SafetyNOT: On the usage of the SafetyNet Attestation API in Android

Muhammad Ibrahim
Purdue University
USA
ibrahi23@purdue.edu

Abdullah Imran
Purdue University
USA
imran8@purdue.edu

Antonio Bianchi
Purdue University
USA
antoniob@purdue.edu

ABSTRACT

Many apps performing security-sensitive tasks (e.g., online banking) attempt to verify the integrity of the device they are running in and the integrity of their own code. To ease this goal, Android provides an API, called the SafetyNet Attestation API, that can be used to detect if the device an app is running in is in a "safe" state (e.g., non-rooted) and if the app's code has not been modified (using, for instance, app repackaging). In this paper, we perform the first large-scale systematic analysis of the usage of the SafetyNet API. Our study identifies many common mistakes that app developers make when attempting to use this API. Specifically, we provide a systematic categorization of the possible misusages of this API, and we analyze how frequent each misuse is. Our results show that, for instance, more than half of the analyzed apps check SafetyNet results *locally* (as opposed to using a remote trusted server), rendering their checks trivially bypassable. Even more surprisingly, we found that none of the analyzed apps invoking the SafetyNet API uses it in a fully correct way.

CCS CONCEPTS

• **Security and privacy** → **Software reverse engineering**; **Intrusion detection systems**.

KEYWORDS

attestation, Android, tampering, SafetyNet, API misusage, reverse engineering

ACM Reference Format:
Muhammad Ibrahim, Abdullah Imran, and Antonio Bianchi. 2021. SafetyNOT: On the usage of the SafetyNet Attestation API in Android. In *The 19th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '21), June 24-July 2, 2021, Virtual, WI, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3458864.3466627

1 INTRODUCTION

Because of Android's high market share, many companies and businesses use Android applications to provide their services but there are characteristics of Android that can be exploited by attackers to compromise Android devices and applications. These characteristics include rooting the device, repackaging applications, monitoring network traffic, and using unofficial OS distributions. Rooting the device allows users to achieve superuser privileges, change system settings, and gain access to private memory areas. There are tools available for these exploits which make them relatively easier to perform. For this reason, some applications want to check the status of OS and their code. Specifically, they want to check if the OS on the Android device is non-rooted, and the application is not modified. Because by having higher privileges (rooting) and modifying the application, an attacker has much more feasibility to steal information, access premium features, send false information, and compromise the application server.

There are ways of detecting root and repackaging on Android which developers can implement in their applications [19, 46]. However, these methods are shown to be bypassable [28, 37]. Traditionally apps can check for the presence of the binary su (which allows obtaining root privileges) to verify that the device has not been rooted. Complementarily, they can check the contents of the file in which they are stored to verify that they have not been modified. Performing these ad-hoc checks has 2 main limitations:

(1) These checks are complex and error-prone to implement.
(2) Since these checks are performed in the app's code, an attacker can reverse engineer the app's code and patch it so that it does not perform these checks anymore.

To ease this task and provide a unified, easy-to-use solution, Google implemented a comprehensive API called SafetyNet Attestation API [24] in Android for detecting compromised devices and tampered applications. SafetyNet determines the state of the app and the device integrity in a separate component, running outside the app's process. Apps can call SafetyNet to obtain the results of these checks and send these results to a remote server, where they can check if they are correct. SafetyNet is better than other approaches because on the client-side it only requires calling the API and sending the result to the backend server. Developers do not need to implement their own checks that are likely to be incomplete and bypassable. The server can decide what steps to take based on the received information. For example, the server can send an error message and terminate current session with a device if it failed attestation. This helps the authors of an application to implement a system for detecting tampering of their application and the integrity of the devices which are running their application.

However, SafetyNet is not straightforward to use for a developer with no security expertise. SafetyNet must be used in a secure manner and verification of the result must be done on the server-side in order to get a correct attestation. If SafetyNet is not used correctly it can be bypassed by attackers. For example, if the verification of the attestation result is done on the application side, an attacker can modify the application to bypass the verification.

MobiSys '21, June 24-July 2, 2021, Virtual, WI, USA

ACM ISBN 978-1-4503-8443-8/21/07.
https://doi.org/10.1145/3458864.3466627

We have analyzed 163,773 top Android applications and found that 62 applications try to leverage SafetyNet API to defend against tampering. We analyzed their SafetyNet usage and found that none of the applications are doing the checks correctly. We found 21 applications that use SafetyNet to change app behavior when tampering is detected. We bypassed SafetyNet and prevented this behavior change in 16 (84%) of these applications.

While our analysis is not fully automated, nor fully precise, it constitutes a first step toward automatically verifying the correct usage of the SafetyNet API on the large scale. Future work in this area could build upon our analysis and develop further vetting tools targeting this API. These vetting tools could be integrated in the market-level app approval process.

In summary, in this paper we make the following contributions:

(1) We develop a semi-automated pipeline for the analysis of the usage of the Attestation API.
(2) We perform the first large scale study of SafetyNet Attestation in Android applications analyzing 163,773 APKs.
(3) We identify common mistakes that developers have made that lead to SafetyNet Attestation being ineffective.
(4) We show how the SafetyNet checks can be bypassed in the applications that misuse the API.

2 BACKGROUND

Typically, Android applications are downloaded and installed from the Google Play Store that is available on the Android device. Specifically, on the Play Store developers must sign the APK before uploading and publishing their application. An APK is signed using a key that only the developers can access. Android does not allow installation of an unsigned APK. APKs can be disassembled into smali code and modified. Smali is disassembled Dalvik bytecode. Dalvik bytecode is compiled Java code of applications that runs on Dalvik VM on Android.

2.1 Android Tampering

An application can be modified by third parties by modifying the APK of an application. APK is modified by disassembling the APK into smali and modifying the smali code. The modified smali can be repackaged to an APK but needs to be signed again. If an attacker tries to modify and repackage, they will have to sign the APK first before it can be installed on an Android device. Repackaging the applications changes the hash of APK. If the developers use a different key to sign, the signing certificate of the APK will also be changed.

An attacker can try to change application behavior by modifying application code or by modifying OS. The OS modifications can include rooting the device and installing custom OS or kernel. We will define a device to be "tampered" if the attacker is able to inject code and modify network packets of an application to change its behavior. We will define an application to be "tampered" if it has been repackaged and has a different signing certificate and hash from the original application. We define this method of controlling application behavior as tampering.

So app developers may want to check the integrity of their applications code and the state of the client device. However, developers' techniques to detect tampering and attackers' efforts to hide tampering are a cat-and-mouse race. SafetyNet Attestation API provides a unified solution to this issue.

2.2 Android Application Obfuscation

The code of many Android applications is obfuscated. In particular, many applications use ProGuard [25] to obfuscate their code, since it is the default obfuscation tool for Android, available as part of the Android compilation toolchain. ProGuard obfuscates the application code by changing method and class names. For example, a method named *getSecretKey* gets renamed to *aaa*.

Renaming method names makes reverse engineering of applications harder, since it complicates understanding the application code. In addition, it complicates hooking an application's code. In fact, hooking is typically performed, using tools like Frida [2], by specifying the method name to hook.

Consequently, changing method names makes hooking harder, since obfuscation could change the name of the method that a researcher wants to hook.

2.3 Attacker Model

In this paper we assume, an attacker can change client application code and compromise the Android device. By changing application code, the attacker can patch the application to remove integrity checks. The attacker can compromise the Android device by methods such as rooting. The attacker is then able to inject code while the client application is running and can analyze and modify the client application's network traffic.

However, we also assume that attacker cannot compromise SafetyNet Attestation API. This means that an application using SafetyNet Attestation API will get the correct information about device integrity from SafetyNet even though the device is compromised. Compromising SafetyNet Attestation API would mean attacker is able to change integrity checks that are performed by SafetyNet before the client application receives them. However, we assume the attacker can modify the application or change/spoof the SafetyNet results after the application has received them.

This paper focuses on how SafetyNet is used and how to defeat SafetyNet checks in apps that use it wrongly. Therefore, a root attacker able to compromise the SafetyNet API is considered out of scope. The issue of securing SafetyNet against root attackers is orthogonal to the issue of using this API correctly.

