A Strengthened Android Signature Management Method

Taenam Cho[1] and Seung-Hyun Seo[2]
[1] Dept. of Information Security, Woosuk University
Jeonbuk, South Korea
[e-mail: tncho@ws.ac.kr]
[2] Dept. of Mathematics, Korea University, Sejong Campus
2511 Sejong-ro, Sejong City 339-770, Korea
[e-mail: crypto77@korea.ac.kr]
*Corresponding author: Seung-Hyun Seo

Received January 10, 2014; revised June 16, 2014; revised August 11, 2014; accepted January 5, 2015; published March 31, 2015

Abstract

Android is the world's most utilized smartphone OS which consequently, also makes it an attractive target for attackers. The most representative method of hacking used against Android apps is known as repackaging. This attack method requires extensive knowledge about reverse engineering in order to modify and insert malicious codes into the original app. However, there exists an easier way which circumvents the limiting obstacle of the reverse engineering. We have discovered a method of exploiting the Android code-signing process in order to mount a malware as an example. We also propose a countermeasure to prevent this attack. In addition, as a proof-of-concept, we tested a malicious code based on our attack technique on a sample app and improved the java libraries related to code-signing/verification reflecting our countermeasure.

Keywords: Android, Code-signing, Security, Malware

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (No. 2012R1A1A3015030). The authors would like to thank Kyongyong Park for implementing app.

http://dx.doi.org/10.3837/tiis.2015.03.021 **ISSN : 1976-7277**

1. Introduction

The smartphone has become a necessity in today's modern world. iOS, Android, RIM, MS, Symbian, Blackberry, and Bada are the most prominent and are representative of the largest market shares. In a report by Gartner, in 2010 among smartphone operating systems, Symbian had the largest market share with 41.2 percent [1] but by the first quarter of 2013, Android displaced Symbian for the number one spot and commanded a 74.4 percent market presence [2]. Due to the rapidly increasing adoption of smartphones, the spread and incidence of mobile malwares has also skyrocketed. Therefore smartphone security and privacy has become a major concern. As a result, each respective smartphone OS developer has formulated their own independent security policy resulting in varying degrees of requirements and standards. Android, being the most popular OS of choice, is incredibly open-source allowing for innovation but this also leaves it vulnerable and an attractive target for attackers.

As each operating system has developed, the methods used by for attackers have increased in frequency and creativity. In order to protect the system from attackers, multiple defense mechanisms have been conceived. One of the ways that have been developed is to apply a method called code-signing. Developers who utilize code signing have formulated their own cryptographic technology for use in these distinct digital signatures. Apps which do not carry the appropriate signature are prevented from installing on a phone. The private keys used in code signing are kept and signed by the developer, proving its authenticity. A public key, which is also sent by the developer, can be used by a user or the market to verify the authenticity of the signature. Thus, the digital signature of the developer can be verified as well as ensure that the app has not been tampered or falsified.

Malwares specific for the Android OS have used a variety of techniques to compromise security. Of those techniques, the most popular and widely used method is the repackaging. To repackage an app, the attacker removes the app's signature, inserts the malicious code, and then generates a new signature in place of the original. This 'repackaged' app is then distributed with the modified signature. In order to detect the inserted malicious code, the app must be disassembled and undergo a complex process of side-by-side comparison with the original app source code. Particularly, since Android uses a code-signing method that utilizes self-signed certificates of developers without the verification of certification authority, repackaging types of malware are easily distributed.

Android adopts SHA1, RSA and DSA algorithms for signature. Even though these algorithms are secure, there are vulnerabilities due to the mangement scheme of signature generation and verification. When the app has two or more signatures and one of them is modified, the verification process cannot recognize the modification. This problem is related not only to signature verification process but also to signature generation process. In this paper, we first analyze the critical vulnerabilties of Android signature management scheme and show the new attack technique by utilizing these vulnerabilties. Our proposed attack is a method where source code modification can be avoided. Thus using our attack technique, those without intimate knowledge of hacking techniques can easily cause a large ripple effect of widespread damage with a malware. Using this new type of attack, the malicious code is inserted in the folder where the code signature files are stored. This method is not commonly detected by the average users who are unfamiliar with the structure of the Android apps. We also proposed a solution to this type of attack through a series of experiments which demonstrate the feasibility of the proposed countermeasures.

In the next section, we discuss the most common types of smartphone malware. In Section 3, we discuss the related research. Section 4 gives an overview Android signature mangement

scheme and its vulnerabilities. Section 5 shows how through our implementation techniques, malicious apps manage to exploit these vulnerabilities. Section 6 covers countermeasures against these vulnerabilties and its implementation results are described. In the last section, we provide conclusions and our research results.

2. Types of Mobile Malware

In this section, we present the types of mobile malware according to their current classification as provided [3] .

2.1 Repackaging

Repackaging is the most commonly used technique to deceive smartphone users into installing mobile malware. In order to repackage apps, attackers download legitimate popular apps, disassemble them, embed malicious code, and then reassemble and upload the modified apps to the official Android app market or unregulated app markets. Unwary users download and install these repackaged apps, because the malwares are disguised within seemly legitimate apps. The repackaged apps not only feature the same functionality as the original apps, but also include malicious codes which collect sensitive information and/or obtain monetary profit. 86% of Android malware are repackaged versions of legitimate applications [4]. There are a lot of mobile malwares which use the repackaging technique, such as AnserverBot [5], ADRD [6], Pjapps [7] and etc.

