Journal of Engineering and Applied Sciences 13 (6): 1567-1574, 2018
ISSN: 1816-949X
© Medwell Journals, 2018

Forged Android Mobile Apps. Detection System with Server-Side Signature Verification Method

Jaekyu Lee and Hyung-Woo Lee
Division of Computer Engineering, Hanshin University, 137 Yangsan-dong, Osan,
Gyeong-gi, Republic of Korea

Abstract: Android Apps. developed in Java language is vulnerable to repackaging attacks as it is easy to decompile an App. Therefore, obfuscation techniques can be used to make it difficult to analyzing the source of Android Apps. However, repackaging attacks are fundamentally impossible to block. Especially, it has been confirmed that most Android-based smart phones do not support verification process for the forged applications. Android is compiled into a class from a Java source and then compressed and stored as a Dex file to run in the Dalvik virtual machine. Then package the Dex file with xml+resource and distribute it as APK file. Therefore, if you add a module that maliciously acts after decompiling a Java class file in a normal APK file, you can create a Counterfeit App. In this study, we propose a process to repackage malicious Forged Apps. from normal APK files and propose a method to detect Forged Apps. Accordingly, the user installs and uses a Fake App. that appears to be functioning normally. In this case, the user is easily exposed to attacks such as leakage of personal information. Therefore, in this study, we have constructed Mobile Apps. identification system that applies the signature self-verification server monitoring method for Android Apps. and proposed a method of judging Android mobile Forgery Apps. by performing the verification process.

Key words: Android, mobile Forged Apps., repackaging, signature self-verification, detection, process

INTRODUCTION

Recently, as the number of smartphone users increases rapidly, the need for security enhancement techniques is increasing. Mobile Apps. based on the Android platform are being developed in the Java language and Malicious and Counterfeit Apps. are increasing due to the de-compilation weakness. Based on these vulnerabilities, anyone can create a Fake App. by adding source code that performs malicious actions. Therefore, there is a problem that the Forged mobile App. is easily distributed and installed through open market such that the malicious code with high security risk is hidden inside the Android application.

Recently found Forged mobile Apps. use a more advanced attack mechanism than traditional attacks, so, it can gain malicious privileges on Android devices, steal Google authentication tokens and gain access to Gmail, Google Play and other key services. In addition, once the Gooligan (Anonymous, 2018j) Malicious App. is installed, malicious code hidden inside the App. is activated based on modified FakeInst (Anonymous, 2018l), Vdloader (Anonymous, 2018m) and Trojan horses cooperated with C&C server. Therefore, privacy data and financial information in the mobile device are leaked to the outside attacker (Lee and Lee, 2015a, b; Ham and Lee, 2014).

Therefore, we analyzed the internal structure and operation mechanism of Forged mobile Apps. and provides a function to detect those counterfeit Mobile App. As a solution, we overviewed the signature generation and verification process of the application installation and developed a module that can be applied in real time for the detection of Fake Apps. Using proposed mechanism, it is expected to provide reliable information protection service and secure mobile usage environment in Android environment.

Android APK file structure: Android App. is written in Java. Java source is compiled as class files (class), converted into Dalvik Executable using 'Dx tool'. Android App. source executes on Dalvik Virtual Machine (DVM), a register based virtual machine for embedded devices, un-like Java Virtual Machine (JVM), a stack based virtual machine. Executable code of the App. is inside Dalvik executable (Dex) file. Androidmanifest.xmL has important information like package name, permissions, activities, services, receivers and content providers. Icons,

Corresponding Author: Hyung-Woo Lee, Division of Computer Engineering, Hanshin University,
137 Yangsan-dong, Osan, Gyeong-gi, Republic of Korea

