Tamper Detection Scheme Using Signature Segregation on Android Platform

Jiwoong Bang[1], Haehyun Cho[2], Myeongju Ji[3], Taejoo Cho[4], and Jeong Hyun Yi[5]*
Department of Computer Science and Engineering, Soongsil University
Seoul, 156-743, Korea
{[1]jiwoongbang, [2]haehyuncho, [3]wlaudwn007, [4]taejoo90}@gmail.com, [5]jhyi@ssu.ac.kr

Abstract— As Android apps are vulnerable to repackaging attacks, in order to protect these app, a code that runs a tamper detection function is inserted and obfuscated to protect the inserted function while or after building the app. However, with the use of currently released app analysis tools, many tamper detection methods are rendered ineffective. Therefore, it can be used for repackaging attack. To protect against these weaknesses, in this paper we propose the APK Attester scheme that detects app tampering on the platform in order to provide a secure app running environment for users in defense against repackaging attack.

Keywords— Mobile security; anti-reverse engineering

I. INTRODUCTION

Android app can be registered or distributed by anyone due to the market policy and self-signing. In other words, the attacker can download a app from the market and use apktool to confirm the app's code and modify it into malicious apps. After modifying the app, the attacker can sign it instead of the original developer and then distribute it. To complement this vulnerability many apps such as, financial apps or apps handling private information are applied with obfuscation and tamper detection techniques.

Such tamper detection methods usually consist of using an app's hash value or signature information to insert code that detects tampering. In addition, by obfuscating the inserted code, the tamper detection code will be protected from repackaging attacks[1]. However, with analysis of obfuscated apps made possible through the use of various released dynamic analysis tools, many obfuscation tools are rendered futile. With results from those tools the attacker can render tamper detection routines or manipulate results to look like normal apps, so the existing obfuscation tools reveal a limitation in app protection[2]. Furthermore to release tampered app, attackers must resign it. However, since the attacker cannot know the developer's private key, the signature value will always be different. One can recognize the tampered app via the sign key, but general user cannot recognize it.

To compensate this problem, we propose in this paper the APK Attester scheme[3][4] that detects app tampering on the platform in order to provide a secure app running environment for users in defense against repackaging attack.

* Corresponding author

II. RELATED WORK

A. Code Obfuscation

Android apps are obfuscated to protect them from repackaging attacks conducted by self-signing. Obfuscation methods include renaming, string encryption, control flow obfuscation, class encryption, and etc. Android SDK fundamentally provides Proguard's renaming method which can be applied. However, codes only applied with renaming method can be easily understood. Therefore, apps handling private information or financial apps need to apply additional obfuscation tools. Major obfuscation tools are DexGuard[5], DexProtector[6], Stringer[7], Allatori[8], and etc.

B. Tamper Deteaction

Tamper detection is a technique to prevent the execution of a tampered app to verify the integrity of the app. A common method is inserting a code which examines the hash or signature value to decide whether the app is tampered or not. The original hash value, etc. are stored in a server, device, or a string inside the code. However, apps uniquely applied with tamper detection can be bypassed by changing the result where the tamper detection is checked to always be true. Thus, more obfuscation must be a put in to protect tamper detection.

C. APK Repackaging

APK repackaging is modifying smali codes decompiled with tools such as apktool, baksmali, and etc. After inserting additional codes, it is compiled with JarSigner which is the signing process. Fig 1 show the process of app repackaging attack.

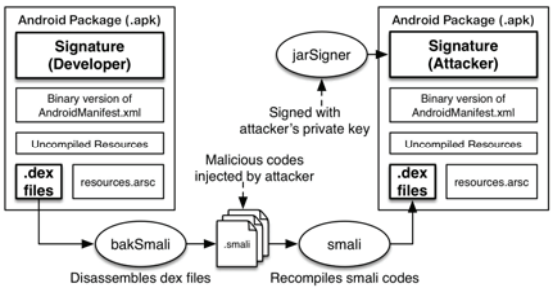

Fig. 1. Process of android repackaging attack

As shown above, repackaging attack is possible because Android apps can be distributed through self-signing. To identify whether the app is repackaged or not, the most accurate way is confirming the signature value. Since the attacker cannot know the developer's key, the attacker must sign the app with a different key. Thereby confirming the signature value is the accurate way to determine whether the app is repackaged or not.

III. System Design

The purpose of the APK Attester proposed in this paper is to provide a secure execution environment by detecting whether the app running on the Android platform has been tampered with or not. Fig 2 shows the overall architecture of the APK Attester.

