On the (In)Security of Manufacturer-provided Remote Attestation Frameworks in Android

Ziyi Zhou, Xuangan Xiao, Tianxiao Hou, Yikun Hu*, and Dawu Gu*

Shanghai Jiao Tong University, Shanghai, China
{jou.dzyi,xgxiao,tomhou2002,yikunh,dwgu}@sjtu.edu.cn
∗ Yikun Hu and Dawu Gu are the corresponding authors.

Abstract. To provide a tamper-proof mechanism for mobile apps to check the integrity of the device and their own code/data, Android phone manufacturers have introduced Manufacturer-provided Android Remote Attestation (MARA) frameworks. The MARA framework helps an app conduct a series of integrity checks, signs the check results, and sends them to remote servers for a remote attestation. Nonetheless, we observe that real-world MARA frameworks often adopt two implementations of integrity check (hardware-based and software-based) for compatibility consideration, and this allows an attacker to easily conduct a downgrade attack to force the app to utilize the software-based integrity check and forge checking results, even if the Android device is able to employ hardware-supported remote attestation securely. We demonstrate our MARA bypass approach against MARA frameworks (i.e., Google SafetyNet and Huawei SafetyDetect) on real Android devices, and design an automated measurement pipeline to analyze 35,245 popular Android apps, successfully attacking all 104 apps that use these MARA services, including well-known apps and games such as TikTok Lite, Huawei Wallet, and Pokémon GO. Our study reveals the significant risks against MARA frameworks in use.

Keywords: Remote Attestation · Android Device Integrity · Android App Integrity.

1 Introduction

On the Android platform, ensuring the integrity of the device environment is crucial for app developers. For instance, most payment and banking apps examine whether the runtime environment is tampered (e.g., the device is "rooted"). If so, they refuse to run, in order to reduce the potential risks to users' property and privacy [3,4]. Many game apps also need root detection to prevent game cheating [1,2]. Moreover, some mobile advertisers would like to verify whether ad clicks are coming from real devices to prevent click fraud [5]. In addition to device integrity, many app developers also hope to verify the app integrity to prevent app repackaging which may cause intellectual property infringement or ad insertion [6,8].

It is, however, difficult to implement a secure integrity check by only utilizing the app client. Malicious users could easily modify or disable the integrity
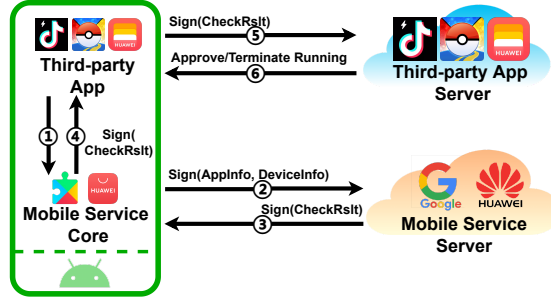
Fig. 1: Workflow of MARA.

check functions to circumvent such defenses. To guarantee a tamper-proof integrity check for Android apps, Android phone manufacturers have implemented integrity check frameworks at the OS level and provided easy-to-use APIs for apps to retrieve integrity check results. We refer to these integrity check frameworks as **Manufacturer-provided Android Remote Attestation** (MARA) frameworks as they have adopted the Remote Attestation (RA) [30] scheme to prevent check results from being tampered or forged. Two commercially available MARA framework implementations, Google SafetyNet [24] and Huawei SafetyDetect [25] are widely used in Android phones sold in the United States and mainland China, respectively. Such MARA frameworks are often parts of the Mobile Service (MS) Cores, that is, the Google Mobile Service Core (GMS Core) [12] and the Huawei Mobile Service Core (HMS Core) [14].

The overview of the MARA's workflow is shown in Figure 1. When an app calls the MARA API provided by the MS Core, MS Core collects information about the device and the app, signs and then sends these information to the Mobile Service server, and ultimately returns an integrity check result signed by the MS server. If the result indicates, for example, that the device is an emulator instead of a physical device, the app server will return specific commands to the app (e.g., asking the app to terminate execution).

To the best of our knowledge, no previous work has comprehensively and accurately evaluated the security of popular MARA frameworks. Aldoseri *et al.* [33] theoretically analyzed the security of some MARA protocols. They, however, only focused on the high-level design and did not analyze the actually implemented version of the deployed MARA framework. We observe that many of the implementation details **did not** fully follow the protocol specification, as shown in Section 3.2, and this would lead to severe security violations even though the original protocol has been verified. Some researches [26,27,29] focused on the internal mechanism of Google SafetyNet, but they did not conduct a security analysis and did not discuss other MARA frameworks. Other researches [28,32] detected the misuse of Google SafetyNet API in Android apps and did not analyze the security of the MARA frameworks themselves.

Our Work. Despite all previous research efforts, we argue that the security of MARA frameworks remains an unanswered research question. In this paper, we comprehensively analyze the security status of two mainstream MARA services

– Google SafetyNet and Huawei SafetyDetect (SafetyNet and SafetyDetect in short, respectively).

We first conduct a thorough reverse engineering to recover the underlying mechanisms of both SafetyNet and SafetyDetect. We find that their attestation protocols follow a similar weakness: the MS Core signs the device and app information, and sends the signed information to the MS Server. In this process, MARA allows for two signing modes of MS Core. If the device supports Trust-Zone, then a hardware-backed KeyStore [23] will be employed during the signing process; otherwise, the signing is entirely executed by the software-level MS Core. We refer to the check result in these two signing modes as *hardware-based check result* and *software-based check result*, respectively. A severe security vulnerability here is that **the hardware-based integrity check cannot be enforced**. That is, most app servers unconditionally accept a software-based check result, even if the device supports hardware-based integrity check. If we deceive the app server that the current device does not support hardware-based attestation, then a software-based check result is accepted, regardless of the actual situation. Since **the software-based signing process can be emulated**, if the MARA protocol is determined, attackers can impersonate MS Core to sign arbitrary messages and cheat the MS Server to return a benign result to both the caller app and the app server. This has finally led to a decrease in the security of the MARA mechanism.

We design an automated measurement pipeline and utilize it to analyze 35,245 top popular apps, confirming that 104 apps, including popular apps and games such as TikTok Lite, Huawei Wallet, Pokémon GO, and NBA LIVE, use the MARA service. Our test shows that there does exist **a generic downgrade attack to bypass MARA**: our software-emulated bypassing approach succeeds against all these apps, which demonstrates the insecurity of the widely used MARA protection.

In a nutshell, this paper makes the following contributions:

- An in-depth reverse engineering to reveal the underlying mechanism of typical MARA frameworks.
- The identification of a common design flaw in these frameworks, which can be exploited to bypass MARA protection.
- A large-scale measurement on real-world apps to evaluate the associated security risks.

Ethical Considerations. We have reported the issue of Google SafetyNet to Google Bug Hunters [36] in May 2022, and Google Security Team has filed the bug based on our report. For the issue of Huawei SafetyDetect, we have reported it to CNCERT/CC[1], and CNCERT/CC has verified our findings and documented related vulnerability under CNVD-2023-57655.

---

[1] National Computer Network Emergency Response Technical Team/Coordination Center of China, the national CERT of China and responsible for handling severe cyber-security incidents [35].