2.2 Update Attack

This update attack technique may still repackage popular apps, but instead of embedding the malicious payload as a whole, it only contains an update component to fetch or download the additional malware during runtime. This technique makes detection difficult. Once the user accepts an updated version with the malicious code, the malware is installed. Recent malware such as BaseBridge [8] and DroidKungFuUpdate [9] adopt this attack technique. Once the BaseBridge malware is installed and run, it checks whether an update dialogue needs to be displayed. If the user selects 'yes' and accepts the updated version, it subsequently embeds the malicious code and the user's smartphone is infected. This updated version is not the expected original app, but a malware. The DroidKungFuUpdate malware is similar with BaseBridge, but it does not enclose the 'updated' version inside the original app. Instead of carrying it, it remotely downloads a newer version from the network.

2.3 Drive-by-Download

This technique is similar to that of the traditional drive-by-download attack. Even though it is not directly exploiting mobile browser vulnerabilities, it is essentially enticing users to download attractive apps. For example, it utilizes genuine looking advertisements that link back to fraudulent websites that can download malware onto users' smartphones. Jifake [10] and Zitmo [11] are mobile malware that utilize the drive-by-download technique. The Jifake malware is downloaded when users are redirected to the malicious website. It uses a malicious QR code, which will redirect the user to another URL containing the Jifake malware when scanned. It is the repackaged mobile ICQ client which sends several SMS messages to a premium-rate number.

3. Related Works for Android Security Mechanisms

Android has become the most popular mobile platform. As a result, many researchers have studied the security mechanisms for the Android OS platform. In order to improve the security of Android smartphones, several OS platform-level extension techniques such as TISSA [12] and AppFence [13] have been proposed. These works improve the Android OS framework to support fine grained controls of system resources accessed by risky third-party apps. AppFence modifies Android OS to protect private data from being leaked by providing and imposing fine-grained privacy controls on existing apps. TISSA proposes a privacy mode in Android platform which provides fine-grained control over user privacy. Kirin framework [14] provides a lightweight certification of Android apps to block the installation of suspicious apps by examining the existence of certain dangerous permission combinations. Saint [15] protects the exposed interfaces of apps to others by allowing the app developers to define related security policies for future runtime enforcement. That is, Saint provides install-time permission granting policies as well as runtime inter-application communication policies for improved Android security. L4Android [16] runs multiple OS platforms on a single smartphone for isolation and security. AdDroid [17] and AdSplit [18] are approaches to separate the privileges between the ad library and its host app to remove the permission requests by the host app on behalf of its ad library. However, these works are unable to improve the weakness of the code signature management scheme of Android.

Since the Android code signing method does not provide security for the app, it has been easily exploited to insert malicious code. In order to install Android apps on a device, the developer's code signature files must be included. Recently Cho suggested that in the current situation, when malware is distributed, the burden of responsibility can be shifted [19-20]. Neither the developer nor the Market is willing to take responsibility. In order to avoid this unproductive situation, Cho proposed a dual-signature model where both the developer and the Market independently sign the app to create a final verification seal and provide authenticity. Public-certified certificates are preferable as opposed to self-certified certificates. This can avoid the deadlock between the two parties since both independently sign the app and any modification can be traced back. However, the proposed double signature technique is simply an improvement between the two signatures which can assist in the detection of modification. It does not address the key vulnerability inherent in the Android signature management scheme.

4. Android Code Signature Management Scheme

4.1 Android Applications

4.1.1 Android Platform

The Android platform is built on Linux Kernel and the Java library. In addition, the inherent library and API stack become the basis for the Android framework [21]. Therefore, the structure and security concept of Android applications are based on Linux and Java. The Android app’s execution environment is called Dalvik virtual machine.

4.1.2 Android Application Structure

The fully developed version of an application is published as the release version. The app’s Java file is compressed in a jar file with an .apk extension. The basic file structure of an app is shown in **Fig. 1**. classes.dex is the compiled source file in which class files are included.

Resouce files for strings, images, user interface layouts are stored in the folder called *res*. The file resources.arsc contains binary information about resources such as the graphics, sounds, and dex (dalvik executable file). In addition to the various folders and files AndroidManifest.xml is contained in the root folder. The manifest presents essential information about the application to the Android system. The system must have the information before the app runs any code of it. For example, it contains the Java package name for the application, components of the application such as activities, services and content providers, permissions that the application must have and the list of the libraries that the application must be linked against [22] .

4.1.3 Directory Structure of Android Devices

The .apk files downloaded through the Android market are stored in /data/app. The data used by the app are saved under the designation: /data/data/*package_name*/. *package_name* is used as a identifier to distinguish between different apps in the device and it is defined in the AndroidManifest.xml. For example, an app with the package name com.android.sample uses /data/data/com.android.sample/database folder to store the database files. The /data/data/com.android.sample/files folder is used to store the app files.