Indeed, in certain scenarios (e.g., specific Android devices and versions), root attackers could compromise the SafetyNet API itself. However, Google is taking step toward making SafetyNet increasingly resilient against root attackers [39], including using Trusted Execution Environments (e.g., TrustZone) to verify a device's operating system integrity. Nevertheless, it is important to note that if the SafetyNet API is not used correctly by app developers, attackers can bypass its checks, even if the API itself has not been compromised.

3 THE SAFETYNET ATTESTATION API

Google provides the SafetyNet Attestation API [24] to attest the integrity of an Android device and of an app. Developers can use this API by importing the *SafetyNet* package in their application.

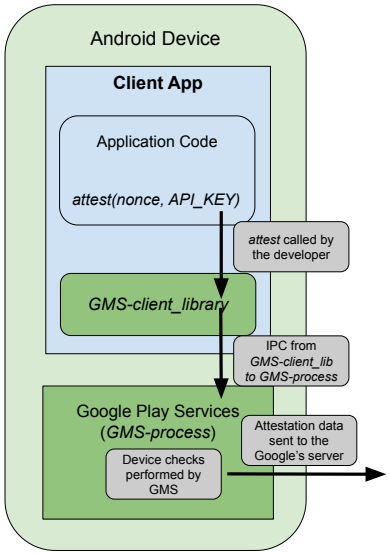

Figure 1: SafetyNet API

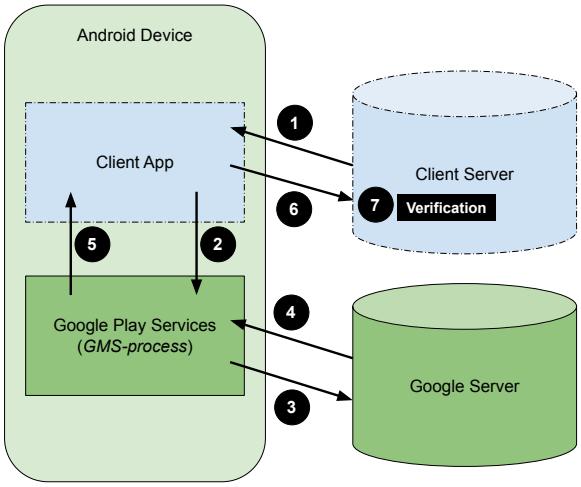

Figure 2: SafetyNet Attestation

The Attestation API can be called by using the *attest* function as stated below:

```
SafetyNet.getClient(this).attest(nonce, API_KEY)
```

The attest function takes two arguments: a *nonce* and an *API Key*. The nonce must be generated in the backend server of the application and sent to the device when attestation is requested. The API Key is generated using the Google APIs Console. Google APIs Console is an online platform that developers can use to manage their Google APIs. When attestation is requested, code within the closed sourced GMS (Google Mobile Services) package is executed. GMS code makes an IPC call to the GMS process (Google Play Services) that is running separately on the Android device. We will refer to this separate process as *GMS-process* and the code present in developer's application as *GMS-client_library*. The *GMS-process* performs several checks and measurements on the device and sends the results to Google's backend server. This process is illustrated in Figure 1.

Since Google has not made this code open source, exact details of checks and measurements are not known. But researchers have tried to reverse SafetyNet to figure out how the internals work [29]. They have identified various checks which include:

- Looking for the `su` binary that allows root user access. Presence of the `su` binary means there is a high probability that the device is rooted.
- Checking package names of default applications. Changing default application package names, such as for web browsers or text apps, could mean malicious apps are trying to pretend be those apps.
- Collecting global settings values like if non-market apps can install and if adb [21] is enabled. Non-market apps are more likely to have malicious code.
- Checking for proxies configured on the device and their IPs. Suspicious proxies could mean network traffic is being intercepted.
- Looks for malicious CA Certificates in the cert store.

Google's server sends back signed attestation data to the *GMS-process* on the device. Finally, the *GMS-process* sends the signed attestation data to the client app via an IPC. Client app sends the signed attestation data to its backend server for verification.

Overall, the Attestation process is composed of the following steps which are illustrated in Figure 2:

1. A nonce generated by the client server is sent to client app
2. The client app requests Attestation from the *GMS-client_library* by calling the *attest* method with the Nonce and the API Key
3. The *GMS-process* performs checks and sends the results to the Google Servers
4. The Google Servers send SafetyNet Attestation data to the *GMS-process*
5. The *GMS-process* sends the SafetyNet Attestation object to the client app
6. The client app extracts the SafetyNet JWS and sends it to the client server
7. The client server performs verification of SafetyNet JWS

The client application receives a *AttestationResponse* object that includes a JWS (JSON Web Signature) of SafetyNet Attestation data. This object is received via a callback that is registered when attest function is called. JWS can be extracted by using the *getJwsResult()* method of *AttestationResponse* object. This JWS should be sent to the client application's backend server for attestation.

The format of JWS from SafetyNet Attestation looks as follow:

```
{
"timestampMs": 9860437986543,
"nonce": "R2Rra24fVm5xa2Mg",
"apkPackageName": "package.of.client.app",
"apkCertificateDigestSha256": ["base64
encoded, SHA-256 hash of the certificate
used to sign requesting app"],
"ctsProfileMatch": true,
"basicIntegrity": true,
```

}

This JWS should be sent to the client application's backend server for attestation. The server should verify the SSL certificate chain included in the JWS and use that certificate to verify the signature of the payload. This ensures that the payload was not spoofed or tampered with. The server should also verify the nonce to prevent replay attacks and other values to make sure the Attestation data was sent by the legitimate application. After that, the server can look at the *ctsProfileMatch* and *basicIntegrity* boolean values to assess the integrity of client device. Based on the assessment performed by the server, the server should send the application instructions on whether to continue execution on the device or not. If attestation fails, the server should stop communication for that session with the device and instruct the application to terminate or show an error message.

4 CATEGORIZATION OF SAFETYNET MISUSAGES

In this section, we will discuss several ways in which SafetyNet Attestation can be used in an insecure manner and can be bypassed by attackers. The circled numbers refer to the step of SafetyNet Attestation mentioned in the previous section.

The misusages discussed in this section are comprehensive and cover all possible ways in which the SafetyNet API can be mishandled when used by a developer. In Figure 1, the dotted boundary entities represent where developer can misuse SafetyNet. Green areas are not under control of the developer, they are either handled by the Android OS or Google. So possible misusages can occur at step ❶, ❷, ❺, ❻, and ❼. We analyze each of these steps and inspect all possible mistakes a developer can make that can render SafetyNet Attestation ineffective.

At step ❶, the server is required to send a nonce that is used in the *attest* function. If the nonce is not generated correctly, the SafetyNet Attestation becomes vulnerable to replay attacks. At step ❷, an API key is required to call the *attest* function. Mistakes in the API key usage will always result in unsuccessful SafetyNet Attestation. At step ❺, attestation result is received from the Google and not handling the results and errors correctly can make the SafetyNet Attestation ineffective. At step ❻, attestation result is sent to the server. If there are mistakes in sending the result or in step ❼ when verifying the result at server, SafetyNet Attestation will be useless. Our misusage categories cover all four of these steps, so we can claim our misusages are comprehensive. We define these categories and discuss their details in the following subsections.

4.1 Local Checks (*Mis-LoCh*)

As mentioned in the previous section, SafetyNet Attestation returns a JWS object ❺ which represents the device and application state. This JWS should be sent to ❻ and verified at the backend server ❼ of the application. If the device fails SafetyNet integrity tests, then the server should follow whatever protocol was intended for that case.

Local checks can be defined as parsing and verifying the attestation data included in the SafetyNet JWS on the client device. If an application performs local checks in the client device, then these checks can easily be bypassed by an attacker. For instance, an attacker can use APK repackaging to remove the checks or spoof the attestation results in the application. On a tampered device, the attacker can use function hooks to spoof the results stored in the SafetyNet JWS. In this way the server or the application on the client device has no way of telling if it is running on a tampered device or the application is tampered. In later section, we show case studies where applications are doing local checks and how they can be bypassed.