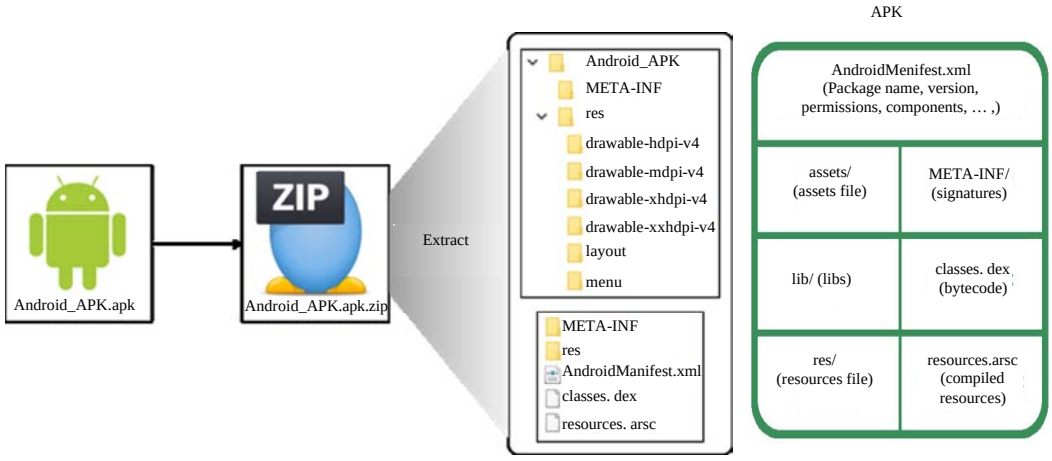

Fig. 1: APK file structure

shortcuts, images, string constants, dimension constants, etc. which are made available through resources. Classes. Dex file contains executable code. META.INF folder has developer certificate information. This content is compiled into a single package Android PacKage (APK). Once the App. is developed, developer self-signs it with his/her own private key and publishes it on official or 3rd party Android market (Enck *et al*., 2009; Zaidi *et al*., 2016).

In order to avoid source code theft or repackaging Android studio (Anonymous, 2018c) basically provides an obfuscation tool. Changing the variable name or class name to a meaningless name changes the program logic, making it difficult for the attacker to analyze and interpret the source code. However, there is a problem that repackaging can't be completely prevented. Instead of getting the complete original source code it does not prevent you from creating a Forgery App. by adding sources to your existing operating logic and repackaging it again. In the end, like the normal App., the basic behavior can provide similar malicious behavior while providing similar behavior. Test results showed that most Apps. were easy to create Malicious Apps. and only a few Apps. included the ability to verify and block Forged Apps. themselves. But this is not entirely impossible to create a Forgery App. Some Apps. can be judged to be safe because they perform the verification process on their own but if they include dependencies on other Apps. if the App. is vulnerable to forgery attacks, the problem arises that a Counterfeit App. can be created. Most malicious applications run malware as soon as they run. Even if the execution of the malicious application is terminated it is likely that the malicious code has already been installed (Fig. 1).

MATERIALS AND MEHTODS

Repackaging for Forged mobile Apps.

Mobile Apps. production method: The process of creating Counterfeit Apps. on mobile terminals can be divided into Native App., Mobile Web App. and hybrid web. Native Apps. are a way for users to download binary executables directly to their mobile devices and store and install them on the terminal. To create a Native App., the developer creates the source code, creates resources such as image and audio, compiles it using the SDK for the mobile OS and creates a binary executable file. Therefore, there is a disadvantage that the native application production methods are different from each other according to the mobile OS and the development period and cost are relatively high. But for Native App. creation, it is the most commonly used method for most Counterfeit App. development because it has the advantage of using both low-level API and high-level API provided by mobile OS.

The mobile web App. production method is to develop a mobile web service based on HTML5 using a rendering engine called WebKit (Anonymous, 2018a) which is included in the browser of the mobile terminal. Because it is developed as a standard web language compatible with WebKit it supports multi-platform by default. However, as described above, unlike the Native Apps. that run independently with the mobile OS, the mobile web App. runs only within the mobile web browser installed in the terminal, so, the scope and function of the API that the mobile web App. can access there is a limit. That is, the Native App. has access to the API provided by the user's mobile device but in the case of the mobile web App., there is a restriction that the Native App. operates in a limited manner within the browser.

The hybrid production approach mixes Native App. development with mobile web App. development. The web part is a way to reside on the server and integrated into the application code and stored in the local device. Because it uses HTML code hosted on the server, you can update your App. regardless of the approval, distribution, security regulations, etc., defined in the App. Store. However, this method has a disadvantage in that

when the local device is not connected to the network, the web contents can't be connected to a specific server and thus can't be used.