2   Background

2.1   Remote Attestation

Remote Attestation (RA) [30,31] is a mechanism to verify the integrity and trustworthiness of remote computing devices or systems. A typical remote attestation procedure [30] involves three main parties: the **Attester** is the device or system being attested and generates believable information about itself ("Evidence"); the **Verifier** verifies the Evidence and generates the "Attestation Result"; the **Relying Party** makes the final decision based on the Attestation Result from the Verifier.

On Android, there are several manufacturer-developed implementations for remote attestation. We call them Manufacturer-provided Android Remote Attestation (MARA) frameworks. In these MARA frameworks, mobile devices and their apps typically act as the Attester, the manufacturer's Server usually acts as the Verifier, and the Relying Party is the App Server.

2.2   Mobile Service Core

MARA frameworks are often implemented in the Mobile Service Cores (MS Cores), such as Google Mobile Service Core (GMS Core) [12] and Huawei Mobile Service Core (HMS Core) [14]. These MS Cores are usually integrated into the OS. The phones produced by Google, Huawei and their licensed co-operators [15,16] are pre-installed with MS Cores since shipped from the factory. For other phones, phone users can also easily install MS Cores by themselves. Since apps like Google Play Store [17], YouTube [18], and Huawei Health [20] can only be used after MS Core is installed, MS Cores have over billions of users [22] and cover almost all countries around the world [21]. This has expanded the impact of security issues within GMS Core and HMS Core.

2.3   Integrity on Android

On Android, integrity protection mainly concerns three aspects:
(1) Device Integrity. Device integrity refers to whether the software and hardware environment of a device has not been unauthorizedly tampered with. Specifically for Android, actions that compromise device integrity include rooting, unlocking the bootloader, changing the SELinux status, using emulators to impersonate a device, and so on. Since running the app on compromised devices can adversely affect the app's service [1,2,3,4,5], ensuring the device integrity is necessary for app developers.
(2) App Integrity. On Android, app integrity mainly refers to whether the app is identical to its official version. The most typical attack behavior that compromises app integrity is app repackaging, which can bring many security risks [6,7]. The repackaged app differs from the original app in aspects such as the APK file digest and app signing certificate fingerprint [48].
(3) Data Integrity. The integrity of certain sensitive data (such as a gamer's score) transmitted from mobile apps to app servers needs to be safeguarded, as
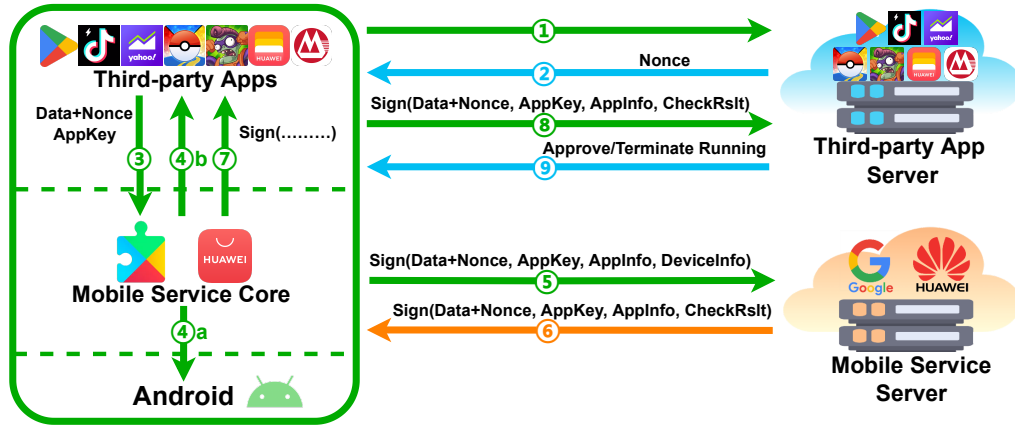
Fig. 2: Key design of the MARA scheme.

attackers may attempt to tamper with these data through network man-in-the-middle (MITM) attacks.

3   MARA Frameworks Demystification

In this section, we introduce the underlying mechanism of two typical MARA frameworks, namely Google SafetyNet and Huawei SafetyDetect. Through reverse engineering, we found that their attestation protocols followed a similar scheme, as described in Section 3.1. In Section 3.2 and 3.3, we introduce the detailed attestation protocols of SafetyNet and SafetyDetect respectively.

3.1   MARA Scheme

Key Design Figure 2 presents the high-level design of MARA schemes.

First, the Third-party App on the user's mobile phone obtains a *Nonce* from the App Server (Step ① and ②). Then, the App calls the MARA API, together with *Data+Nonce* and *AppKey* as the parameters (Step ③). Here, *Data* refers to the sensitive data that the App wants to protect from being tampered with, such as the gamer's score. The *Data* will be concatenated with the *Nonce*. *App-Key* is exclusive to a specific app and is pre-assigned to app developers by the manufacturer. It is a fixed string and is usually hard-coded in the App.

Following this, the Mobile Service Core will retrieve the properties of the device (Step ④a) and the App (Step ④b) to get the *DeviceInfo* and the *AppInfo*. Afterwards, the *DeviceInfo* and *AppInfo*, along with the *Data+Nonce* and *App-Key*, will be signed and sent to the Mobile Service Server (Step ⑤). The signing process typically involves a signing key that has been pre-negotiated between the Mobile Service Core and the Mobile Service Server. After verifying the signature, Mobile Service Server will check the integrity of the device based on the *Device-Info* and generate *CheckRslt*. Mobile Service Server will then sign the *CheckRslt*, along with the *Data+Nonce*, *AppKey*, and *AppInfo*, using its private key. This signed message will ultimately be obtained by the App Server (Step ⑥, ⑦ and ⑧).

App Server need to verify the Mobile Service Server's signature and check the *Nonce*. Finally, based on the integrity of the device and app, App Server needs to make decisions (Step ⑨). Device integrity can be retrieved directly from the *CheckRslt* and app integrity needs to be judged by comparing *AppInfo* with that of the legitimate app client.

Integrity Protection Through the above process, MARA Service can provide caller app with integrity protection in three aspects:

(1) Device Integrity Check. Third-party App Server can judge the integrity of the device based on the *CheckRslt* received in Step ⑧. Such a check aims at avoiding the risk of running on a compromised device (e.g., a rooted device or a device whose bootloader is unlocked). The main checking items for device integrity are listed in Appendix A.1.

(2) App Integrity Check. The Third-party App Server receives the *AppInfo* in Step ⑧. Mobile Service Server's signature ensures that *AppInfo* has not been tampered with. Such a check primarily aims at preventing app repackaging [6]. Checking items for app integrity are listed in Appendix A.2.

(3) Data Integrity Protection. In some cases, attackers may tamper with sensitive data sent by the app client through a network man-in-the-middle (MITM) attack. To mitigate this situation, in Step ③, the *Data* is sent to Third-party App Server after being signed. Thus, cheaters can no longer directly tamper with the *Data*.

3.2   Details about SafetyNet

To utilize SafetyNet service, the Third-party App need to integrate the SafetyNet SDK developed by Google. Figure 3 shows the protocol flow of the SafetyNet attestation step-by-step. The whole process can be divided into three phases:

(1) Initialize. This phase corresponds to Step ① to Step ③ in Figure 2. In this phase, Third-party App passes *Data+Nonce* to GMS Core through SafetyNet SDK.