4.2 Using Google Test Server (*Mis-Test*)

Google provides a verification service for SafetyNet. This is a test server to which a client application can send a SafetyNet JWS and the server will verify the signature and send the integrity results back to the application. This service is only intended for testing purposes. Using this service in production environment can render the SafetyNet Attestation useless. The Google test server only returns the boolean values that represent the integrity results and does not provide any signature for verification. The Google test server returns data like *basicIntegrity*, *apkPackageName*, signature validity of the SafetyNet JWS and *ctsProfileMatch*. The data sent by the Google test server can be checked locally, thus being affected bu the issue explained in Section 4.1. Alternatively, these results can be sent to the application's backend server. However, the data sent by the Google test server is not signed. Therefore, if the device or the application is tampered, an attacker can easily change these values so that it looks like the application is running in non-tampered conditions. We found that some developers, possibly due to misinterpretation of the documentation, use this test server in production builds.

4.3 Local Nonce Generation (*Mis-LoNon*)

The *attest* function in SafetyNet API takes a nonce value as an argument ❷. This value is included in the JWS result returned by the SafetyNet API. The attestor (server) can match the value in the JWS and the value passed to the function to verify that correct JWS result is being attested to avoid replay attack.

If the nonce is generated locally in a tampered device or application, then an attacker can use a nonce from a previous result to perform a replay attack. This attack can be performed by first obtaining the SafetyNet JWS in a state in which the device passes SafetyNet checks. We will refer to this result as SnetJWS1.

Now we will look a tampered device or application in which the attacker can inject code and consequently SafetyNet checks fail. In this device, the attacker can hook *getJwsResult()* and return the SnetJWS1 instead of the value that was originally supposed to be returned.

Now we will consider the case where a nonce was not generated by the server. Usually in this case, the nonce is generated within the application on client device. This nonce is sent to the server with the JWS. Since the device or the application is tampered, the attacker can replace the nonce sent to the server with the nonce present in SnetJWS1. When the result is sent to server, the server will have no way of verifying if the result is coming from the correct SafetyNet Attestation call or not. The server will verify values in

SnetJWS1 and the spoofed nonce. The server will never know that the device or the application is tampered.

If the nonce was generated at the server (1), it can send the nonce to the client device and wait for the result. When verifying the result, the server can match the value it generated with the value contained in the JWS to verify if the JWS is replayed or not.

We found that in many cases, the nonce is not sent by the server to the app in Step (1). In these cases, the nonce is generated locally by the app and the server does not check its value. Therefore, we can send a spoofed SafetyNetJWS.

4.4 Wrong Verification at Server (*Mis-SerVeri*)

Complete verification of SafetyNet JWS requires implementing several checks at the server side (7). These checks include:

(1) Validating the SSL certificate chain and the SSL hostname.
(2) Verifying the signature of the JWS message.
(3) Matching the values in the JWS payload to the expected ones. These values include:
   (a) The nonce used for attestation
   (b) The APK Package Name
   (c) The hash of application's signing certificates
   (d) The timestamp representing the time when the JWS was generated by the Google's servers

If a server fails to perform these checks, an attacker might be able to send a tampered SafetyNet JWS to the server and evade detection. For example, modifying the values of attestation like *basicIntegrity* will invalidate the signature of JWS. If the signature and SSL certificate are not verified properly, the modifications will not be detected.

4.5 Sending Partial JWS to Server (*Mis-PartJWS*)

An application may send specific values extracted from the string representation of SafetyNet JWS to their servers (6). Most of the times these are boolean values of *ctsProfileMatch* and *basicIntegrity*. Using JWS this way defeats the purpose of having certificates and signatures for validation. In fact an attacker can easily replace these values on a tampered device or application and the servers have no way knowing whether they were tampered or not.

4.6 Not Handling Errors (*Mis-Err*)

Sometimes SafetyNet Attestation fails to execute because the *GMS-process* encounters an error while performing the device integrity checks. When this happens an object containing error information is sent to the calling application (5). This object has no information about device integrity contained in JWS that is returned when SafetyNet Attestation call succeeds.

An application needs to handle these errors by trying SafetyNet Attestation again or following the protocol in case SafetyNet Attestation call was successful and integrity checks were not passed. If the errors continue, the server will be unable to obtain a valid JWS. An application that is not handling error correctly will continue to operate without being aware that the device or the application is tampered or not.

An attacker can leverage this by triggering errors that result in failure of execution of SafetyNet. For instance, by passing in bogus *API Key* to the Attestation API.

4.7 Null/Wrong API Key (*Mis-APIKey*)

SafetyNet Attestation API requires an API Key that is acquired from the Google APIs Console (2). If a wrong key or a Null value is passed to the API it returns an error instead of the JWS result. Applications using a Null/Wrong API Key and not handling the resulting error will never get the device attestation done and the application will continue normally in a tampered state.

4.8 Using Deprecated API (*Mis-DepAPI*)

SafetyNet Attestation has a deprecated API that always returns an error when it is used. Using the deprecated API (2) makes SafetyNet Attestation useless even if all other steps are followed correctly because the application will never receive a SafetyNet JWS.

4.9 Calling SafetyNet Only at First Launch (*Mis-Launch*)

SafetyNet is only useful if it is called (2) at appropriate time during an application's life cycle. For instance, when application is being launched or when some sensitive information is being handled like when performing transactions. Furthermore, applications need to make sure SafetyNet Attestation is performed every time these activities occur. If an application performs SafetyNet Attestation only at particular instances of these activities, then the Attestation is vulnerable to attackers. An attacker can launch the sensitive activity when SafetyNet Attestation is done in a non-tampered state so that SafetyNet checks pass. However, the attacker can tamper the device or the application when application decides not to do SafetyNet Attestation.

In practice, attackers may not have the freedom to arbitrarily tamper/un-tamper a device or app. However, we have found examples of applications that do SafetyNet Attestation only at their first launch. This makes it very easy for the attacker to run these applications in tampered state. The attacker only needs to launch the application once in a non-tampered state so that application thinks the device is safe to run on. Attacker can then tamper the device or the application for further execution of the application because the application will not be doing SafetyNet Attestation anymore.

5 ANALYSIS

In our study we analyze 163,773 Android applications for usage of SafetyNet Attestation. This section explains the process of dataset collection and the stages of automated analysis used to narrow down the dataset. These steps are performed to vet the applications in our dataset so the relevant applications can be analyzed manually. Countering application obfuscation, mentioned in Section 2, is a major challenge for our analysis. The steps for vetting the applications involve downloading the APK then performing static analysis then performing dynamic analysis and finally reverse engineering the applications manually. This process is shown in Figure 3. After each step, the list of the APKs left to be analyzed reduces. We will

refer to the different stages of the list of the APKs as a *Collection*. For example, the initial list of the APKs will be *Collection-1*. The remaining *Collections* are defined in their respective Sections and are also illustrated in Figure 3.

5.1 APK Collection

We collect APKs from the Google Play Store and a third-party marketplace called ApkPure [11]. In order to download APKs we need a list of their package names. We use web crawling to scrape package names from the Google Play Store's website. The package names are collected from top charts of every app category on Google Play Store.

After getting a list of the package names, we download the APKs from third-party websites and by using Android emulators and physical devices to get APKs from the Google Play Store. This list of the applications is named *Collection-1* in Figure 3.

All of the collected applications are available on the Google Play Store. However, to speed up the collection process, we also used third-party websites [10, 12] (which "mirror" the Google Play Store and allow direct downloading of the APK files). The usage of third party websites also allowed us to obtain apps that are not accessible to us on the Google Play Store due to regional restrictions.

5.2 Static Analysis

From *Collection-1* we want to identify applications that potentially perform SafetyNet Attestation. To this aim, we can search for applications that contain calls to SafetyNet API functions, such as *attest*. However, due to obfuscation (as explained in Section 2.2) this approach is not be reliable. For this reason, instead, we rely on code features that indicate usage of SafetyNet and are resilient to obfuscation.

To find such features, we reverse engineered the *GMS-client_library* (see Figure 1), and we found that the string 'safetynet' is always present in the applications that use the SafetyNet Attestation API, even when obfuscated. This term is present as a hard-coded string argument to an Android Bundle object's *get* function to access meta data about the application invoking SafetyNet API. The reason this feature is reliable is that being a Java string and not a class or method name, it is not changed by the ProGuard obfuscation.

Knowing this feature, we then proceed in searching for applications that contain the term 'safetynet.' In fact, such applications most likely contain the *GMS-client_library* code, which is used to handle calls to the SafetyNet API.

Concretely, we first decompile apps from our *Collection-1* dataset using apktool [1] to generate their smali code. Then, we perform an initial string search, using *grep* [27], on the decompiled smali code using "safetynet" as the search term. We use Docker [16] containers and Kubernetes [15] to make the analysis parallelized and improve its scalability. We called the resulting list of applications *Collection-2*.