Production of mobile Forged Apps.: Most malware developers or malicious attackers analyze the behavior of the target App. Generally, a Fake App. production target sets an App. that operates in a Native App. mode or a hybrid mode. When a hybrid application is created for a Fake App. it has the advantage of being able to use various functions provided by a mobile device through a native method while sharing and spreading it on a multi-platform through a web interface. In addition, the server hosting method makes it easier to distribute Fake Apps. containing malicious code to mobile terminals. If a Fake App. is produced through the Native App. method, since both the low-level API and the high-level API provided by the user terminal can be used, malicious code such as obtaining information in the user terminal and leaking it to the outside is inserted. Therefore, it is a widely used method for making Counterfeit Apps. (Faruki *et al*., 2015; Rahman *et al*., 2016; Verma *et al*., 2016).

In this study, we made Fake Apps. in a native way. We implemented a spyware function that includes the same interface and functions as the existing normal App. but also extracts important information stored in the terminal without knowing it. In addition, a function to operate in the background was added to transmit the mobile terminal user information to a specific C&C server specified by the attacker and developed a function of secretly transmitting the information to the SMS number designated by the attacker.

Mobile Forged Apps. production process: Tools such as Signapk.jar (Anonymous, 2018k), Apktool (Anonymous, 2018b), dex2jar (Anonymous, 2016) and JD-GUI (Anonymous, 2018e) were used to build mobile Forged Apps. Signapk.jar was used to generate the signature for the APK file based on the '.pem' certificate. We then used Apktool to decompile and repackage the APK file. We extracted the AndroidManifest.xml and smail folder through the decompile process and made a Fake App. by modifying/modifying it. Dex2jar is used to analyze APK file. It extracts Dex file from APK file to be analyzed and converts it into jar format to extract original Java source code. We also used JD-GUI to analyze the jar format file. Specifically, the process of creating a mobile Counterfeit App. is as follows.

Mobile Forged App. creation procedure:

- Create a Malicious App. that redirects to a specific number when receiving an SMS message
- Use APKtool to disassemble the App. created in #1 to extract the files that perform core functions
- Select a normal App. to be used for creating Counterfeit Apps. and download them as an APK file
- Repackage the file by combining the extracted files from #2 in the normal App. APK file
- Perform the signature process for the created Counterfeit App. distribute it and perform the test process

Create Malicious Apps. that redirect SMS messages to a specific number: In the first step of the Forged App. production process shown above, we created an App. that uses the receiver function to transfer characters to another number when the user receives the text. A receiver is an Android component that receives status from a terminal while continuously checking for events. Usually you use a broadcast receiver to share data between applications. To read and transfer characters on Android, you need to include the necessary permissions in AndoirdManifest.xml. Therefore, we added the necessary authorization and the code to activate the receiver as shown in Fig. 2. When the next message is received, the on receive method is activated to redefine the Android receiver to read the message and send it back to a specific number.

Repackage the Malicious App's. core code into a Fake App.: We use APKtool to perform the decompile process for Malicious Apps. that provide the SMS text transmission function. As a result, the smail folder is created and the My Broadcast Receiver.smail file is copied. Now, we will use APKtool again to disassemble the normal App. and modify the AndroidManifest.xml file to add permissions and activate the receiver. The modification process is the same as the previous Malicious App. creation process. Then, insert the my broadcast receiver smail file extracted from the malicious application into the smail folder and then perform the repackaging process using APKtool. Finally, if we perform the signing process to distribute the APK, the creation of the Forged App. is completed. After completing the fabrication of the Forged App. if you check the internal structure of the App. Forged with JD-GUI, you can see that the source code for malicious action is added to the normal App. as shown in Fig. 3.

Detection of Forged Mobile Apps.