Depending on the version of Android used, folders may vary, in this paper we describe them for Android 2.3.3 (Gingerbread) version (all versions are similar). In this version, /data/app folder and /data/data folder are system folders and are only accessible through rooting. In addition, each app's data are managed in the sandbox area which is the Java security mechanism. Thus, only the respective app or the administrator can access each folder. This compartmentalizes apps so they cannot access the data folders of other apps.

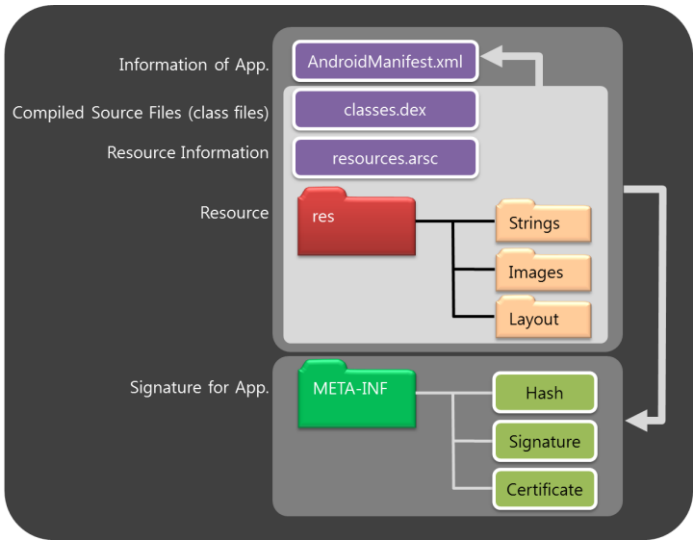

Fig. 1. File Structure of Android Applications

4.2 Code-Signing Procedure

4.2.1 Purpose of Code-Signing

The Android system requires that all installed applications be digitally signed with a certificate whose private key is held by the application's developer. Most smartphone platforms require code signing but each platform maintains different procedures with different purposes. For example, the iPhone requires one of the most demanding and strict compliances of policy while in the case of Android phones, a lenient, almost lax policy based on the concept of an open market is applied. This has both pros and cons for each ecosystem. The purposes of code-signing in the Android platform are as follows: [23] .

(1) Application upgrade - To update an application, the developer must continue to sign the updates with the same certificate or set of certificates. When the system installs an update to an application, it compares the certificates in the new version with those of the existing version.
(2) Application modularity - The Android system treats applications that are signed by the same certificate as a single application. Developers can deploy their applications in modules, and users can update each of the modules independently if needed.
(3) Code and data sharing through permissions - By signing multiple applications with the same certificates and using signature-based permission check, applications can share code and data in a secure manner.

4.2.2 Signing Procedure

Google, the provider of the Android platform Dalvik, supports APIs which can be plugged in for the Java development environments such as Eclipse. **Fig. 2** shows the relationship between the Eclipse [24] and the code-signing related tools. Android SDK (Software Development Kit) is based on JDK (Java Development Kit) [25] . ADT (Android Development Tools) to utilize Android SDK is available in Elipse [26] . JDK keytool [27] , jarsigner [28] and zipalign [29] are associated with a code signing tools. Developers can generate code signing keys manually using JDK or automatically using code-signing wizard of Eclipse. Depending on the method, the default algorithms and names for the keys are different. In this paper, the method will be optional, depending on convenience. Steps for signing are as follows:

(1) Developer using the keytool generates the public-key/private-key used for code signing. The key and certificate are stored in the repository known as keystore [27] . Both DSA [30] and RSA [31] algorithms are supported for signing and SHA1 [32] is supported as hash function. In the instance that Jarsigner is used, DSA is set as default while RSA is utilized when wizard is used. When a key is created, using a X.509 format, a self-signed certificate is generated. Each key is protected by passwords imposed on the key and/or the keystore.
(2) Using jarsigner, with the private key stored in the keystore, a developer signs the app in jar format. Personal keys can be identified using aliases. Suppose the key's alias is myKey and the used hash and signature algorithm are SHA1 and RSA. In the signed apk, META-INF directory is created; and signature files, MANIFEST.MF, MYKEY.SF, MAYKEY.RSA are generated and then stored in META-INF (when wizard is used, CERT.SF, and CERT.RSA are generated regardless of the alias). The relationship between these files and the structure are shown in **Fig. 3**.