5.3 Dynamic Analysis

Apps in *Collection-2* are then analyzed dynamically, on physical Android devices. The objective is to identify applications that are calling the SafetyNet Attestation API during their execution. To detect SafetyNet invocation dynamically, we need to find a method that is invoked when SafetyNet Attestation is performed so we can hook it using Frida. Frida allows hooking of functions while the application is running if method name and signature are provided.

As explained in the previous section, we cannot hook methods in the *GMS-client_library* or inside the client application's main code because, in most cases, they are obfuscated. Therefore, we need to find a method that we can reliably hook even after obfuscation.

Frida also allows hooking methods in the *GMS-process* (see Figure 1). Since these methods are not in an app's code, but in a system component, they keep the same name, regardless of the analyzed app. More precisely, while their names are also obfuscated, their obfuscated names remain the same, unless the *GMS-process* is modified (this may happen, for instance, during an operating system update, but it does not affect our analysis). In addition, since *GMS-process* is always running in background, hooking *GMS-process* is required only once and that hook can be used for every new application that is analyzed.

Therefore, we decided to detect when SafetyNet is called, by hooking methods within *GMS-process* that are always called when an app invokes SafetyNet. To identify these methods, we reverse engineered the code of the *GMS-process*, and we found a class called *AttestationData*, whose method names were not obfuscated.

Specifically, we need to find a method that is resilient to obfuscation and is invoked both at success and failure of SafetyNet Attestation. To this aim, we use manual dynamic analysis (helped by Frida) to identify a method that is always called when an app invokes SafetyNet Attestation. We will refer to this function as *SnetHook* (its real method name is obfuscated and changes across different operating system updates).

Knowing that *SnetHook* is always invoked when the SafetyNet Attestation API is called, we use it to identify those apps, among the apps in *Collection-2*, that call the SafetyNet API at runtime.

Specifically, each application in *Collection-2* is installed on a Google Pixel 3a and tested in the following way. *SnetHook* is hooked in the *GMS-process* using Frida. Then, after its installation, the application is launched and we automatically interact with it for about 10 seconds, as we will explain later. If an application tries to perform the SafetyNet Attestation, it will bef detected by *SnetHook* invocation. We name the resulting list of applications *Collection-3*. *Collection-3* contains applications that are dynamically confirmed to invoke the SafetyNet Attestation API. Applications that are not using SafetyNet are not further analyzed.

We hypothesized that most applications will perform attestation at launch time, since this is the behavior suggested by Google guidelines [40]. We further solidified our hypothesis by manually confirming that Attestation is performed at launch time in a subset of applications from *Collection-2*. We created this subset by considering 6 applications that other researchers mentioned as apps performing SafetyNet Attestation [31, 38].

We tested our dynamic analysis on this subset and 4 sample SafetyNet applications [6, 7, 45, 47] and we confirmed that our framework detected the SafetyNet invocation.

Nevertheless, to partially address applications that do not invoke the SafetyNet Attestation API immediately after they start, we use the adb and the Monkey [8] tools synergistically. Specifically, the Monkey tool is used to launch open user-reachable activities within the analyzed app, while adb is used for generating user input. The

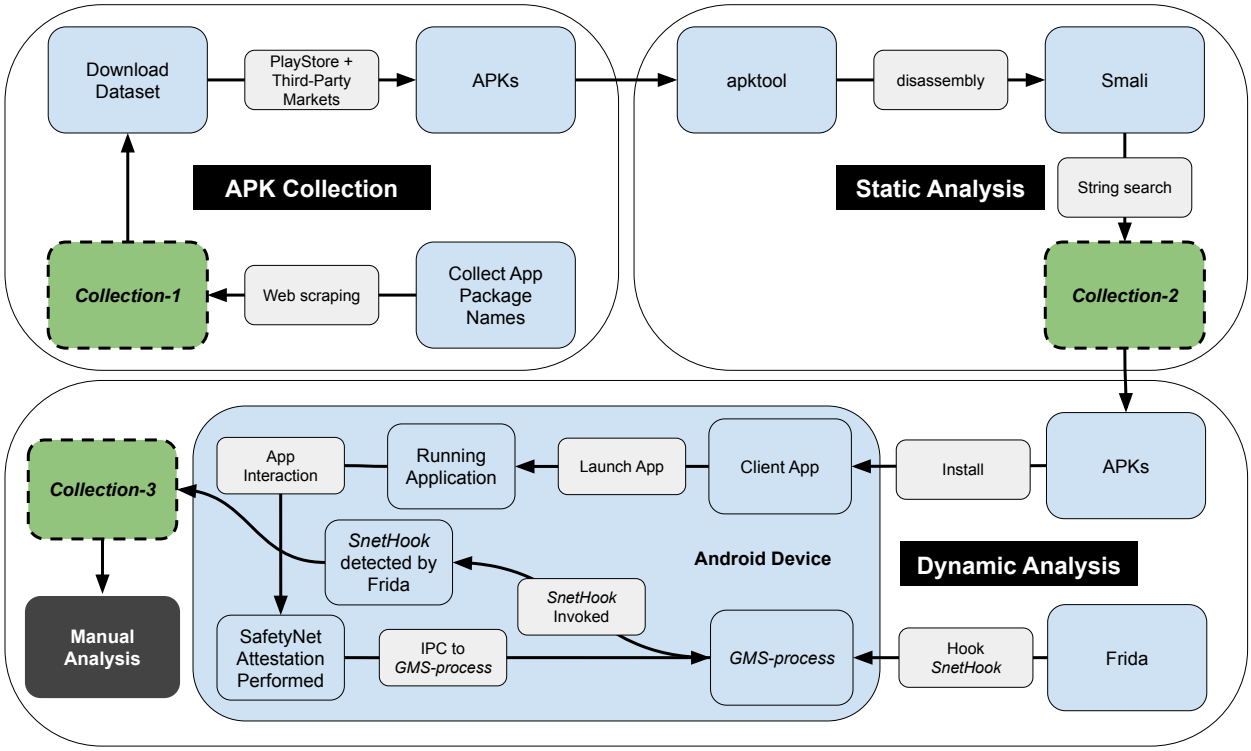

Figure 3: Automated Analysis Pipeline

configurations we used for this tools allow the Monkey tool to visit each activity multiple times during the analysis, while random clicks and swipes are performed on each activity using adb.

These design choices of our dynamic analysis are justified by the fact that generating user input with the Monkey tool can sometimes result in unwanted behaviors (such as the WiFi turning off, the device shutting down, or Android switching to another application). These unwanted behaviors can interfere with our analysis and with the triggering of the Attestation API. Therefore, we decided to use adb instead of Monkey for input generation.

Although effective for our goals in the majority of the analyzed apps, we acknowledge that our dynamic analysis has limitations, which can result in false negatives. We will discuss these limitations in Section 8.2,

5.4 Manual Analysis

The applications in *Collection-3* are further analyzed manually. The goal of the manual analysis is to see how the applications are using the SafetyNet Attestation results to detect tampering. By analyzing the usage of the Safety Attestation, we are be able to see if the applications contain any of the misusages mentioned in Section 4.

The applications are decompiled using *jadx* [4] to generate Java code. We then use string search on the generated Java code to narrow down the code we need investigate manually. String search includes looking for terms like 'basicIntegrity', 'ctsProfileMatch,' and 'safetynet'. These terms are used because they are included in the SafetyNet JWS object returned by the SafetyNet API, and they are likely to be used in class names, method names, and log/debug code involving SafetyNet.

In addition, we used *HttpCanary* [3] for network traffic analysis. *HttpCanary* is an Android application that intercepts network traffic of selected applications on an Android device. Network traffic is analyzed using *HttpCanary* to see what is being communicated with the application's server. By analyzing network traffic, we confirm if applications are sending the SafetyNet JWS to their backend server and observe how the server responds.

Although some of the misuages are mutually exclusive, we formulated a pipeline which allows to skip checks for certain misusages if specific conditions are met. In Figure 4, if any of the misusages or conditions shown in a box are met, then the misusages in the subsequent boxes do not need to be checked. For example, if the SafetyNet JWS is not being sent (box **2**) to any server then checks for *Mis-LoNon*, *Mis-Test*, *Mis-SerVeri*, and *Mis-PartJWS* (box **4** and **5**) are not needed. The misusages in box **4** and **5** involve server interaction so it does not make sense to check for box **4** and **5**. Likewise, finding *Mis-APIKey* or *Mis-DepAPI* means no other checks are needed because no SafetyNet JWS will be received by the application.

