Computing, Networking And Communications (WiMob), Rome, 2017.

[7] B. Prünster, G. Palfinger und C. Kollmann, „Fides: Unleashing the Full Potential of Remote Attestation," in Proceedings of the 16th International Joint Conference on e-Business and Telecommunications, 2019.

[8] M. Ibrahim, A. Imran und A. Bianchi, „SafetyNOT: on the usage of the SafetyNet attestation API in Android," in Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services, 2021.

[9] S. Berlato und M. Ceccato, „A large-scale study on the adoption of anti-debugging and antitampering protections in android apps," Journal of Information Security and Applications, Bd. 52, 2020.

1. Android Developers Documentation: SafetyNet Attestation API. https://developer.android.com/training/safetynet/attestation ↩

2. Portions of this page are reproduced from work created and shared by the Android Open Source Project and used according to terms described in the Creative Commons 2.5 Attribution License. ↩

3. Portions of this page are reproduced from work created and shared by the Android Open Source Project and used according to terms described in the Creative Commons 2.5 Attribution License. ↩

4. GitHub: Google Play Integrity Demo. https://github.com/chonkyboi-simon/google-play-integrity-demo ↩