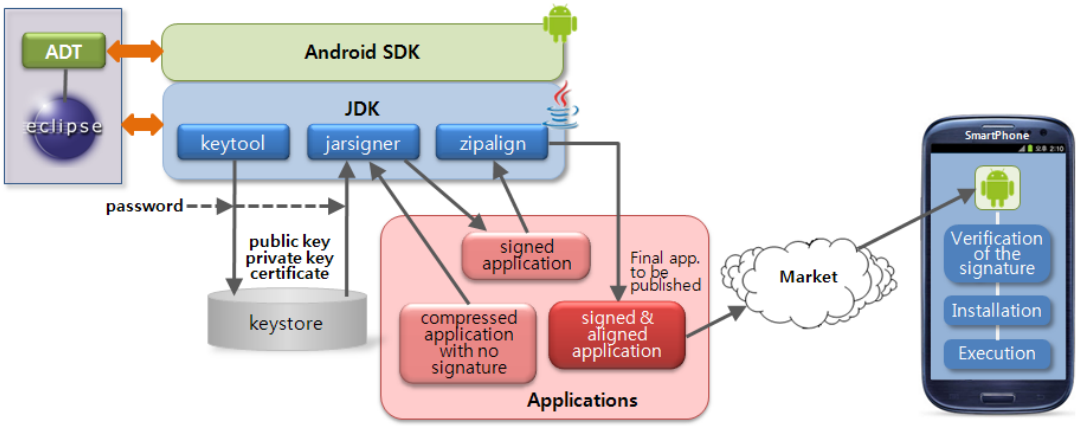

Fig. 2. Android Signature and Tools [20]

Once signed, zipalign is used for packing. This process is not integral towards signing and is intended for efficiency, thus can be skipped.

4.2.3 Verification Procedure for the Signature

When an app is installed on a device, its signature is verified. If it fails, the app cannot be installed (See **Fig. 2**). An in-depth view of the signature file structure is shown in **Fig. 3**. When signed using the Eclipse Wizard, the signature files in the META-INF folder are MANIFEST.MF, CERT.SF and CERT.RSA. Each file is organized as follows.

- MANIFEST.MF: contains the manifest version, signature provider and hash values. Each hash value is derived from each file in the apk and consists of three lines. First line is the path including the filename, and the second line contains the SHA1 value for file represented by the base code 64 [33] and the third line is an empty line.
- CERT.SF: contains the hash values derived from the hash values in the MANIFEST.MF. The version information and the signature provider are stored in the first two lines. The third line is the SHA1 hash value of MANIFEST.MF. The hash information for each file consists three lines. The first line includes the path and filename. The second line contains the SHA1 value for the corresponding three lines of a hash value in MANIFEST.MF. The third line is an empty line.
- CERT.RSA: contains the RSA signature on CERT.SF and the certificate. This signature is generated by the private key of myKey. The certificate contains the public key of myKey and follows the X.509 standard format [34] .

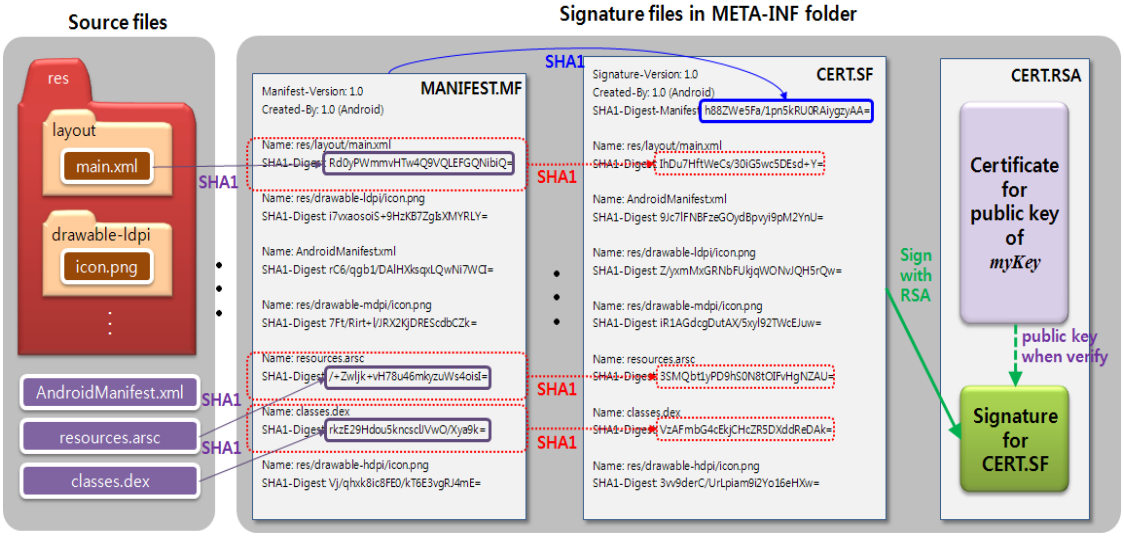

Fig. 3. Signature File Structure

These files, depending on the structure must meet all of the following conditions for the signature verification to be successful.

- MANIFEST.MF: contains the correct hash values of each file except signature files stored in the META-INF. Conversely, it contains hash values only for the files contained in the apk.
- CERT.SF: contains the correct hash value for MANIFEST.MF and the correct hash values for each hash values stored in MANIFEST.MF. Conversely, CERT.SF file contains hash values only for the hash values described in MANIFEST.MF.
- CERT.RSA: certificate issuer's signature has not expired. When verifying the signature with the public key contained in CERT.RSA, it is confirmed that the signature is intended for CERT.SF

4.2.4 Double-signing

When using Eclipse wizard, only one signature may exist for one apk because only unsigned files can be signed. However, when using jarsigner, the signatures can be duplicated because the names of the signature files have the same aliases of the key. Therefore, an apk can be signed 2 or more times with different keys. Because the structure of a multi-signed apk is the same as a double signed apk, for convenience, we will describe only the structure of double signing.