We name the list of applications in which the misusages were found as *Collection-4*.

Figure 5 details the steps of our manual analysis.s The details of how applications are put in misusage categories are given in the following paragraphs.

We find *Mis-LoCh* by searching for the strings included in the SafetyNet JWS, since those strings must be used to access the contents of the JWS. We analyze the Java code to see if an application

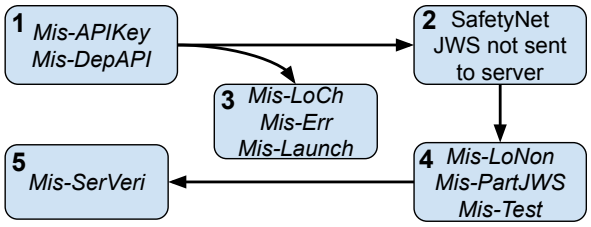

Figure 4: Misuse Analysis Pipeline

is accessing and using the JWS data only locally within the application. *Mis-LoCh* is also confirmed by analyzing the network traffic during the Attestation process to check if the SafetyNet JWS is present in the outgoing network traffic. If the SafetyNet JWS is not present in the outgoing network traffic and local checks are found, then the application is put in the *Mis-LoCh* category.

Mis-Err is checked by injecting wrong API Key and tracing the code using Frida. By injecting wrong API key, we cause SafetyNet to return an error. By Frida tracing, code that handles errors is located and analyzed. Applications are put in *Mis-Err* category if they are not following the steps mentioned in *Mis-Err* in their error handling code.

We search Java code for the Google's test server's url to locate *Mis-Test*. If the SafetyNet JWS is sent to Google's test server's url then the application is put in *Mis-Test* category.

Mis-LoNon is checked by tracing the nonce from the invocation of *attest* back to its origin. If the nonce is generated on the device and does not come from a server then the application is put in *Mis-LoNon* category.

Mis-Launch is checked by hooking *SnetHook* in the *GMS-process* and detecting invocation of *SnetHook*. The application is launched multiple times and if *SnetHook* invocation was detected only at first launch then the application is put in *Mis-Launch* category.

Mis-DepAPI is checked by hooking *SnetHook* in the *GMS-process* and seeing if an error is returned instead of the SafetyNet JWS. If there is always an error, the application code is reviewed manually to find the cause of such error. If the application is found to be using the deprecated API, the application was put in the *Mis-DepAPI* category.

Mis-SerVeri is checked by sending modified SafetyNet JWS to the application's backend server. The server's response is then captured and analyzed using *HttpCanary*. Only one specific field of the SafetyNet JWS is modified at a time to infer the checks that are being performed on the server.

To check if the server verifies the correctness of the nonce, we send to the server a previously generated SafetyNet JWS using Frida. By doing so, we change the nonce while keeping the JWS signature valid, effectively performing a replaying attack. The APK's signing certificate's hash is modified by disassembling the APK then recompiling and signing with a new key. If the server sends no response or is unable to detect modification in the SafetyNet JWS then the application is put in *Mis-SerVeri* category.

Mis-PartJWS is checked by manual Java code review. Specifically, we look for applications extracting Attestation results, such as the *basicIntegrity* boolean value, from SafetyNet JWS. If an application is sending only the extracted values instead of the whole SafetyNet JWS, the application is put in *Mis-PartJWS* category.

Mis-APIKey is checked by hooking a *GMS-client_library* class. Recall that this misusage corresponds of invoking the method `attest` with a Null or wrong API key. As previously explained, we cannot just hook the `attest` method, since this method may have been obfuscated. Therefore, we decided to hook a method (which we call `attest_inner`) within the *GMS-client_library*, which is indirectly called by the `attest` method, taking, as an argument the API key. While the method name of `attest_inner` can also change due to obfuscation, we designed a way to reliably detect it. Specifically, we noticed that this method is the constructor of a class containing a specific hard-coded string, which is not modified by obfuscation.

Therefore, we first search for this specific string, determine in which class it is used. The constructor of this class is `attest_inner`. By hooking it and checking one of its arguments we can dynamically determine the API key used to call the SafetyNet `attest` method.

5.5 Bypassing SafetyNet

After finding the misusages in the applications, the next step of our analysis is to confirm if the SafetyNet Attestation checks can be bypassed in those applications. We consider SafetyNet checks to be bypassed in an application if the application when running in a tampered device behaves the same than when running in an untampered device. To achieve this goal, we first need to find applications that stop running, show warnings, or behave differently based on the results they get when calling the SafetyNet API. For these applications, we study if an attacker can make them behave normally even in a system in which SafetyNet checks fail. To find applications that actually use SafetyNet Attestation data (i.e., applications that change their behavior based on SafetyNet results), additional vetting of *Collection-4* is required.

Specifically, we run these applications on three different Google Pixel 3a devices (named *dev0*, *dev1*, and *dev2*), with different configurations. These devices are as similar as possible in every aspect expect for the following properties: The device named *dev0* runs an unrooted version of Android. Therefore, in this device SafetyNet checks succeed. Whereas, both *dev1* and *dev2* fail SafetyNet checks. However, *dev1* is a device in which the *GMS-process* thinks that the device integrity is guaranteed. Therefore, when an app calls SafetyNet the returned results will show that the device integrity is ok. On the contrary, *dev2* is a device in which the *GMS-process* thinks that the device integrity is not guaranteed. Therefore, when an app calls SafetyNet the returned results will show that the device integrity is not ok.

To achieve these properties in *dev1* and *dev2* we use a rooting application called Magisk [5]. By using Magisk's hiding feature, in *dev1*, rooting and instrumentation tools are hidden from the *GMS-process*, so that SafetyNet checks pass in *dev1*.

After setting up the *dev0*, *dev1*, and *dev2*, we use them to test each application in *Collection-4*. Specifically, every application is run on these three devices to determine if it uses SafetyNet results.

In fact, an application using the SafetyNet API typically exhibits behavioral differences if it detects tampering. For instance, it could show error or warning messages, exit immediately after launch, or

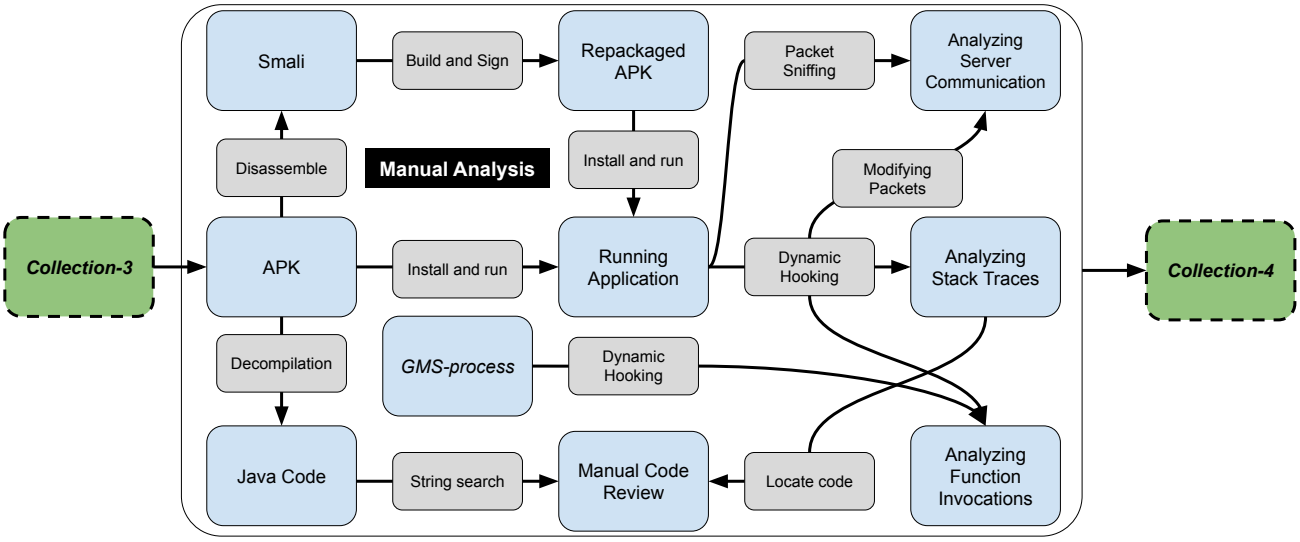

Figure 5: Manual Analysis Pipeline

disable some features. We consider an application to be behaving the same if it does not show any of the aforementioned behavioral differences when tested on the different devices (*dev0*, *dev1*, and *dev2*).