(2) Obtain Signed Check Result. This phase corresponds to Step ④ to Step ⑥ in Figure 2. Through this phase, GMS Core will obtain the *CheckRslt* signed by GMS Server.

Specifically, after receiving the App's invocation, GMS Core will download the program file of a customized virtual machine (VM) (Step 5 to 6), and launch the VM) (Step 7). The *VM Program* is an APK-compressed file named "the.apk". It will be saved in GMS Core's private directory. After launched, VM will download customized VM bytecode from GMS Server (Step 8 to 9) and retrieve some device properties through executing these bytecode (Step 10). Finally, VM will get *DeviceInfo$_2$*.

On the other hand, GMS Core will also retrieve some other device properties and get *DeviceInfo$_1$* (Step 10). Afterwards, GMS Core will calculate the SHA-256 value of a protobuf [66] message containing *Data+Nonce*, *AppInfo*, *DeviceInfo$_1$* and *TimeStamp* to get *Digest* and send *Digest* to VM (Step 11). The VM will sign *Digest* and *DeviceInfo$_2$* using *VK* (short for "VM Key"). This *VK* is embedded in
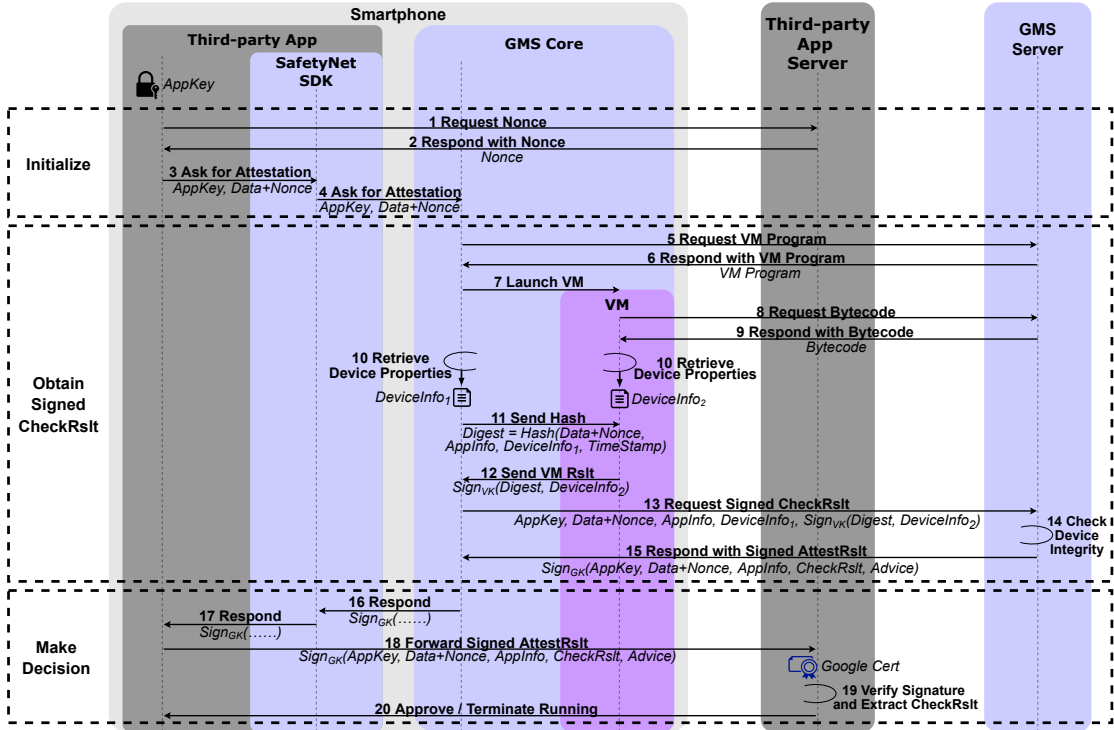
Fig. 3: The protocol flow of SafetyNet attestation.

the VM bytecode and the signing algorithm is HS256. We suppose that the reason for carrying out the signing process within VM is because VM offers a higher level of security. Through signing *Digest*, the device properties retrieved by GMS Core and VM, namely *DeviceInfo₁* and *DeviceInfo₂*, both get authenticated.

The VM-signed message will be forwarded to GMS Server in request for *CheckRslt* (Step 13). GMS Server will check device integrity based on *DeviceInfo₁* and *DeviceInfo₂* (Step 14), and respond with the signed *CheckRslt* and *Advice*. *Advice* is a string explaining why the SafetyNet attestation fails, such as "LOCK_BOOTLOADER,RESTORE_TO_FACTORY_ROM" (Step 15).

(3) Make Decision. This phase corresponds to Step ⑦ to Step ⑨ in Figure 2. In this phase, the signed *CheckRslt* will be forwarded to Third-party App Server (Step 16 to 18). App Server will verify the signature using Google's public key [38], and make decisions based on *CheckRslt*, *AppInfo*, and *Nonce* (Step 19 to 20).

3.3 Details about SafetyDetect

Similarly, the Third-party App needs to integrate the SafetyDetect SDK. Figure 4 shows the protocol flow of the SafetyDetect attestation.

(1) Initialize. This phase has no significant difference from the initialization phase of SafetyNet.
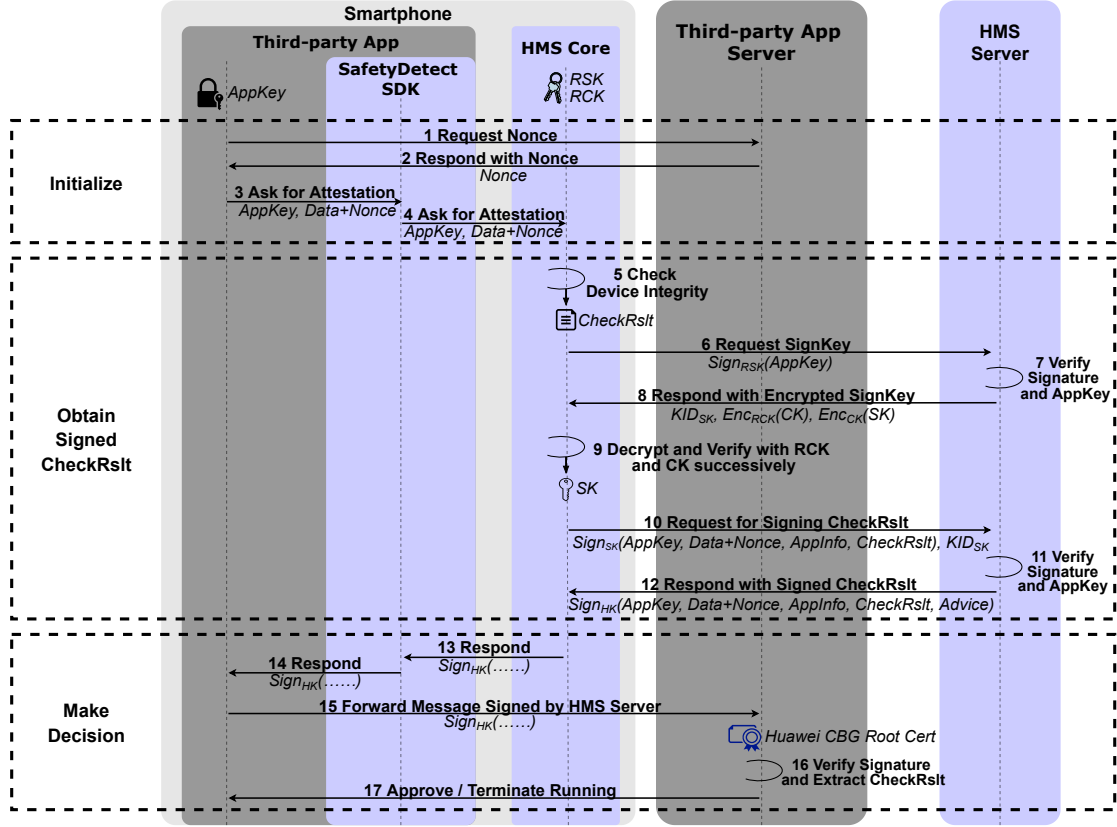
Fig. 4: The protocol flow of SafetyDetect attestation.