(1) Double signing

If we first generate an RSA signature using the key with the alias of FIRST and the next we generate the DSA signature using the key with the alias of SECOND, a structure as shown in **Fig. 4** can be created. In the case of the first signature, the signature method used is the same as previously stated. In the case of the second signature, it does not contain the first signature file as also shown in **Fig. 4**. MANIFEST.MF file is shared among the first and second signatures. The generations of the signatures are independent and so without checking the signature creation time; there is no way to verify the sequence of creation. Thus it is not possible to verify whether the first signature is a forgery through use of the second signature. In **Fig. 5**, we show this process. Two .SF files contain the same hash information while only differing in its

sequence. SECOND.SF does not contain the hash values of FIRST.SF and FIRST.RSA. Thus, the second signature is verified as valid even after the first signature is deleted.

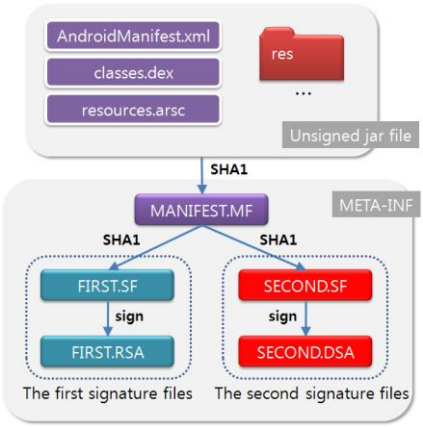

Fig. 4. The Structure of Double Signed Files

(2) Verification of double signature

We use the JDK (version 1.7.0_45) jarsigner to verify the double signed .apk file. jarsigner shows the validated results of each signature. We tested the scenario where one of two signatures is invalid. The experiments were run for the three cases as follows.

[Case 1] Expired signature was inserted

As shown in **Fig. 6**, a warning message appears that the certificate has expired. However, installation and execution of the app on a smartphone is still possible because the signature is accepted as valid.

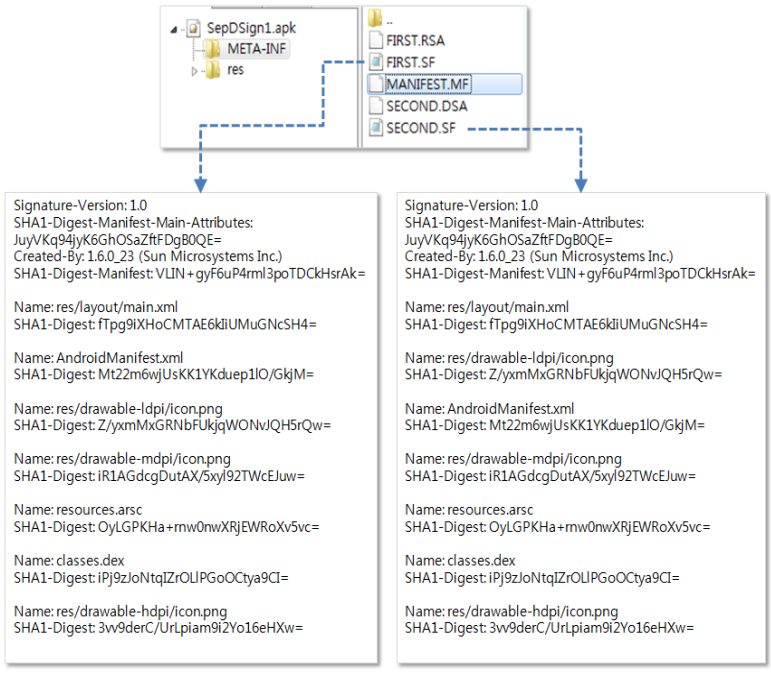

Fig. 5. The Contents of Two .SF files

Fig. 6. One of Certificates is Expired

[Case 2] Invalid signature file such as one for another app was inserted
In this case, a warning message is displayed, but the signature of app is considered as valid and it is possible to install and run the app on a smartphone (See **Fig. 7**).

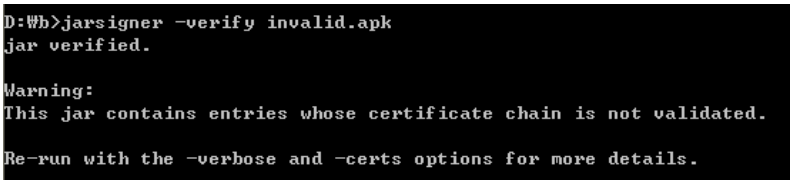

Fig. 7. One of Signatures is not Valid

[Case 3] Arbitrary file was inserted
Two image files (e.g., the image does not follow the X.509 certificate format) extensions were changed to .SF and .RSA and inserted. As shown in **Fig. 8**, the signature verification of the app is success with a warning message (Lower versions of JDK don't display even any warning message).
If verified by jarsigner, first and second warning messages are displayed as the output, although in all three cases, the app is successfully installed and executed on a smartphone because in all three cases, the signatures of app are accepted as valid.