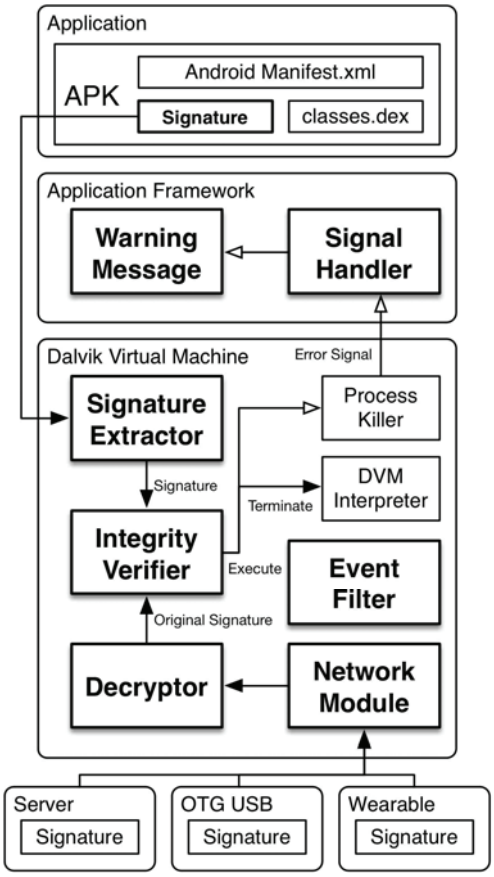

Fig. 2. Architecture of proposed system

A. Signature Extractor

When the app is ran, the Signature Extractor module extracts the executed app's identification information from the APK file installed in the device. The app's identification information uses the app's signature information and hash value. Because one's signature is obligatory required when an app's code is modified, an attacker without the key of the original developer cannot restore the identification information of the original app. In addition, because the app's hash value is altered when code is modified, this also can be purposed as identification information used for tamper detection. Extracted identification information is sent to the Integrity Verifier module.

B. Event filter

Event Filter is a module for monitoring the execution of the app. Also, it sends execution information and device's unique information, to the Network Module once it is executed. When platform applied with APK Attester, the app cannot prevent from running Event Filter in the app level. Because Dalvik Virtual Machine is modified to execute Event Filter when codes are executed. Therefore, devices applied with APK Attester cannot render tamper detection routines.

C. Network Module

Network module can be divided into two parts. First, when a app is ran, it sends execution information and device's unique information received from the Event Filter. After that it receives the app's original unique information encrypted with the device's unique information. Then, the encrypted data is sent to the Decrypted module. This process can be applied to not only server but also wearable devices, OTG USB, etc.

D. Decryptor

The encrypted original identification information received from the network module is decrypted using the Decryptor Module. Decryptor module decrypts the encrypted app identification information with the app's unique information. Then, the decrypted original identification information is transmitted to the Integrity Verifier.

The encrypted original identification information received from the network module is decrypted using the Decryptor Module. The Decryptor Module decrypts it with the device's unique information. Then, the decrypted original identification information is transmitted to the Integrity Verifier.

E. Integrity Verifier

It compares the executed app's identification information and the original identification information and determines whether the app has been tampered with or not. In the case that, when comparing the two pieces of information, the two values differ, the app is determined to have been tampered and is forced terminated using Process Killer. Moreover, a signal is sent using Process Killer. Because Integrity Verifier is executed when Dalvik VM is initialized, an app can't run unless the app's integrity is confirmed.

F. Signal Handler

It manages the app's termination that results from detection of tampering. When the signal is sent, the package name of the terminated app is sent to the Warning Message.

G. Warning Message

When the package name is received from the Signal Handler, a warning message, along with the package name, is output and the user is warned that the app has been tampered and, thus, terminated.

H. Server

Server is contained with app's package name, version information, and unique identification information corresponding to the version. When the server receives app execution information and device's unique information from the device applied with APK Attester, it searches app's original identification information and encrypts the information with the device's unique information. The encrypted app's original identification information is sent back to the device.

When a device adjusted with the proposed method runs a app, it sends app's identification information and device's unique information to the server. The server confirms the received app's identification information and checks the original identification information. Then, sends it to the device after encrypting it with the device's unique information. Afterwards when the device receives the encrypted data, it decrypts it again and checks whether the app is tampered or not. If the app is not tampered, the app is executed normally. On the other hand, if the app is tampered, a signal is triggered. As a result, the apps is killed and warns the user. Fig 3 is an overall flow of APK Attester.