(2) Obtain Signed Check Result. Similar to the corresponding phase of Safe-tyNet, HMS Core will obtain the *CheckRslt* signed by HMS Server through this phase.

Specifically, HMS Core will retrieve device properties, judge whether the device has been compromised, and generate *CheckRslt* (Step 5). Afterwards, by proceeding from Step 6 to Step 9, HMS Core will obtain a *SK* (short for "SignKey") from the HMS Server. This *SK* will later be used to sign *CheckRslt*. In Step 6, HMS Core signs *AppKey* with *RSK* and sends it to HMS Server. Here, *RSK* refers to "RootSignKey", which is a signing key pre-stored in HMS Core and remains fixed. The signing algorithm is HS256. Upon receiving the request from HMS Core, HMS Server verifies the signature and *AppKey* (Step 7) and responds with KID_{SK}, $Enc_{RCK}(CK)$, and $Enc_{CK}(SK)$ (Step 8). Here, KID_{SK} is the Key ID of *SK*; *RCK* refers to "RootCryptoKey" and is a fixed crypto key pre-stored in HMS Core. HMS Core uses *RCK* to decrypt $Enc_{RCK}(CK)$ and uses *CK* to decrypt $Enc_{CK}(SK)$ (Step 9). The algorithm used for encrypting *CK* and *SK* is AES-GCM, so in Step 9, HMS Core will also verify the *Tag*s during decryption.

Finally, HMS Core obtains *SK*, which will be cached and reused within a period. HMS Core use it to sign *CheckRslt* (Step 10). HMS Server will verify the

signature and *AppKey* (Step 11), sign *CheckRslt* with *HK* (short for "Huawei Key") and send it back (Step 12).

(3) Make Decision. Third-party App Server will verify Huawei's signature with Huawei CBG Root CA certificate [39] (Step 16), and make decisions (Step 17).

4 Bypassing MARA Protection

In this section, we will propose our approach to bypass MARA. In Section 4.1, we first introduce the attack scenarios as well as the attacker's capabilities under each scenario. Afterwards, we illustrate the fundamental observation that lays the basis for our bypassing approach in Section 4.2. From Section 4.3 to 4.5, we respectively introduce the attack procedure targeting device integrity, data integrity, and app integrity. In Section 4.6, we introduce the implementation of the attack.

4.1 Scenarios and Attacker's Capabilities

Scenario 1: Bypassing Device Integrity Check The scenario here is that an official app has been installed on a compromised device, and the app uses the MARA service. The attacker hopes to prevent the app server from detecting any abnormalities in the device environment.

Attackers tend to run apps on such compromised devices for various reasons. In one case, the phone owner is the attacker and wants to use some game cheats [13]. In another case, the attacker is a malware [37] which has performed privilege escalation and wants to monitor or interfere with other legitimate apps.

Attacker's Capability. In this scenario, we assume that the attacker has **root privilege** on the device. This assumption is reasonable because if the device is compromised, it means that the device has already been rooted, bootloader-unlocked, or flashed with a custom ROM. These circumstances all enable the attacker to easily obtain root privilege.

Scenario 2: Bypassing App Integrity Check This scenario involves a repackaged app being installed on a non-rooted device and the app has integrated the MARA service. The attacker aims to make the repackaged app appear as the official version.

In reality, there are many cases of app repackaging. For example, some attackers may repackage popular apps to embed advertisements [8] and trick users into clicking on them.

Attacker's Capability. In this scenario, we assume that the attacker **can repackage** the app. Note that we neither require the attacker to have root privileges nor do we require them to possess the developer's *App Signing Key* [48].

Scenario 3: Bypassing Data Integrity Protection In this scenario, the attacker hopes to tamper with the *Data* exchanged between the app and the app server by performing a network man-in-the-middle attack. MARA service prevents attackers from directly intercepting and tampering with the network data, as the attackers cannot forge the signature of MS Server.

In real-life situations, for important data such as the gamer's score, apps typically use HTTPS for transmission. Therefore, in order to perform network interception, the attacker needs to insert a MITM certificate into the Android System Trust Store [40] and inject code into the victim's app process to bypass Certificate Pinning [41]. Both of these two operations require root privilege.

Attacker's Capability. In this scenario, we assume that the attacker has **root privilege** on the device. Such an assumption is reasonable because if an attacker is able to decrypt and manipulate HTTPS packets, they must have already obtained root privilege.

4.2   Fundamental Observation

There is a fundamental observation that lays the basis for our attacks against these MARA frameworks. The observation is that, **most app servers do not have a mandatory requirement for hardware-based check result**. Through reverse engineering, we found that the *DeviceInfo* sent by MS Core in Step ⑤ of Figure 2 contains two parts: software-based and hardware-based. The hardware-based part will only be available if the device is equipped with TrustZone; otherwise, only the software-based part will appear.

Taking GMS Core as an example. If the device supports hardware-based attestation, the VM inside GMS Core will retrieve a certificate chain from the device's hardware-backed KeyStore [23] and put it into *DeviceInfo$_2$* (see Figure 3). One certificate in this chain contains device information in its Extension Data, such as the verifiedBootState and the verifiedBootHash. Therefore, through this certificate chain, the GMS Server can obtain the device's bootloader status. For example, if the bootloader is unlocked, the GMS Server may consider that the device has been compromised.

We conducted tests on over 35,000 popular apps and found that, among the apps where we have confirmed the use of MARA service, **none of them has a mandatory requirement for hardware-based check result**. In other words, as long as a benign software-based check result is returned, the app server will not stop the app's running or prompt any security risks. We suppose this is because the apps aim to be compatible with a wider range of devices. This can lead to a downgrade attack: if the attacker can make the app server believe that the device does not support hardware-based attestation and always provide a benign software-based check result to the app, then the attacker can bypass such MARA.

We can implement a "trusted device" to launch the attack: such a "trusted device" is considered to be trusted by MS Server; but in reality, this "trusted device" is compromised, and the attacker can use it to forge MS Core's signature on arbitrary data.