Mobile Forged Apps. detection and verification process: Android is similar to signing a jar file. In the META-INF folder, there are three files: MANIFEST.MF, CERT.SF and CERT.RSA. Because APK is a kind of compressed file, it generates a signature by extracting the hash value of all

```
<uses-permission android:name="android.permission.READ_SMS"/>
<uses-permission android:name="android.permission.INTERNET"/>
<uses-permission android:name="android.permission.RECEIVE_SMS"/>
<uses-permission android:name="android.permission.SEND_SMS"/>
<uses-permission android:name="android.permission.READ_PHONE_STATE"/>

<receiver android:name="MyBroadcastReceiver">
 <intent-filter>
  <action android:name="android.provider.Telephony.SMS_RECEIVED"/>
 </intent-filter>
</receiver>
```

```
@Override
public void onReceive(Context context, Intent intent) {
    Bundle bundle = intent.getExtras();
    String str = "";
    if (bundle != null) { //receiver start
        Object [] pdus = (Object[])bundle.get("pdus");
        SmsMessage [] msgs
                = new SmsMessage[pdus.length];
        for (int i = 0; i < msgs.length; i++) {      //msg call
            msgs[i] = SmsMessage
                    .createFromPdu((byte[]) pdus[i]);
            str += msgs[i].getOriginatingAddress()
                    + "spyware : " +
                    msgs[i].getMessageBody().toString()
                    +"\n";
        }
        smsMsgSent("010xxxxxxxx", str); //send message

    }
}
```

Fig. 2: Addition of activation code after setting Android permissions and its receivers

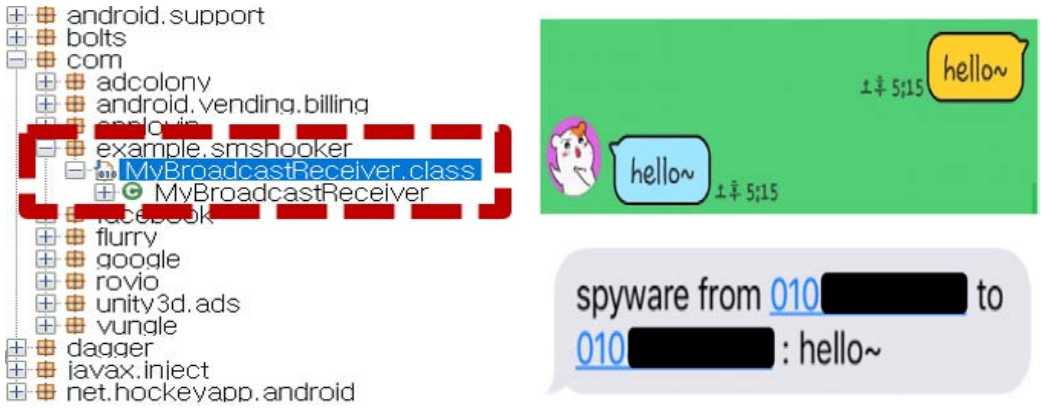

Fig. 3: Fake App. production result with malicious receiver code and its operation test

the files inside the App. Therefore, it is possible to check whether the internal file of the App. is forged or not through the generated hash value. The generated hash value is digitally signed with the private key and distributed with the public key, so that anyone can check the integrity of the App., thereby preventing it from being updated with the Forgery App. Verification is done during installation and the Android OS generates the hash value again and compares it with the existing signed file. Therefore, the Android platform provides a function to prevent normal Apps. from being updated with Malicious Apps. However, it does not provide a function for determining Fake Apps. through the creation and verification of Android signatures (Fig. 4).

Android application will not be installed on the terminal if a signature has not been generated or if the signature verification process fails. After generating SHA-1 based hash values for all files in the APK file, the base64 encoding process is performed. The generated hash value is created with the developer's private key and distributed with the public key certificate as shown in the Fig. 3. You can also verify your signature on your App. through a verification process. CERT.RSA contains a public key and a value encrypted with CERT.SF as a private key. The signature value signed with the private key with the public key is decrypted and compared with the CERT.SF file to perform verification. Only the APP. developer can generate legitimate signature values in APP. This function provides a function to prevent the abnormal update of the already installed Apps. (Fig. 5).