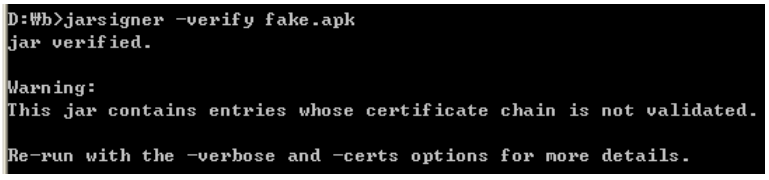

Fig. 8. One of Signatures is not a Signature File

4.2.5 Vulnerability Analysis

As shown in section 4.2.4 only one valid signature is necessary to install or execute the app. The third case shows a serious security vulnerability since an attacker can create two malware files whose extensions are .SF and .RSA respectively and insert them into the META-INF folder. Nevertheless, the signature of the app will be considered as valid. Unlike existing repackaging methods, this method does not require the attacker to resign to successfully insert the malware. The original app and the modified app use the same signature and certificate. Without a thorough analysis, the differences cannot be recognized. In the next section, we show how easy and how pervasive this method of attack can be.

5. Implementation of Malicious App using the Vulnerabilities

In this section, we will show an example of attack which utilizes the vulnerabilities described in Section 4. As this attack requires root privilege, the attack method itself is not epochal. A feature which differentiates this from other attacks is that the malwares can be distributed without alternation of target apps' codes and resigning of the target apps. The purpose of this section is to show the feasibility of the threat via implementation and using the vulnerabilities we have discovered.

5.1 Structure and Function of the App

A Trojan horse type of malicious codes is hidden in the other normal apps. However, to avoid detection, several apps interact maliciously as discussed in [35] . In this paper, we use a similar technique:

(1) We create a malicious app and change the extension of a malicious file into .RSA. This fake file is named HIDDEN.RSA. In order to install and execute HIDDEN.RSA, we created an execution file and the extension was modified to .SF which is named HIDDEN.SF.
(2) We choose an app with high popularity as a target. We insert the fake signature files, HIDDEN.RSA and HIDDEN.SF, into META-INF of the target app. Re-signing this altered app is not required. The only alteration necessary is insertion and compression.
(3) We create an app that triggers the fake signature files in META-INF folder. This is called trigger.apk.

trigger.apk is executed as the following procedure:

(1) Rooting to get root privilege to access another app's data.
(2) Checking .apk files stored in data/app/ on a device if they contain HIDDEN.SF and HIDDEN.RSA in their META-INF folder. Let the infected.apk be the infected app.
(3) Changing the permission of infected.apk in order to extract and execute HIDDEN.SF.
(4) Executing HIDDEN.SF.
(5) Infecting uninfected apk file (namely, healthy.apk) by copying HIDDEN.SF and HIDDEN.RSA in its META-INF folder so the malicious code can be run at a later time.

If you execute HIDDEN.SF which is in infected.apk, the following tasks are performed.
(1) Changing the permission for access to HIDDEN.RSA
(2) Installing HIDDEN.RSA using pm (package manager). The icon for the app can be omitted to prevent the user from recognizing the installation of a malicious app.
(3) Executing HIDDEN.RSA in background mode to avoid detection.
(4) Uninstalling HIDDEN.RSA and removing HIDDEN.RSA's files stored in data/data folder so previous evidence of the app's installation is eliminated.

HIDDEN.RSA installed by HIDDEN.SF is an independent app which acts according to the attacker's need. Attackers can disguise any app as HIDDEN.RSA by changing its extension. This procedure is illustrated in **Fig. 9** with the tripper.apk's package name being designated as trigger.malware.sigfile.

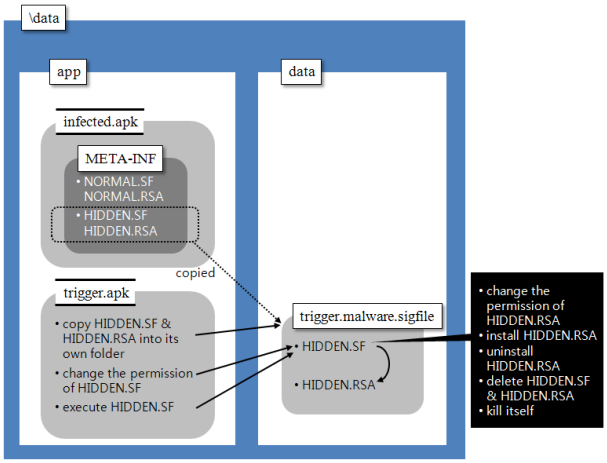

Fig. 9. The Structure and Relation of Apps

5.2 Implementation Results

We implement a test app to demonstrate how simple and easy the creation of malicious apps is by exploiting the vulnerabilities of signature management scheme in double code-signing. HIDDEN.RSA shows a message that HIDDEN.RSA is being executed and its process ID. The trigger app (trigger.apk) outputs the quantity of previously infected apps and the amount of newly infected apps.