Applications behaving the same in all three devices are determined as *not using* SafetyNet results, nor checking for device integrity in any other way. These applications are discarded from further analysis. Applications behaving the same in *dev0* and *dev1*, but differently in *dev0* and *dev2* are, instead, applications that relay on SafetyNet results to decide how to behave. In fact, the only difference between *dev1* and *dev2* is the fact that in the former SafetyNet checks pass, while in the latter they do not. We put these applications in *Collection-5*. Therefore, *Collection-5* contains applications that actively uses SafetyNet results.

For some other applications, their behavior in *dev0* and their behavior in *dev2* are different. However, in these apps also their behaviors in *dev0* and *dev1* are different. We put these applications in *Collection-6*. We assume that applications in *Collection-6* are able to detect that a device is rooted using a mechanism that may not be SafetyNet (e.g., another mechanism way to detect rooting). These mechanisms were discussed in Section 1.

We then try to bypass SafetyNet checks for applications in *Collection-5*. SafetyNet checks are considered to be bypassed if we are able to achieve the same application behavior in *dev2* as in *dev0*. To bypass the checks, we modify SafetyNet JWS values and replay old SafetyNet JWS extracted from devices which pass SafetyNet checks. The SafetyNet JWS is spoofed by hooking the *getJwsResult* function using Frida and returning the modified JWS. In addition, applications with *Mis-launch* misusages are bypassed by strategically converting from *dev1* state to *dev2* state so the application does not perform Attestation in *dev2* state. Converting state refers to removing cloaking capabilities of Magisk so that *dev1* does not pass SafetyNet checks. In some applications functions performing local checks are hooked and bypassed using Frida.

We also manually analyze the applications in *Collection-6* to see if they are using both SafetyNet and other checks.

Details of the results of the manual analysis of applications in *Collection-5* and *Collection-6* will be provided in Section 6 and Section 7.

6 RESULTS

This section summarizes the results of our analysis. We analyzed a total of 163,773 applications (*Collection-1*) statically by string search and found 19,834 (12.11%) applications (*Collection-2*) that can potentially have SafetyNet code. After testing 19,834 applications dynamically, SafetyNet Attestation invocations were detected in 62 (0.31%) applications (*Collection-3*). The results are shown in Table 1:

Collection	Apps	Collection	Apps
Collection-1	163,773	*Collection-4*	62
Collection-2	19,834	*Collection-5*	19
Collection-3	62	*Collection-6*	14

Table 1: Automated Analysis Results

After analyzing each of the applications from *Collection-3* manually, we found all of the applications have at least one type of misuse mentioned in Section 3. There were a total of 62 applications in *Collection-4*. The distribution of misusage categories is shown in Table 2.

The applications from *Collection-4* were vetted for bypassing SafetyNet checks. After vetting, there were 19 applications (*Collection-5*) that behaved differently on *dev0* and *dev1* but the same behavior in *dev0* and *dev2*. SafetyNet checks were bypassed in 16 applications from *Collection-5*. The remaining 3 could not be bypassed because they were additionally using native code and alternate methods to do the device checks. There were 14 applications from *Collection-4* that behaved differently in *dev0* and *dev2*, and also behaved differently in *dev0* and *dev1*. These applications (*Collection-6*) were

Misuage	Apps	Misuage	Apps
Mis-LoCh	32 (51.6%)	*Mis-Err*	20 (32.3%)
Mis-Test	1 (1.6%)	*Mis-APIKey*	2 (3.2%)
Mis-LoNon	11 (17.7%)	*Mis-DepAPI*	0 (0.0%)
Mis-SerVeri	11 (17.7%)	*Mis-Launch*	14 (22.6%)
Mis-PartJWS	4 (6.5%)		

Table 2: Manual Analysis Results

performing their own checks in addition to SafetyNet Attestation. We were able to bypass SafetyNet and their own checks in 5 of these applications. Out of remaining 9, 4 were using native code and other 5 were not relying on the results of SafetyNet to detect tampering. These results are shown in Table 3.

Collection	Total	Bypassed
Collection-5	19	16
Collection-6	14	5

Table 3: Bypassing Results

Table 4 shows the categories of applications in *Collection-3*.

Category	No. of Apps
Travel and Navigation	8
Food and Restaurant	12
Medical and Fitness	2
Banking and Finance	6
Shopping	7
Security and Authentication	6
Communication	7
Social and Entertainment	14

Table 4: Categories of the apps in *Collection-3*

7 CASE STUDIES

This section takes a closer look at the applications that used SafetyNet Attestation to alter their applications' behavior. SafetyNet checks in all of these applications were bypassed dynamically by using Frida hooks.

7.1 Punchh Library

Ten applications were using a third-party library that handled the SafetyNet Attestation. We will refer to the library as Punchh Library because of the package name found in decompiled code. As an example, we are going to look at a restaurant application called Del Taco. The situation is similar for other 9 applications which are also restaurant apps. Del Taco only performs local checks (*Mis-LoCh*) that are present in the Punchh Library and no data is sent to the server. The SafetyNet JWS is not used anywhere in Del Taco expect within the Punchh Library code. The pseudo code of local checks in Del Taco is as follows:

```
if (SafetyNetJWS.hasBasicIntegrity) {
    continue application execution
} else {
    show error and stop application
}
```

The "if" condition which checks *basicIntegrity* boolean was bypassed using Frida. Besides that, Punchh Library only checks *basicIntegrity* boolean locally. Checking only *basicIntegrity* makes it even easier for attackers to run the application in tampered state. Because *basicIntegrity* is "less strict" than *ctsProfileMatch* and remains "true" even with unlocked bootloader, custom ROMs, and uncertified devices. Ideally an application should be checking for both *basicIntegrity* and *ctsProfileMatch*. Also, Del Taco is generated a nonce locally (*Mis-LoNon*) instead of getting it from the server.

7.2 Mis-APIKey

Two applications were passing in Null as the API Key to the *attest* function. This causes *GMS-process* to return a NETWORK_ERROR whenever the applications try to perform SafetyNet Attestation. One of the applications is PayPal Mobile Cash app that has more than 100 million downloads. Other application is DigiLocker that has more than 10 million downloads.

7.3 TextNow

TextNow is an SMS (short message service) texting application. TextNow's backend server was tested by changing values in JWS like nonce, apkCertificateDigestSha256 and ctsProfileMatch. Server responded correctly to changes by sending errors like "Incorrect Nonce" when nonce was changed and "Incorrect Signature" when JWS blob was modified. One field that server did not check correctly was *apkCertificateDigestSha256*. apkCertificateDigestSha256 is the hash of application's signing certificates that can be used to verify if the application was modified and repackaged. TextNow was repackaged and signed with a different key. This changed apkCertificateDigestSha256 of the APK. We will refer to this APK as *apkNewKey*.

TextNow was using API Restriction from Google Play Console to limit usage of their SafetyNet API key only to applications signed by their key. This was bypassed for *apkNewKey* by overwriting "X-Android-Cert" HTTP Header in *GMS-process* with the original apkCertificateDigestSha256. Now SafetyNet JWS for TextNow has apkCertificateDigestSha256 of *apkNewKey*. This SafetyNet JWS was sent to TextNow's backend server which was not able to detect this modification. Thus, this is a case of *Mis-SerVeri*.

TextNow also had *Mis-Launch* because the application only performed attestation at first launch. The application was launched one time in non-tampered state and TextNow's backend server received a SafetyNetJWS that showed device is not tampered. Then the device was tampered, and application was launched again. This time application did not perform SafetyNet Attestation, so server was unaware that the application is running on a tampered device.

7.4 Geon

Geon is an augmented reality application. Geon allows you to earn gift cards and vouchers by doing tasks. Geon sends only boolean values (*Mis-PartJWS*) of *basicIntegrity* and *ctsProfileMatch* instead of sending the whole JWS to its backend server. This makes spoofing the Attestation result trivial by just changing the boolean values sent to the server. Geon also tries to do local checks (*Mis-LoCh*)

which can be bypassed using Frida code injection and generates local nonce (*Mis-LoNon*).