Mobile Apps. signature self-verification method: We use antivirus program to enhance security of mobile device. And security code is included mobile application for security enhancement. Therefore, this kind of mechanism makes it easy to control and manage mobile device safely by performing an application verification processes. As a more advanced solution an application can send signature value of its own applications to the specified verification server to check whether it is forged or not. A factor that interferes with the verification process is the disconnection of the network. In order to prevent the internal logic from changing to the FAKE server instead

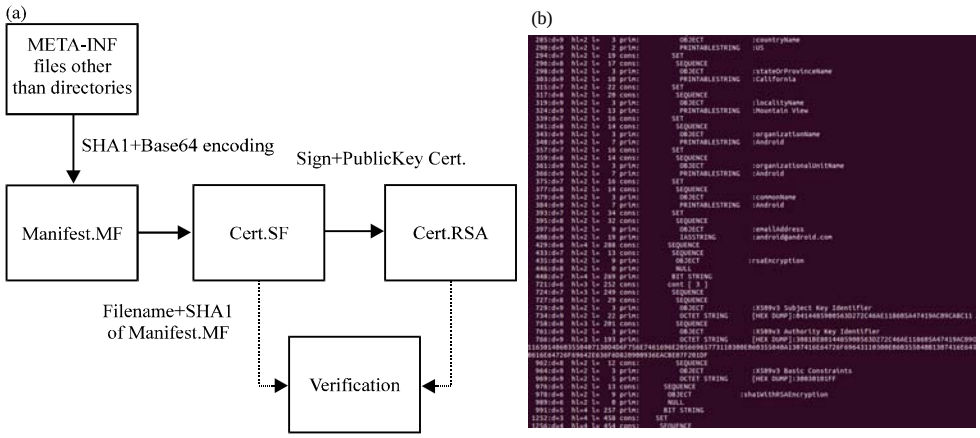

Fig. 4: a, b) Android signature generation and verification process

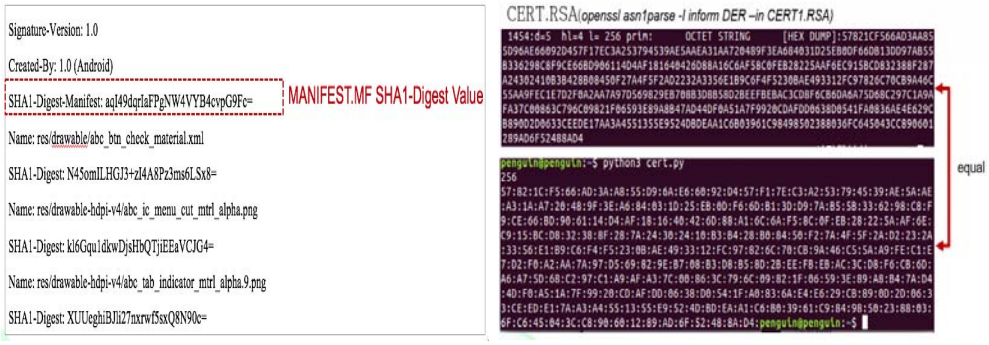

Fig. 5: Android signature verification example

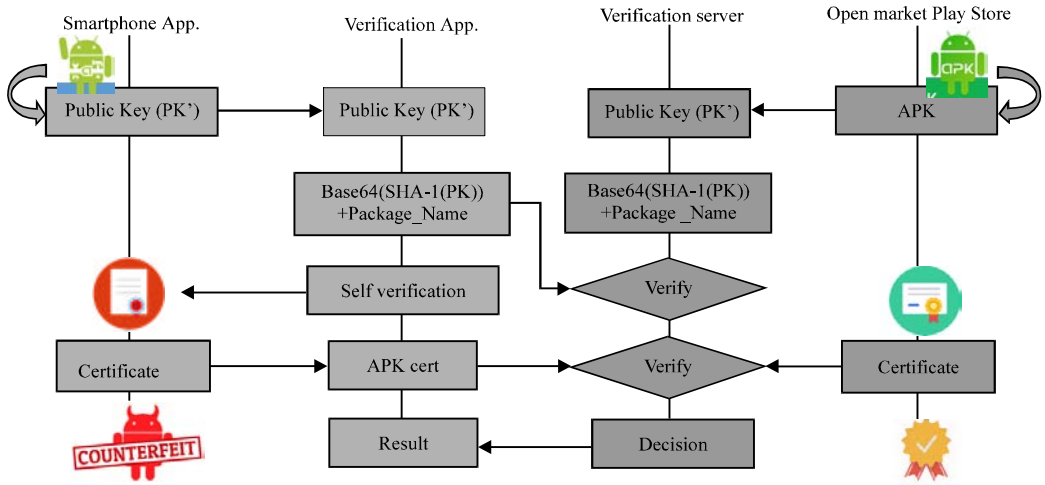

Fig. 6: Server based self-verification mechanism

of the verification server, the verification server remembers whether or not each verification application is installed and the last verification time (Fig. 6).