Fig. 10. Installed Apps

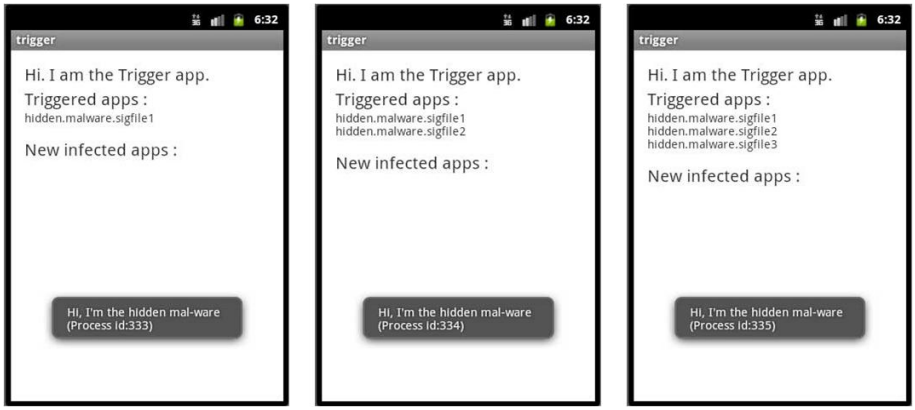

Fig. 11. Execution of the Infected Apps by the *trigger*

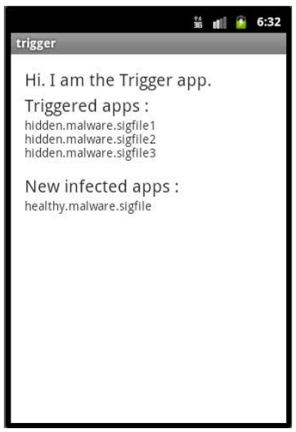

Fig. 12. The Newly Infected App by the *trigger*

Fig. 10 shows the apps installed on the device. infected1, infected2, infected3 are apps that have already been infected. Infected apps have package names of infected.malware.sigfile1, infected.malware.sigfile2 respectively, and infected.malware.sigfile3 while healthy is the uninfected app whose package name is healthy.malware.sigfile. In **Fig. 11**, the trigger app displays the package names of malicious apps in which the fake signature file, HIDDEN.RSA, is contained. The toast messages on the bottom of screen are displayed by HIDDEN.RSA. The process ID in the message shows that each message is displayed by different apps. **Fig. 12** shows the screen displaying the list of newly infected apps after the trigger app completes its run.

6. Countermeasures

As discussed previously, the malicious app that was shown in this study was made by exploiting a vulnerabilities of signature management scheme in multi-code-signing. These vulnerabilities that cause this exploiting are described below:

(1) The requirement for signature generation does not include the existing signature files.
(2) When there are multiple signatures, the installation is allowed even if only one signature is valid.
(3) If a signature file does not conform to standard format such as X.509 as requested by Jarsigner, the file is not subject to verification and causes a loop-hole.

One possible solution would be to discontinue support of multiple code-signing. However, this solution does not solve the problem of developer-market accountability. As an example, this solution would remove the mechanism where both the developer and the market independently sign the app in order to ease the burden of responsibility and simplify tracking the distribution as proposed by [20] . To do this and to ensure further change of distribution model of apps, it may be necessary to require 2 or more code signatures. Therefore improvements must be made to the structure of multi-signed app in future versions as follows:

(1) If there are pre-existing signatures when a new signature is generated, those existing signature files must be included together to create the final signature.
(2) If signature files do not meet the supported file structure, the verification must fail.

(3) If there are multiple signatures, the verification for the app must be fail if even one signature is unverifiable.

In order to satisfy these conditions, we propose a novel method of code-signing. In the following section, we discuss the algorithms, as well as double signature for scalability to support up to a multi-signature.

6.1 Code Signature Generation Process

If there are multiple signature files, the order of creation is extremely important. However, since the timestamp on the machine which created the signature file can differ from the endpoint user's system time, it is difficult to determine precedence and therefore the applicability of this method is infeasible. We solved this problem by creating independent folders for each signature. Signature file folders were created as META-INF1, META-INF2, ..., META-INF*m* with the last folder being designated META-INF. When new signatures are added, the META-INF is renamed as META-INF*m*+1 while the code signature that targets all files is generated to the new META-INF folder (See **Fig. 13**). Our proposed method attempted to vary as little as possible from the signature file creation requirements. To increase the efficiency of code-signing process, it can be modified to generate a signature based on the last META-INF*i* folder opposed to generating it from all app files. C-like pseudocode is similar to Algorithm 1.