4.3   Bypassing Device Integrity Check

As described in Section 4.1, we assume that the attacker has root privilege. Therefore, the attacker can tamper with the data sent and received by the Third-party App through code injection. The attacker aims to make the compromised
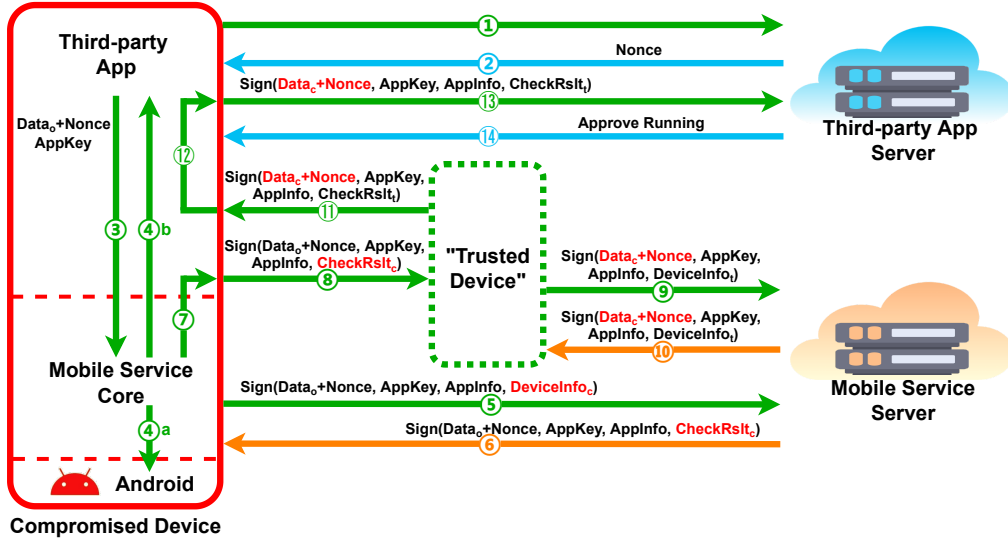
Fig. 5: Bypassing device integrity check and data integrity protection.

device be regarded to be trusted by MS Server. We use $DeviceInfo_c$ and $DeviceInfo_t$ to refer to the compromised device's $DeviceInfo$ and that of the trusted device, respectively. $CheckRslt_c$ and $CheckRslt_t$ have corresponding meanings as well. The process of bypassing device integrity check is illustrated in Figure 5.

From Step ① to Step ⑥ in Figure 5, the attestation process is proceeding normally. The "Trusted Device" intercepts the return value (Step ⑧) before App receives it from GMS Core (Step ⑦). The reason for choosing Step ⑧ instead of an earlier step such as Step ③ is that, at Step ⑧, the entire $AppInfo$ can be obtained by the "Trusted Device", which facilitates the construction of the request in Step ⑨. Afterwards, the "Trusted Device" obtains the signed $CheckRslt_t$ from the Mobile Service Server (Step ⑩), and replaces the signed $CheckRslt_c$ with the signed $CheckRslt_t$ (Step ⑪).

4.4   Bypassing App Integrity Check

As described in Section 4.1, in this scenario, we assume that the attacker can repackage the app. The repackaged app is installed onto a non-rooted device. We use $AppInfo_o$ and $AppInfo_r$ to refer to the $AppInfo$ of the official app and the repackaged app, respectively. The process of bypassing app integrity check is illustrated in Figure 7 in Appendix B.

4.5   Bypassing Data Integrity Protection

Due to the shared assumption between scenario 1 and scenario 3 (see Section 4.1), we merge the two processes into a single Figure 5 to save space. We use $Data_c$ and $Data_o$ to refer to the compromised $Data$ and the original $Data$, respectively. As described in Section 4.1, the attacker needs root privilege to intercept the HTTPS data packets, which means that the device has been compromised. Therefore,

the "Trusted Device" also needs to replace $DeviceInfo_c$ with $DeviceInfo_t$ to bypass device integrity check.

4.6   Attack Implementation

Whether it is in scenario 1, 2, or 3, the implementation of the attack mainly involves two parts: (1) Injecting code into the victim app to intercept the return value from the MS Core; (2) Implementing a "Trusted Device" to sign on arbitrary data.

Code Injection. We integrated the SDKs of SafetyNet and SafetyDetect, and developed two demo apps as the victim apps. In scenario 1 and scenario 3, since the attacker has **root privilege** on the device, we use Frida [42] to perform dynamic instrumentation on the victim apps. In scenario 2, we decompile and recompile the victim apps using ShakaApktool [43], and insert our logic into the original apps through smali [46] code.

Implementation of "Trusted Device". For HMS, we implement such a "Trusted Device" through a protocol-emulating script. The script is implemented in Python and runs on a laptop. For GMS, the $DeviceInfo_2$ is generated by executing the VM's bytecode (Step 10 in Figure 3) and the bytecode obtained from the GMS Server is different each time (Step 9 in Figure 3). Thus, in order to implement a "Trusted Device", we need an Android OS environment to support the running of the VM. We implement such an environment based on Magisk [52]. Specifically, We patched an official firmware (ROM^2) with Magisk, which will create a virtual $/system$ partition under the $/data$ partition. Magisk then redirects GMS Core's calls from the original $/system$ partition to the virtual $/system$ partition and unmounts any root privilege-related file systems to ensure that the retrieved $DeviceInfo$ is benign. We flashed this patched ROM into a OnePlus 5T phone, allowing this virtual environment to be hosted on this phone.

5   Evaluation

5.1   Effectiveness of Our Bypassing Approach

Experimental Setup To evaluate the effectiveness of our bypassing approach, we tested our bypassing approach on 10 Android smartphones. The 10 devices were running on different operating systems, ranging from Android 5 to Android 11, and covered different brands including Huawei, Xiaomi, OnePlus, Lenovo, and Nokia. These phones are all rooted to facilitate the experiment on bypassing device integrity checks. Details about the smartphones are listed in Table 3 in Appendix C.

Experiment Steps We install GMS Core and HMS Core (if they are not pre-installed) on all these mobile phones and try to bypass the device integrity check and app integrity check based on our approach. Testing for bypassing data integrity protection is not necessary because (1) it has the same assumptions as

---

^{2} OnePlus5T Hydrogen_43_OTA_065_all_2012030405_03dba2c095454647

bypassing device integrity check (see Section 4.1); (2) according to Section 4.3 and 4.5, as long as the attacker can implement a "trusted device" and bypass device integrity check, he can control the *Data+Nonce* sent by the "trusted device" to the MS Server and thus bypass data integrity protection.

Comparison with Existing Tools in the Wild As a comparison, we found the two most mentioned MARA bypassing implementations from the XDA forum [49]: Universal SafetyNet Fix [50] and Shamiko [51]. Overall, both Universal SafetyNet Fix and Shamiko take effect by **interfering with the MS Core's retrieval of the device information (during Step ④a in Figure 2). They target the MS Core.** In contrast, our method takes effect **between Step ⑦ and Step ⑧ in Figure 2. Our method targets the victim app.**

Table 1: Success rate of our bypassing approach compared with Universal SafetyNet Fix and Shamiko.

Byapsssing approach	Test item		Device No									
			#1	#2	#3	#4	#5	#6	#7	#8	#9	#10
Universal SafetyNet Fix	Device Int.	GMS	✔	–	✔	✘	✔	–	–	✘	–	–
		HMS	✔	✔	✔	✔	✘	✔	✔	✘	✘	✘
	App Int.						✘					
Shamiko	Device Int.	GMS	✔	–	✔	✘	✔	–	–	✘	–	–
		HMS	✔	✔	✔	✔	✘	✔	✔	✘	✘	✘
	App Int.						✘					
our approach	Device Int.	GMS	✔	–	✔	✔	✔	–	–	✔	–	–
		HMS	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
	App Int.						✔					

"Int." is short for "Integrity"; ✔: Succeeded in bypassing check; ✘: Failed to bypass check; –: Failed to invoke the attestation API due to MS Core's internal error.