8 LIMITATIONS AND FUTURE WORK

8.1 Static Analysis

Our static analysis can deal with default obfuscation provided by ProGuard. If some other technique that obfuscates hard-coded strings (such as the "safetynet" string) is used then our methodology will fail. Since ProGuard is the default obfuscation provided by Android Studio, we believe our analysis methodology can deal with the majority of the obfuscated apps. Around 60% of the application from *Collection-2* use ProGuard to obfuscate the Attestation API, while the rest do not obfuscate the Attestation API.

As future work, our analysis can be improved by comparing signatures of methods involved in the API with the method signatures present inside the application. Call graphs and method signatures can be used to identify specific API usages without relying on method names (which can be obfuscated). In particular, the detection of a specific path in the call graph and the usage of methods with specific signatures could suggest the usage of the SafetyNet Attestation API.

The significant difference in the number of applications from *Collection-2* to *Collection-3* is due to presence of the "safetynet" keyword in contexts other than the Attestation API. The keyword is contained in the GMS library, which is included in any application that uses APIs from this library and there are other APIs, besides the Attestation API (such as the Safe Browsing API [23]) which also include this keyword. If an application uses any of the SafetyNet APIs, it will be detected in our analysis. This results in false positives and comparatively large number of applications in *Collection-2*. However, these false positives are handled in the subsequent steps of the analysis pipeline.

8.2 Dynamic Analysis

Our dynamic analysis relies on the application invoking SafetyNet during our automated interaction. If the application invokes SafetyNet at later phase during its execution, then the invocation will not be detected by our automated analysis pipeline. Specifically, if the Attestation is performed on activities that are only reached after specific user input is entered like user name and password. Our analysis does not generate complex user input (like typing credentials) and relies on random clicks and swipes.

As explained in 5.3, we think the Attestation API is usually invoked immediately after an application is launched. However, when this property is not true, our analysis could result in false negatives.

We acknowledge that our methodology has scaling limitations due to the manual steps in the analysis pipeline. However, as future work, it is possible to automate the manual parts using static analysis. For instance, detection of *Mis-APIKey* could be automated by performing data flow analysis and determining the value of the API key given to the `attest` method. Similarly, *Mis-LoNon* and *Mis-PartJWS* could be automatically detected by performing taint analysis on the control flow graph, looking for paths between specific sources and sinks (such as methods for generating random bytes and parsing JWS data structures).

Likewise, *Mis-Test* could be detected automatically by a combination of data flow and taint analysis. Specifically, checking if the SafetyNet JWS is being sent to a network API that is using the test server's address as the destination.

8.3 Bypassing SafetyNet

Seven applications from *Collection-4* were using Reactive and Native code to detect tampered devices/applications. Since SafetyNet Attestation is supposed to be used in Java code, analyzing those applications and their detection methods is out of scope for this paper.

Four applications from *Collection-4* were using detection mechanisms in addition to SafetyNet Attestation and those could not be bypassed. Since in this paper we focus on SafetyNet, we did not reverse engineer enough to bypass the additional checks. Their exploits can be part of a future work.

9 RELATED WORK

9.1 Security API Misusage

Many previous works analyze API misusage and failure of following proper security practices in Android but "to the best of our knowledge" there is no previous work on the usage of SafetyNet Attestation. Bianchi et al. [14] look at the how applications use the Fingerprint API in Android. The authors identify ways in which the API can be misused, and their findings indicate that only 1.8% of the apps use the API in a secure manner. Ghafari et al. [20] develop a static analysis tool for detecting security mistakes in android apps that are a result of bad coding practices.

Mahmud et al. [33] looks at how Android applications are handling credit card information. They develop a static analysis tool, Cardpliance, using Amandroid [48] to check PCI DSS compliance of Android applications.

A tool for statically detecting SafetyNet misusages can be part of a future work.

Egele et al. [17], Shuai et al. [43], and Muslukhov et al. [36] look at misusage of cryptographic APIs in Android. Fahl et al. [18] and Oltrogge et al. [9] analyze how misuse of SSL/TLS APIs can make Android applications vulnerable to Man-In-The-Middle attacks.

Some of these works use static analysis, other dynamic analysis. In our work we decided to use a combination of both because we wanted to confirm our findings dynamically and performing static analysis beforehand resulted in efficient automation.

9.2 Device and App Integrity

There has been research on device and application integrity checks but there has not been a comprehensive work focusing specifically on the SafetyNet API. Kim et al. [28] look at how finance applications perform integrity checks. In their work, they analyze checks that applications have implemented locally by themselves or by using third-party libraries. As mentioned in Section 4.1, local checks are not reliable for detecting tampering. These checks create the same situation as in *Mis-LoCh* and can be easily bypassed. Our key argument is that the verification of attestation results on the server side and SafetyNet (if used correctly) allow to implement these checks safely, otherwise the attestation will not be reliable.

Li et al. [30] do a literature review of Android application repackaging detection tools. The authors point out the current tools for repackage detection are not scalable and are only suitable for closed datasets. The goal of this work is to reboot research in repackage detection. Merlo et al. [34] leverage native code for performing app integrity checks. Merlo et al. [35] look at current anti-repackaging techniques and how to circumvent them.

Berlato et al. [13] look at anti-tampering techniques in Android. In this work, the authors fingerprint various anti-tampering methods including SafetyNet Attestation and statically detect their usage in applications. Their work focuses on finding which applications are using these techniques. However, this work does not investigate if the applications have used these techniques correctly or not. Our work integrates their findings by also analyzing how the applications use the SafetyNet API.

Zungur et al. [51] present a framework for dynamically investigating anti-tampering mechanisms in Android applications. In their work, the authors leverage UI to assess the application state in different tampered environments (using non-tampered environment as a baseline). Their framework reports the tampered environment against which an application is defending rather than the defense APIs/mechanisms themselves.

9.3 Bypassing Root Checks

Other works look deeply at rooting techniques and how they can be evaded. Soewito et al. [44] analyze rooting checks in Android apps and bypasses them using Frida hooking. DroidRanger [50], Google Bouncer [32], RootExplorer [19], and Zhou et al. [49] statically fingerprint Android applications that can possibly contain rooting payload. PREC [26] looks at mitigating root exploits dynamically.

Sun et al. [46] look at how different Android components are exploited to achieve rooting on a device. They look at applications like *SuperSu* that automate the rooting process and analyze how applications in the market detect rooting. They identify different methods that applications are using to detect root which include checking Build Tag of Android Image, various Shell Commands, running processes, installed packages, System Properties, and directory permissions. In this work, they develop a tool named *RDAnalyzer* to bypass all of these root checking mechanisms by API hooking. Nguyen Vu et al. [37] did a similar study on root detection and evasion, the authors identified two types of rooting: hard rooting and soft rooting. In both works the authors suggest the usage of kernel level root checks to mitigate evasion. Sun et al. [46] suggest use of trusted execution environments (TEE) to perform the checks so they cannot be bypassed on a rooted device. In line to what is proposed by these works, SafetyNet uses system-level components to perform integrity checks and Google suggested that it is planning to use TrustZone-enforced checks [41] for SafetyNet Attestation.

10 DISCUSSION

Our results reveal that *Mis-LoCh* is the most found misusage. We speculate that one of the reasons behind this issue is developers' reliance on third-party libraries to verify app and device integrity. These libraries, such as the aforementioned Punchh library (see Section 7.1), can only provide code performing checks on the client device the apps run into. This is because these libraries do not have access to the applications' backend servers so developers. Therefore, apps using these libraries only rely on the checks that these libraries perform locally.

We also found that Google and other third parties offer sample applications [6, 7] to exemplify the usage of the SafetyNet Attestation API. By analyzing these sample applications, we found that they also only verify the SafetyNet results locally. Therefore, we speculate that one of the reasons of the *Mis-LoCh* misusage is developers following the code of these sample applications.

As of now, Google only provides server-side code for verification in two languages (C# and Java) [42]. Providing server-side verification code in additional languages will ease the correct usage of the SafetyNet API. In addition, warnings can be shown in Android Studio that alert the developers when the Attestation API is not used correctly. Specifically, developers can be warned about sending the SafetyNet JWS to their server and a link to the sample verification code can be provided in the warning. This analysis could be integrated in the existing "code inspection" feature that is already provided by the Android Studio Lint tool [22].