Android Apps. developed in Java language is vulnerable to repackaging attacks as it is easy to decompile an App. Therefore, obfuscation techniques can be used to make it difficult to analyzing the source of Android Apps. However, repackaging attacks are fundamentally impossible to block (Enck *et al*., 2009; Zaidi *et al*., 2016; Rahman *et al*., 2016). Especially, it has

been confirmed that most Android-based smart phones do not support verification process for the forged applications. Android is compiled into a class from a java source and then compressed and stored as a Dex file to run in the dalvik virtual machine. Then package the Dex file with xml+resource and distribute it as APK file (Shabtai *et al*., 2012; Lee and Lee, 2018). Therefore, if we add a module that maliciously acts after decompiling a Java class file in a normal APK file, we can create a Counterfeit App.

As described previously, we propose a process to repackage malicious Forged Apps. from normal APK files and propose a method to detect Forged Apps. Accordingly, the user installs and uses a Fake App. that appears to be functioning normally. In this case, the user is easily exposed to attacks such as leakage of personal information. Therefore, we have constructed Mobile Apps. identification system that applies the server-side signature self-verification system for Android Apps. and proposed a method of judging Android mobile Forgery Apps. by performing the verification process.

RESULTS AND DISCUSSION

Implementation and experiments

Design and implementation of mobile Forged Apps. detection system: We used Android studio to create an MFI App. that will be installed on a monitoring terminal to detect mobile Forged Apps. In addition, a validation server that provides Forged App. detection was built using node.js and MySQL. When the verification App. starts, self-sign service runs in the background. The service internally verifies its own signature value at regular intervals and sends the value to the verification server. You can also invoke the list of installed packages to perform individual verification or verify all at once. The verification result will return one of three things: trust, risk or warning. When the signature value of the application exists in the server and the result of the comparison is the same, the danger is similarly when the signature value of the application exists but the comparison result is inconsistent, the warning indicates that the normal signature value of the application is not present in the server. When the Fake App. detection method proposed in this study is applied it is confirmed that the detection function for the Fake App. is provided as shown in the following (Fig. 7).

Performance measurement of mobile Forged Apps. detection: In order to perform a detection process for a Fake App., a process of producing a direct Fake App. was performed. Generally, we made four types of Apps. (Angry bird (Anonymous, 2018d), Shinhan Bank Apps. (Anonymous, 2018o), Ahnlab V3 Mobile Security (Anonymous, 2018n) and Kakao Talk (Anonymous, 2018f) which are widely distributed through the Google Play Store. Inside the Forged App., we added a malicious code module to send and receive Fake SMS. Fake Apps. are implemented to operate with the same interface as existing normal Apps. And a system that provides verification Apps. and a server-based signature self-verification function to detect such Fake Apps. As a result of

Fig. 7: Implementation of mobile forgery identification system and its experiment result

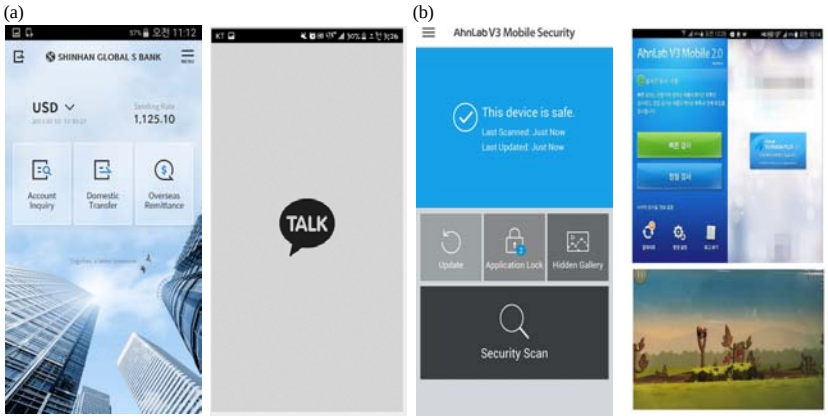

Fig. 8: a) Fake Apps. with errors and b) Fake Apps. that works properly

measuring whether or not to detect Forged Apps. it was found that conventional mobile vaccines did not detect modified Fake Apps. However, if we use Fake App. detection method with server-side self-signature verification developed in this study, we can provide identification and verification process for certification values contained in suspicious Mobile App. As a result, we were able to provide the ability to identify normal and Counterfeit Apps. efficiently.