Experiment Results The experiment results are shown in Table 1.

Bypassing device integrity check. On some phones, compatibility issues between the OS and the GMS Core have resulted in the failure of the SafetyNet service. Although having tried multiple combinations of GMS Core and ROM, we were still unable to make GMS Core execute correctly on these phones. Apart from these 5 issues, our approach succeed in all the other 15 cases.

Regarding the Universal SafetyNet Fix and Shamiko, they succeed in 9 out of 15 cases related to device integrity. Both Universal SafetyNet Fix and Shamiko need to unlock the victim device's bootloader. Therefore, they cannot be applied to devices #4, #9, and #10 as the root privilege on these devices is obtained through vulnerabilities (e.g., CVE-2020-0069 [53] on device #4), instead of through unlocking the bootloader and flashing a custom ROM. On device #5, HMS Server returned the advice of "RESTORE_TO_FACTORY_ROM" in the *CheckRslt*, which might be due to the detection of abnormal SELinux status. On device #8, both GMS and HMS detected abnormalities in the bootloader status and returned advice like "RESTORE_TO_FACTORY_ROM, LOCK_BOOTLOADER".
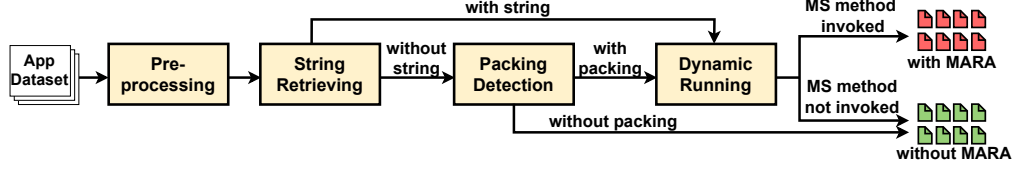
Fig. 6: Overview of our analysis pipeline for identifying affected apps.

Bypassing app integrity check.   Based on the assumption we presented in Section 4.1, in this scenario, the attacker does not have root privilege but can repackage the victim app. Therefore, neither Universal SafetyNet Fix nor Shamiko can be used. Our method only requires the attacker to interfere with the victim app, thus enabling the attacker to bypass the app integrity check through app repackaging.

5.2   Large-Scale Measurement Study

First, we conducted a measurement study on real-world apps to identify apps that use MARA services. Then, we analyzed involved security issues.

Dataset   To better reflect the usage of MARA frameworks worldwide, we crawled all the applications in the top lists of APKPure [54] and Chinese 360 Mobile Market [55]. After removing the losses during download, we collected a total of 35,245 apps, of which 20,253 came from APKPure (including 47 categories) and 14,992 came from the 360 Mobile Market (including 21 categories).

Measurement Approach We employed a mix of static and dynamic analysis mechanisms to identify the apps that use MARA services. The overall pipeline of our approach is illustrated in Figure 6. We first use static analysis to filter out potential apps and then conduct dynamic analysis to further confirm the results.

(1) Preprocessing.   There are two steps in preprocessing: further filter out corrupted apps and extract APKs from XAPK [58] files for subsequent static analysis.

(2) String Retrieving and Packing Detection.
The result of straightforwardly retrieving the signatures of classes and methods is far from satisfying due to code obfuscation (e.g., ProGuard [56]) and app packing [34].

- Code obfuscation will remove the signatures of classes and methods from the app code, thus we choose to retrieve constant strings. We use Apktool [44] to decompile the app's code and use grep [45] with regular expressions to retrieve strings inside the MARA SDKs. Apps with string features will be dynamically analyzed.
- App packing can hide almost all static code features, including constant strings. To tackle this issue, we implemented a packing detector. For apps without string features, if the packing detector identifies that it has been

packed, we will also conduct dynamic detection on it. The packing detector is implemented based on Soot [47] and integrates the signatures of 11 mainstream packers.

(3) Dynamic Running. We use Monkey [57] to generate UI events and drive the tested app to run on a OnePlus 5T device. Meanwhile, we use Frida [42] to hook the MS process. If the app invokes the MARA service during its running process, the corresponding methods in the MS process will be invoked. To prevent the MS process from being killed and causing our hooking to fail, we regularly use an app to call the MS process to keep it active at all times. In addition, we noticed that many apps utilize the SafetyNet service when users log in via a Google account. Therefore, we logged in to our Google account on the device in advance to facilitate triggering the SafetyNet service.

Table 2: Overview of app measurement results

MARA	Total Apps	Static Analysis			Dynamic Analysis	
			String Retrieving	Packing Detection		
SafetyNet	35,245	potential	4,296	11,234	use	73
					don't use	11,161
		unsuspicious	30,949	24,011		
SafetyDetect	35,245	potential	226	7,158	use	31
					don't use	7,127
		unsuspicious	35,019	28,087		

Results and Findings Our app measurement results are shown in Table 2. In total, among the 35,245 top Android apps, we identified 73 apps that have used the SafetyNet service and 31 apps using the SafetyDetect. These apps include some top most popular applications and games, such as TikTok Lite, Huawei Wallet, China Merchants Bank, Pokémon GO, NBA LIVE, and so on. Detailed information on our measurement result is available at `https://github.com/zhouziyi1/MARA`.

To detail, when detecting SafetyNet-related apps, our string retrieving process enables us to locate 4,296 candidates. The packing detection process helps us locate 6,938 additional candidates. To this end, we have a total number of 11,234 candidates through the static analysis process. Our dynamic running confirmed that 73 out of these 11,234 apps indeed invoke the SafetyNet service. For SafetyDetect, we ultimately confirmed 31 apps.

Through further manual inspection, we found that most MARA invocations occur when users log in. For example, among the 31 SafetyDetect-related apps, 24 apps call SafetyDetect service when users log in, while the remaining 7 apps call the service as soon as the user launches the app.

Case Studies

(1) Pokémon GO. Pokémon GO [9] is one of the most popular and highest-grossing mobile games worldwide, with over 572 million downloads [11] and $6 billion in player spending [10]. It allows players to catch nearby "Pokémon" based on players' geographical location. If players can use mock locations instead of real ones, they can cheat in the game. One type of mock location tool [60,59,62,61] does not require root privilege, but they can be easily blocked by Pokémon

GO. Another type of mock location tool [63] requires root privilege and is more difficult to detect. To block such mock location tools, Pokémon GO utilizes the SafetyNet service. Specifically, when a user logs into Pokémon GO with a Google account, Pokémon GO calls the SafetyNet API and checks the integrity of the current device. If the current device has been rooted, Pokémon GO prohibits the user from logging in. We implemented our attack method (as described in Section 4.6) on a rooted Mi 8 phone and successfully bypassed Pokémon GO's root detection. This allows us to successfully perform location spoofing using Fake GPS Location Spoofer [63].