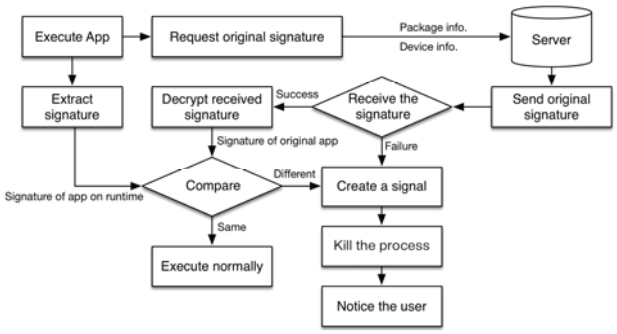

Fig. 3. Overall flow of the proposed system

Because APK Attester does not store within the device data that can determine whether or not an app has been tampered, it does not need additional obfuscation and provides safety from repackaging attacks that utilize data fabrication. Furthermore, because original information used to determine whether an app has been tampered can be stored in OTG USB and wearable devices in addition to the server, the Attester can provide a secure execution environment even in environments where connection with the server is limited.

IV. Experimental Results

A. Implementation

In order to construct the APK Attester proposed in this paper, we modified the AOSP (Android Open Source Project) Android 4.4.4 kitkat version. The modified source was built as a hammerhead image and flashed onto a device. Extractor, Integrity Verifier, Decryptor, and Network Module were developed and, by modifying the Android source code, Signal Handler and Warning Message were developed.

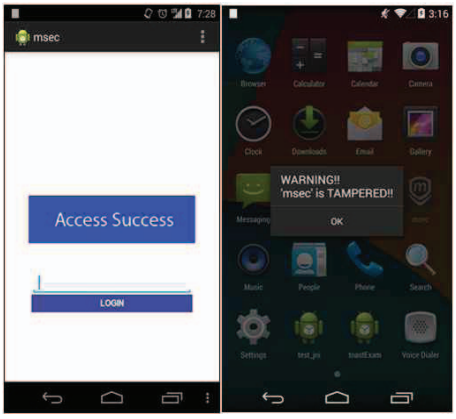

Fig. 4. Execution Result

B. Results

The result of tamper detection via the use of the APK Attester is shown in Fig 4. An app that had been running normally being repackaged and rerun result in the app's identification information now differing. We were able to confirm that the app was forced terminated and a warning message was sent.

C. Limits

Currently APK Attester basically examines the app based on the app's package name. When the attacker changes the packages name and codes into a executable code, and then distribute it, APK attester cannot tamper detect the app.

V. Discussion

APK Attester has a limitation, that it can only be applied by the manufacturer because the android platform source must be modified. However, contrary to this limitation, is that common users cannot verify the codes. Also, in the app level, since the app is unavailable to render tamper detection routine, it has an advantage of providing a more secure environment than inserting additional tamper detection code inside the app's code.

VI. Conclusion

Existing tamper detection methods required the insertion of tamper detection code in apps. Additionally, additional obfuscation tools for protecting the inserted code needed to be applied. However, the APK Attester proposed in this paper detects tampering of an app on the Android platform and, thus, is able to provide a safe execution environment, secure from repackaging attack, without revealing the tamper detection code and without additional obfuscation.

In an IOT environment where many devices are connected to one another, the smartphone acts as the gateway. Therefore, a secure execution environment for smartphones will have a direct role in creating a safe environment for other IOT devices. As a result, instead of the already ineffective existing tamper detection methods, platform level tamper detection methods such as APK Attester will be need to be used.

ACKNOWLEDGMENT

This research was supported by Global Research Laboratory (GRL) program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT, and Future Planning (NRF-2014K1A1A2043029).

REFERENCES

[1] Jin-Hyuk Jung, Ju Young Kim, Hyeong-Chan Lee, and Jeong Hyun Yi, "Repackaging Attack on Android Banking Applications and Its Countermeasures", Journal of Wireless Personal Communications, Journal of Wireless Personal Communications, Vol. 2014, Article ID:237125, Mar. 2014.

[2] Piao YX, Jung JH, Yi JH. Deobfuscation analysis of DexGuard code obfuscation tool. The International Conference on Computer Applications and Information Processing Technology, 2013; 107–110.

[3] Jeong Hyun Yi, Myeongju Ji, Jiwoong Bang and Taejoo Cho, "User Terminal to Detect the Tampering of the Applications Using Signature Information and Method for Tamper Detection Using the Same", PCT/KR2015/002200, March 2015.

[4] Jeong Hyun Yi, Jiwoong Bang and Taejoo Cho, "User Terminal to Detect the Tampering of the Applications Using Hash Value and Method for Tamper Detection Using the Same", PCT/KR2015/002198, March 2015.

[5] DexGuard. http://www.saikoa.com/dexguard

[6] DexProtector, http://www.dexprotector.com

[7] Stringer, https://jfxstore.com/stringer/

[8] Allotori. http://www.allatori.com/