When we checked the test results, we were able to confirm that the receiver was functioning properly in our game and Vaccine Apps. and we were able to confirm that the malicious action was performed. However, in case of Shinhan Bank Apps. and chat applications (Kakao Talk), Malicious codes could not be injected because the Counterfeit filtering function is built in the normal App. Especially, when it is a Financial App., it is set to be linked with a Vaccine App. to improve security. However, after adding malicious code to the vaccine App. to create a Counterfeit App., we can confirm that it is possible to create a Counterfeit App. for a Financial App. by inserting/modifying malicious code to work with a Financial App. Also, in case of mobile anti-virus application, we were able to build a Counterfeit App. as shown in Fig. 8 and we were confident that it would provide detection function for malicious mobile vaccines and game Counterfeit Apps.

We tested whether the existing mobile vaccine detected the Fake App. developed in this study. Testing with mobile antivirus such as Malwarebytes for Android (Anonymous, 2018h), McAfee Mobile Security (Anonymous, 2018i), Kaspersky Mobile Antivirus (Anonymous, 2018g) and V3 Mobile Security (Anonymous, 2018n) showed that all of the existing antivirus programs did not detect Fake Apps. Failure to detect Fake Apps. in a typical mobile antivirus program is a serious problem. As mentioned earlier, we can easily create a malicious Counterfeit App. that provides a similar interface to a normal App. Therefore, it is necessary to study a technique that can continuously improve detection performance for malicious counterfeit Apps.

CONCLUSION

Currently, most mobile Apps. do not include monitoring and self-verification of forgery. However, only specific Apps. used in the banking industry are verifying their own counterfeiting. In addition, the mobile vaccine App. which is required to filter malicious activity is not able to detect malicious Forgery App. and can't detect and check whether the malicious application is counterfeited. Therefore, in order to prevent this, we designed and implemented a Fake App. detection mechanism that is more improved than the existing method by integrating and managing forgery by performing a self-verification method for Fake App. signatures. In detail, the client/server-based signature value self-verification method for the analysis target can provide a reliable verification process for the forgery-proofing application and can reduce the possibility of misuse detection, so that, it can be applied to all mobile applications. Therefore, this will provide safe Mobile Apps. usage environment.

ACKNOWLEDGEMENT

This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science and Technology (NRF-2017R1D1B03035040). And This article is a revised and expanded version of a study (Lee and Lee, 2018) entitled "Android Mobile Forged Apps. Detection with Server-Side Signature Verification", presented at Advanced and Applied Convergence, the 4th

International Joint Conference on Convergence (IJCC2018) held on January 31-February 7, 2018, Hawaii, USA.

REFERENCES

Anonymous, 2016. Dex2jar: Tools to work with android.dex and java.class files brought to you by: pxb1988. Slashdot Media and Dice Inc., Redwood City, California. https://sourceforge.net/projects/dex2jar/.

Anonymous, 2018a. A fast, open source web browser engine. WebKit. https://webkit.org/

Anonymous, 2018b. A tool for reverse engineering Android APK files. GitHub Inc., San Francisco, California, USA. https://github.com/iBotPeaches/Apktool.

Anonymous, 2018c. Android studio the official IDE for Android. Android. https://developer.android.com/studio/index.html.

Anonymous, 2018d. Angry birds classic. Rovio Entertainment, Espoo, Finland. https://play.google.com/store/apps/details?id=com.ahnlab.v3mobilesecurity.soda.

Anonymous, 2018e. JD project. Java Decompiler. http://jd.benow.ca/

Anonymous, 2018f. Kakao talk: Free calls and text. Kakao, Jeju City, South Korea. https://play.google.com/store/apps/details?id=com.kakao.talk.