(2) Huawei Wallet. Huawei Wallet is a payment app with over 100 million active users [64]. To prevent root malware (such as [37,53,65]) from threatening users' property and privacy, Huawei Wallet checks the device's integrity using the SafetyDetect service right after app's startup. If the device doesn't meet the security requirements, Huawei Wallet displays a prompt to the user. The app will only continue running after the user confirms the risk. We successfully deceived the SafetyDetect service with our bypassing approach (as described in Section 4.6) on a OnePlus 9R phone, making it difficult for users to realize the existence of malware. Afterwards, malware with root privilege can access files in the private directory of Huawei Wallet, stealing user account information, program logs, cookies, and other sensitive data.

6   Conclusion

In this paper, we conducted an in-depth security study on Google SafetyNet and Huawei SafetyDetect, two mainstream Manufacturer-provided Android Remote Attestation (MARA) frameworks, and found a new way to bypass their integrity check. *Compared with existing bypassing implementations in the wild, our bypassing approach can succeed on more devices.* To further evaluate the real-world impact of our identified issues, we performed a large-scale measurement over a set of top popular apps, and analyzed related security implications to the apps that integrate MARA service.

7   Acknowledgments

References

1. Tian Y, Chen E, Ma X, et al. (2016, December). Swords and shields: a study of mobile game hacks and existing defenses. In Proceedings of the 32nd Annual Conference on Computer Security Applications (pp. 386-397).
2. Karkallis P, Blasco J, Suarez-Tangil G, et al. (2021). Detecting video-game injectors exchanged in game cheating communities. In Proceedings of the 26th European Symposium on Research in Computer Security, Darmstadt, Germany, October 4–8, 2021, Part I 26 (pp. 305-324).
3. Nguyen-Vu L, Chau N T, Kang S, et al. (2017). Android rooting: An arms race between evasion and detection. Security and Communication Networks, 2017.
4. Chen S, Fan L, Meng G, et al. (2020, June). An empirical assessment of security risks of global android banking apps. In Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (pp. 1310-1322).
5. Sun S, Yu L, Zhang X, et al. (2021, November). Understanding and detecting mobile ad fraud through the lens of invalid traffic. In Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (pp. 287-303).
6. Li L, Bissyandé T F, Klein J. (2019). Rebooting research on detecting repackaged android apps: Literature review and benchmark. IEEE Transactions on Software Engineering, 47(4), 676-693.
7. Song W, Ming J, Jiang L, et al. (2021, May). App's auto-login function security testing via android os-level virtualization. In 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE) (pp. 1683-1694). IEEE.
8. Xue L, Zhou H, Luo X, et al. (2021, May). Happer: Unpacking Android apps via a hardware-assisted approach. In 2021 IEEE Symposium on Security and Privacy (SP) (pp. 1641-1658). IEEE.
9. Pokémon GO, `https://play.google.com/store/apps/details?id=com.nianti clabs.Pokemongo&hl=en_US`. Last accessed 14 May 2023
10. Pokémon Go hits \$6 billion in player spending, `https://play.google.com/stor e/apps/details?id=com.nianticlabs.Pokemongo&hl=en_US`. Last accessed 14 May 2023
11. Pokémon Go Revenue and Usage Statistics (2023), `https://www.businessofap ps.com/data/pokemon-go-statistics/`. Last accessed 14 May 2023
12. Android - Google Mobile Services, `https://www.android.com/gms/`. Last accessed 14 May 2023
13. Fake GPS Location Spoofer, `https://play.google.com/store/apps/details?i d=com.incorporateapps.fakegps.fre`. Last accessed 14 May 2023
14. HMS Core, `https://developer.huawei.com/consumer/en/hms/`. Last accessed 14 May 2023
15. Mobile Application Distribution Agreement (Android), `https://www.sec.gov/Ar chives/edgar/containers/fix380/1495569/000119312510271362/dex1012.htm`. Last accessed 14 May 2023
16. HMS Core (APK) Preloading Guide: Ecosystem Cooperation, `https://developer.huawei.com/consumer/en/doc/development/hmscore-common-G uides/overview-0000001222509146`. Last accessed 14 May 2023
17. Google Play Store, `https://apkpure.com/google-play-store/com.android.v ending`. Last accessed 14 May 2023
18. YouTube, `https://play.google.com/store/apps/details?id=com.google.and roid.youtube`. Last accessed 14 May 2023
19. HUAWEI Wallet, `https://consumer.huawei.com/en/mobileservices/wallet/`. Last accessed 14 May 2023

20. HUAWEI Health, `https://consumer.huawei.com/en/mobileservices/health/`. Last accessed 14 May 2023
21. HMS Core 5.0 launched for the global developers, `https://www.huaweicentral.com/hms-core-5-0-launched-for-the-global-developers-comes-with-7-new-kits-and-services/`. Last accessed 14 May 2023
22. Google I/O 2023: What's new in Google Play, `https://io.google/2023/program/9019266d-186c-4a61-9cc5-b1c665eb40fb/`. Last accessed 21 May 2023
23. Verifying hardware-backed key pairs with Key Attestation, `https://developer.android.com/training/articles/security-key-attestation`. Last accessed 14 May 2023
24. Protect against security threats with SafetyNet, `https://developer.android.com/training/safetynet`. Last accessed 14 May 2023
25. Safety Detect, `https://developer.huawei.com/consumer/en/hms/huawei-safetydetectkit/`. Last accessed 14 May 2023
26. Mulliner C, Kozyrakis J. (2017). Inside Android's SafetyNet Attestation. Black Hat EU, 2017.
27. Thomas R. (2022). DroidGuard: A deep dive into SafetyNet. Black Hat Asia, 2022.
28. Examining the value of SafetyNet Attestation as an Application Integrity Security Control, `https://census-labs.com/news/2017/11/17/examining-the-value-of-safetynet-attestation-as-an-application-integrity-security-control/`. Last accessed 14 May 2023
29. How I discovered an easter egg in Android's security and didn't land a job at Google, `https://habr.com/en/articles/446790/`. Last accessed 14 May 2023
30. RFC 9334: Remote ATtestation procedureS (RATS) Architecture, `https://datatracker.ietf.org/doc/rfc9334/`. Last accessed 14 May 2023
31. Coker G, Guttman J, Loscocco P, et al. (2011). Principles of remote attestation. International Journal of Information Security, 10, 63-81.
32. Ibrahim M, Imran A, Bianchi A. (2021, June). 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 (pp. 150-162).
33. Aldoseri A, Chothia T, Moreira-Sanchez J, et al. (2023, January). Symbolic Modelling of Remote Attestation Protocols for Device and App Integrity on Android. In 18th ACM ASIA Conference on Computer and Communications Security. Association for Computing Machinery (ACM).
34. Duan Y, Zhang M, Bhaskar A V, et al. (2018, February). Things You May Not Know About Android (Un) Packers: A Systematic Study based on Whole-System Emulation. In NDSS.
35. CNCERT/CC: National Computer Network Emergency Response Technical Team/Coordination Center of China, `https://www.cert.org.cn/publish/english/index.html`. Last accessed 14 May 2023
36. Google Bug Hunters, `https://bughunters.google.com/`. Last accessed 14 May 2023
37. A Very Powerful Clipboard: Analysis of a Samsung in-the-wild exploit chain, `https://googleprojectzero.blogspot.com/2022/11/a-very-powerful-clipboard-samsung-in-the-wild-exploit-chain.html`. Last accessed 15 May 2023
38. Upcoming Change: New certificate chain in the API response signature, `https://groups.google.com/g/safetynet-api-clients/c/-2ShuYt5kFg`. Last accessed 17 May 2023
39. SysIntegrity API, `https://developer.huawei.com/consumer/en/doc/development/Security-Guides/dysintegritydevelopment-0000001050156331`. Last accessed 15 May 2023