11 CONCLUSION

In this paper we performed a comprehensive analysis of the usage of the SafetyNet Attestation API in Android applications. First, we systematically identified all the possible ways in which the SafetyNet API can be misused. Then we performed a large-scale study starting from 163,773 Android apps. Among these apps, we identified 21 applications that rely on SafetyNet to detect tampering. The analysis of these apps reveals that none of them use the SafetyNet API correctly. Due to these mistakes, the checks performed by these apps can be bypassed.

ACKNOWLEDGMENTS

We are grateful to our shepherd for their support and suggestions.

This material is based upon work supported by the NSF under Award number CNS-1949632. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the NSF.

REFERENCES

[1] Apktool. https://ibotpeaches.github.io/Apktool/.
[2] Frida. https://frida.re/.
[3] Httpcanary. https://github.com/MegatronKing/HttpCanary.
[4] Jadx. https://github.com/skylot/jadx.
[5] Magisk. https://github.com/topjohnwu/Magisk.
[6] Safetynet helper sample. https://github.com/scottyab/safetynethelper/.
[7] Safetynet sample. https://github.com/googlesamples/android-play-safetynet/tree/master/client/java/SafetyNetSample.
[8] Ui/application exerciser monkey. https://developer.android.com/studio/test/monkey.
[9] Why eve and mallory still love android: Revisiting TLS (in)security in android applications. In *30th USENIX Security Symposium (USENIX Security 21)*, Vancouver, B.C., 2021. USENIX Association.
[10] APKMirror. Apkmirror. https://www.apkmirror.com/.
[11] APKPure. Apkpure. https://apkpure.com/.
[12] Aptoide. Aptoide. https://en.aptoide.com/.
[13] Stefano Berlato and Mariano Ceccato. A large-scale study on the adoption of anti-debugging and anti-tampering protections in android apps. *Journal of Information Security and Applications*, 52, 06 2020.
[14] Antonio Bianchi, Yanick Fratantonio, Aravind Machiry, Christopher Kruegel, Giovanni Vigna, Simon Pak Ho Chung, and Wenke Lee. Broken Fingers: On the Usage of the Fingerprint API in Android. In *Proceedings of the Annual Network & Distributed System Security Symposium (NDSS)*, 2018.
[15] CNCF. Kubernetes documentation. https://kubernetes.io/docs/home/.

[16] Docker. Docker overview. https://docs.docker.com/get-started/overview/.
[17] Manuel Egele, D. Brumley, Y. Fratantonio, and C. Krügel. An empirical study of cryptographic misuse in android applications. *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, 2013.
[18] Sascha Fahl, Marian Harbach, Thomas Muders, Lars Baumgärtner, Bernd Freisleben, and Matthew Smith. Why eve and mallory love android: An analysis of android ssl (in)security. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, New York, NY, USA, 2012. Association for Computing Machinery.
[19] Ioannis Gasparis, Zhiyun Qian, Chengyu Song, and Srikanth V. Krishnamurthy. Detecting android root exploits by learning from root providers. In *26th USENIX Security Symposium (USENIX Security 17)*, Vancouver, BC, August 2017. USENIX Association.
[20] M. Ghafari, P. Gadient, and O. Nierstrasz. Security smells in android. In *2017 IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2017.
[21] Google. Android debug bridge (adb). https://developer.android.com/studio/command-line/adb.
[22] Google. Improve your code with lint checks. https://developer.android.com/studio/write/lint.
[23] Google. Safetynet safe browsing api. https://developer.android.com/training/safetynet/safebrowsing.
[24] Google. Safetynet attestation api. https://developer.android.com/training/safetynet/attestation, 2020.
[25] Guardsquare. Guardsquare/proguard. https://github.com/Guardsquare/proguard.
[26] Tsung-Hsuan Ho, Daniel Dean, Xiaohui Gu, and William Enck. Prec: Practical root exploit containment for android devices. CODASPY '14, New York, NY, USA, 2014. Association for Computing Machinery.
[27] Michael Kerrisk. grep linux manual page. https://man7.org/linux/man-pages/man1/grep.1.html.
[28] Taehun Kim, Hyeonmin Ha, Seoyoon Choi, Jaeyeon Jung, and Byung-Gon Chun. Breaking ad-hoc runtime integrity protection mechanisms in android financial apps. 04 2017.
[29] John Kozyrakis. Safetynet: Google's tamper detection for android. https://koz.io/inside-safetynet, 2015.
[30] L. Li, T. F. Bissyande, and J. Klein. Rebooting research on detecting repackaged android apps: Literature review and benchmark. *IEEE Transactions on Software Engineering*, 2019.
[31] Brad Linder. Some apps may stop working on rooted android phones due to safetynet update. https://liliputing.com/2020/03/some-apps-may-stop-working-on-rooted-android-phones-due-to-safetynet-update.html.
[32] Hiroshi Lockheimer. Android and security. http://googlemobile.blogspot.com/2012/02/android-and-security.html, Feb 2012.
[33] Samin Yaseer Mahmud, Akhil Acharya, Benjamin Andow, William Enck, and Bradley Reaves. Cardpliance: PCI DSS compliance of android applications. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, August 2020.
[34] Alessio Merlo, Antonio Ruggia, Luigi Sciolla, and Luca Verderame. Armand: Anti-repackaging through multi-pattern anti-tampering based on native detection, 2020.
[35] Alessio Merlo, Antonio Ruggia, Luigi Sciolla, and Luca Verderame. You shall not repackage! demystifying anti-repackaging on android. 2020.
[36] Ildar Muslukhov, Yazan Boshmaf, and Konstantin Beznosov. Source attribution of cryptographic api misuse in android applications. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, ASIACCS '18, New York, NY, USA, 2018. Association for Computing Machinery.
[37] Long Nguyen Vu, Ngoc-Tu Chau, Seongeun Kang, and Souhwan Jung. Android rooting: An arms race between evasion and detection. *Security and Communication Networks*, 2017, 10 2017.
[38] Stephen Perkins. Completely hide root using magisk. https://android.gadgethacks.com/how-to/completely-hide-root-using-magisk-0201243/.
[39] Mishaal Rahman. Safetynet's dreaded hardware attestation is rolling out, making it much harder for magisk to hide root. https://www.xda-developers.com/safetynet-hardware-attestation-hide-root-magisk/.
[40] Oscar Rodriguez. 10 things you might be doing wrong when using the safetynet attestation api. https://android-developers.googleblog.com/2017/11/10-things-you-might-be-doing-wrong-when.html.
[41] Google SafetyNet API Clients team. Feature preview: Safetynet attestation api evaluationtype. https://groups.google.com/g/safetynet-api-clients/c/lpDXBNeV7Fg?pli=1.
[42] Google Samples. android-play-safetynet. https://github.com/googlesamples/android-play-safetynet/tree/master/server.
[43] S. Shuai, D. Guowei, G. Tao, Y. Tianchang, and S. Chenjie. Modelling analysis and auto-detection of cryptographic misuse in android applications. In *2014 IEEE 12th International Conference on Dependable, Autonomic and Secure Computing*, 2014.
[44] Benfano Soewito and Agung Suwandaru. Android sensitive data leakage prevention with rooting detection using java function hooking. *Journal of King Saud University - Computer and Information Sciences*, 2020.
[45] SoftGuide. Safetynet check. https://play.google.com/store/apps/details?id=com.softguide.safetynetcheck.
[46] San-Tsai Sun, Andrea Cuadros, and Konstantin Beznosov. Android rooting: Methods, detection, and evasion. In *Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '15, New York, NY, USA, 2015. Association for Computing Machinery.
[47] Free Android Tools. Safetynet test. https://play.google.com/store/apps/details?id=org.freeandroidtools.safetynettest.
[48] Fengguo Wei, Sankardas Roy, Xinming Ou, and Robby. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. 21(3), April 2018.
[49] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *2012 IEEE Symposium on Security and Privacy*, 2012.
[50] Y. Zhou, Zhi Wang, W. Zhou, and X. Jiang. Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. In *NDSS*, 2012.
[51] Onur Zungur and Antonio Bianchi. Appjitsu: Investigating the resiliency of android applications. 2021.