Anonymous, 2018g. Kaspersky mobile antivirus: AppLock & web security. Kaspersky Lab, Moscow, Russia. https://play.google.com/store/apps/details?id=com.kms.free.

Anonymous, 2018h. Malwarebytes for Android advanced protection against malware, ransomware and other growing threats to Android devices. Malwarebytes, Santa Clara, California, USA.

Anonymous, 2018i. McAfee mobile security and lock. McAfee, Santa Clara, California, USA. https://play.google.com/store/apps/details?id=com.wsandroid.suite.

Anonymous, 2018j. More than 1 million google accounts breached by gooligan. Check Point Software Technologies, Tel Aviv, Israel. https://blog.checkpoint.com/2016/11/30/1-million-google-accounts-breached-gooligan/.

Anonymous, 2018k. Sign APK is used to sign the APK file after repack: The easiest way ever. GitHub Inc., San Francisco, California, USA. https://github.com/techexpertize/SignApk.

Anonymous, 2018l. Trojan: Android/fakeinst threat description. F-Secure, Helsinki, Finland. https://www.f-secure.com/v-descs/trojan_android_fakeinst.shtml.

Anonymous, 2018m. Trojan: Android/vdloader a threat description. F-Secure, Helsinki, Finland. https://www.f-secure.com/v-descs/trojan_android_vdloader.shtml.

Anonymous, 2018n. V3 mobile security-anti malware/booster/apps lock. AhnLab, Inc., Gyeonggi Province, South Korea. https://play.google.com/store/apps/details?id=com.ahnlab.v3mobilesecurity.soda.

Anonymous, 2018o. [ShinhanSol (SOL)-Shinhan bank smartphone banking]. Shinhan Bank, Seoul, South Korea. (In Korean)

Enck, W., M. Ongtang and P. McDaniel, 2009. Understanding android security. IEEE Security Privacy, 7: 50-57.

Faruki, P., V. Laxmi, A. Bharmal, M.S. Gaur and V. Ganmoor, 2015. AndroSimilar: Robust signature for detecting variants of Android malware. J. Inf. Secur. Appl., 22: 66-80.

Ham, Y.J. and H.W. Lee, 2014. Malicious Trojan horse application discrimination mechanism using realtime event similarity on android mobile devices. J. Internet Comput. Serv., 15: 31-43.

Lee, H.S. and H.W. Lee, 2015b. Fake C&C server for evidence aggregation and detection of server-side polymorphic mobile malware on android platform. Intl. Inf. Inst., 18: 3723-3737.

Lee, H.S. and H.W. Lee, 2015a. Implementation of polymorphic malware DB based dynamic analysis system for android Mobile Applications. Intl. Inf. Inst., 18: 3187-3197.

Lee, J. and H.W. Lee, 2018. Android mobile forged apps detection with server-side signature verification. Proceedings of the 4th International Joint Conference on Convergence (IJCC'18), January 31-February 7, 2018, Sheraton Waikiki Hotel, Honolulu, Hawaii, USA., pp: 59-60.

Rahman, M., M. Rahman, B. Carbunar and D.H. Chau, 2016. Fairplay: Fraud and Malware Detection in Google Play. In: Proceedings of the 2016 SIAM International Conference on Data Mining, Venkatasubramanian, S.C. and W. Meira (Eds.). Society for Industrial and Applied Mathematics, Philadelphia, Pennsylvania, USA., ISBN:978-1-61197-434-8, pp: 99-107.

Shabtai, A., U. Kanonov, Y. Elovici, C. Glezer and Y. Weiss, 2012. Andromaly: A behavioral malware detection framework for Android devices. J. Intell. Inform. Syst., 38: 161-190.

Verma, S., S.K. Muttoo and S.K. Pal, 2016. MDroid: Android based malware detection using MCM classifier. Intl. J. Eng. Appl. Sci. Technol., 1: 206-215.

Zaidi, S.F.A., M.A. Shah, M. Kamran, Q. Javaid and S. Zhang, 2016. A survey on security for smartphone device. Intl. J. Adv. Comput. Sci. Appl., 7: 206-219.