40. CA-certificates, `https://android.googlesource.com/platform/system/ca-certificates/`. Last accessed 15 May 2023
41. Pin certificates, `https://developer.android.com/training/articles/security-config#CertificatePinning`. Last accessed 15 May 2023
42. Frida, `https://frida.re/`. Last accessed 15 May 2023
43. ShakaApktool, `https://github.com/rover12421/ShakaApktool`. Last accessed 15 May 2023
44. Apktool: A tool for reverse engineering Android apk files, `https://ibotpeaches.github.io/Apktool/`. Last accessed 15 May 2023
45. Grep, `https://www.gnu.org/software/grep/manual/grep.html`. Last accessed 15 May 2023
46. Smali, `https://github.com/JesusFreke/smali/wiki`. Last accessed 15 May 2023
47. Soot, `http://soot-oss.github.io/soot/`. Last accessed 15 May 2023
48. Application Signing, `https://source.android.com/docs/security/features/apksigning`. Last accessed 15 May 2023
49. XDA Portal & Forums, `https://www.xda-developers.com/`. Last accessed 15 May 2023
50. Universal SafetyNet Fix, `https://github.com/kdrag0n/safetynet-fix`. Last accessed 15 May 2023
51. Shamiko v0.7.2, `https://github.com/LSPosed/LSPosed.github.io/releases`. Last accessed 15 May 2023
52. Magisk, `https://github.com/topjohnwu/Magisk/releases`. Last accessed 27 May 2023
53. CVE-2020-0069, `https://nvd.nist.gov/vuln/detail/CVE-2020-0069`. Last accessed 27 May 2023
54. APKPure: Download APK on Android with Free Online APK Downloader, `https://apkpure.com/`. Last accessed 27 May 2023
55. 360 Mobile Assistant, `http://m.app.so.com/`. Last accessed 27 May 2023
56. ProGuard: Java Obfuscator and Android App Optimizer, `https://www.guardsquare.com/proguard`. Last accessed 27 May 2023
57. UI/Application Exerciser Monkey, `https://developer.android.com/studio/test/other-testing-tools/monkey`. Last accessed 27 May 2023
58. XAPK file, `https://apkpure.com/xapk.html`. Last accessed 27 May 2023
59. FGL Pro, `https://play.google.com/store/apps/details?id=com.ltp.pro.fakelocation&hl=en_US&gl=US`. Last accessed 27 May 2023
60. Fake GPS Location-GPS JoyStick, `https://play.google.com/store/apps/details?id=com.theappninjas.fakegpsjoystick&hl=en`. Last accessed 27 May 2023
61. Cha Cha Helper, `https://www.xxzhushou.cn/?channelid=352666`. Last accessed 27 May 2023
62. Moloc, `https://www.coolapk.com/apk/top.xuante.moloc`. Last accessed 27 May 2023
63. Fake GPS Location Spoofer, `https://play.google.com/store/apps/details?id=com.incorporateapps.fakegps.fre`. Last accessed 27 May 2023
64. Huawei has the highest number of active smartphone users globally: how is this possible?, `https://www.gizchina.com/2022/08/27/huawei-has-the-highest-number-of-smartphone-users-globally-how-is-this-possible/`. Last accessed 27 May 2023
65. One of China's most popular apps has the ability to spy on its users, `https://edition.cnn.com/2023/04/02/tech/china-pinduoduo-malware-cybersecurity-analysis-intl-hnk/index.html`. Last accessed 27 May 2023
66. Protocol Buffers, `https://protobuf.dev/`. Last accessed 30 May 2023

Appendix A   Integrity Checking Items

A.1   Device Integrity Checking Items

- SU Files
- SeLinux Status
- Bootloader Status
- Root Frameworks
- Hooking Frameworks
- Emulators

A.2   App Integrity Checking Items

- App Package Name
- App Certificate Fingerprint
- Apk File Digest
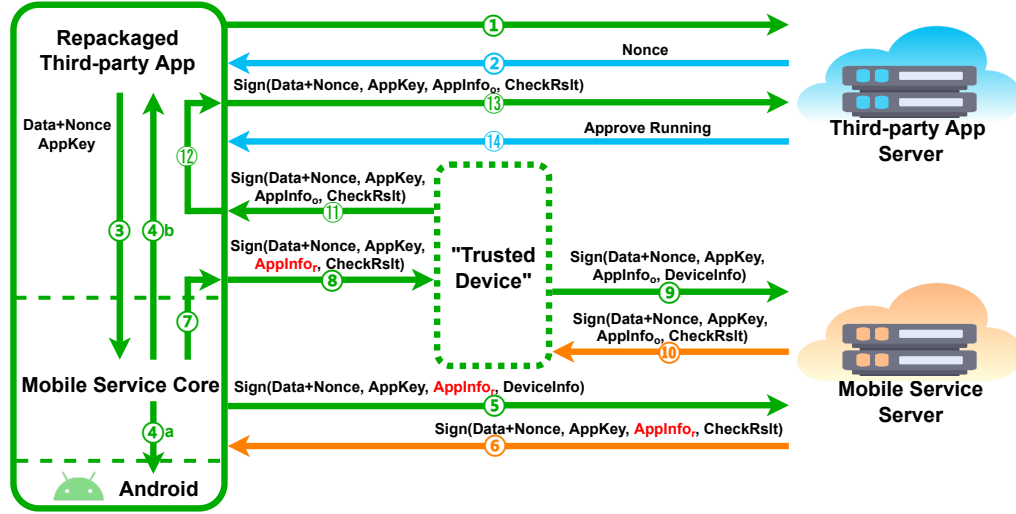
Appendix B   Bypassing app integrity check

Fig. 7: Bypassing app integrity check.

Appendix C   Device details

Table 3: Details about the Android devices used in Section 5.1

No	Model number	Android version	Build number	Bootloader status	GMS version	HMS version
#1	OnePlus 9R	11	Oxygen OS 11.2.4.4.LE28DA	unlocked	21.06.13	6.8.0.332
#2	Xiaomi Mi CC9 Pro	11	MIUI 13.0.4 Stable 13.0.4.0(RFDCNXM)	unlocked	21.21.16	6.8.0.332
#3	Oneplus 5T	10	H2OS 10.0.3	unlocked	22.12.15	6.8.0.332
#4	Nokia X5	9	00CN_2_15A_SP02	locked	22.12.15	6.8.0.332
#5	Xiaomi Mi 8	9	MIUI 10 9.8.22 Beta	unlocked	22.12.15	6.8.0.332
#6	OnePlus 5	9	H2OS 9.0.5	unlocked	22.12.15	6.8.0.332
#7	Motorola P30	8.1.0	ZUI 4.0.374 Stable	unlocked	20.12.16	6.8.0.332
#8	Xiaomi Mi 5	8.0.0	MIUI 10.8.11.22 Beta	unlocked	20.12.16	6.8.0.332
#9	Huawei Mate 9	7	EMUI 5.0	locked	10.2.98	6.10.4.300
#10	Lenovo K5 Note	5.1.1	VIBE UI V3.0	locked	10.0.84	6.8.0.332