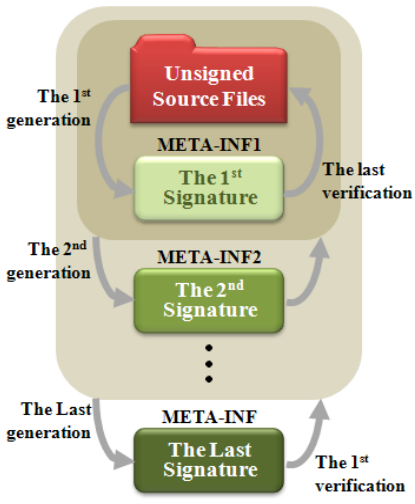

Fig. 13. Signature Folders

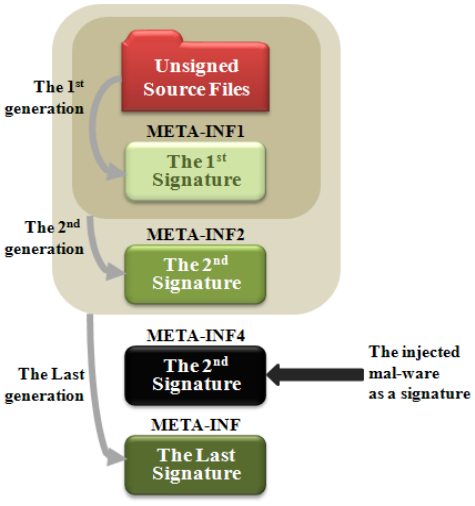

Fig. 14. Injected malware as a Signature File

Algorithm 1. Improved Multi-signature Generation Process

```
1   void GenMultiSign(Application app)
2   {
3       int     i;
4       char    *folderName;
5
6       if (! FolderExist(app, "META-INF")) {
7       // Using original Jarsigner, generate signature files in META-INF folder
8       GenOrgSign(app);
9       return;
10  }
```

```
11      // Find the latest signature folder
12      for (i=1; ; i++) {
13          folderName = MakeFolderName("META-INF", i);
14          if (! FolderExist(app, folderName))  break;
15      }
16      // The new signature folder will be META-INFi
17          Rename(app, "META-INF", folderName);
18          GenOrgSign(app);
19          return;
20      }
```

Taking into account the abnormal case such that an existing META-INF*i* series is discontinuous, we consider a scenario where there are only META-INF1 and META-INF3 by accident or with malicious intent. Our algorithm will not fail. META-INF is renamed to META-INF2 (line 14 and 17), a new signature is generated in META-INF (line 18). However, the verification process described in the next section will conclude that the signature is not valid by causing an error.

6.2 Code Signature Verification Process

The proposed signature verification procedure is shown in Algorithm 2. The verification of signatures occurs in the opposite direction of generation (see **Fig. 13**). After verifying the latest signature in META-INF, the preceding signature files in META-INF*i* are moved into META-INF and verified in turn. The verification of a single signature is depicted through Algorithm 3. As mentioned above, if META-INF*i* series number is incorrect or does not exist, signature verification fails and will be shut down. The final signature folder excluding META-INF can be found in line 14-16. If there is no META-INF*i* folder, META-INF*i*-1 is recognized as the last signature folder. As shown in **Fig. 14**, an attacker may insert malware into META-INF4 folder with a higher series than META-INF2, therefore it is still necessary to check these folders. Because the META-INF3 folder does not exist, META-INF2 is recognized as the final folder and META-INF4 is undetected. However, in this case, the verification for META-INF fails in line 9. This is because just as folders which are not META-INF are included in signature verification, so META-INF4 folder is also included.

Algorithm 2. Improved Multi-signature Verification Process

```
1   ErrorCode VeriMultiSig(Application app)
2   {
3       ErrorCode    result;
4       int           i, j;
5       char         *folderName;
6       Application app1;
7
8       if (! FolderExist(app, "META-INF")) return NO_SIGNATURE;
9       result = VeriOneSig(app);
10      if (result != SUCCESS) return result;
11      app1 = app;  // copy for verification
12      // Find the latest signature folder
13      for (i=1; ;i++) {
14          folderName = MakeFolderName("META-INF", i);
15          if (! FolderExist(app, folderName))  break;
16      }
```

```
17      // Verify the signatures starting from the latest signature
18      for (j=i-1; j>1; j--) {
19          folderName = MakeFolderName("META-INF", j);
20          DeleteFolder(app1, "META-INF");
21          RenameFolder(app1, folderName, "META-INF");
22          result = VeriOneSig(app1);
23          if (result != SUCCESS) return result;
24      }
25      return SUCCESS;
26  }
```

Each signature is verified by Algorithm 3. Since only one signature can exist in the signature folder, if there are more than one (line 7) or if they do not meet the required specifications (line 9-12), the verification process fails. When all conditions are met, verification succeeds based on previous version of process (line 14).

Algorithm 3. Improved Verification Process of One Signature

```
1   ErrorCode VeriOneSig(Application app)
2   {
3       ErrorCode   result;
4       Char        *sigFileName;
5
6       // Only one signature must be in META-INF
7       if (NumOfSig(app) > 1) return TOO_MANY_SIGNATURES;
8       // The signature file format must comply the standard format
9       result = CheckSFFormat(app);
10      if (result != SUCCESS)  return result;
11      result = CheckCertFormat(app);
12      if (result != SUCCESS)  return result;
13      // Using the original jarsigner, verify the signature
14      return VeriOrgSig(app);
15  }
```

6.3 Implementation

6.3.1 Jarsigner Implementation

In this section, we show the results of an experiment utilizing an upgraded jarsigner.

(1) doubleSigned.apk: normally double-signed by our scheme

The alias of the key using the first signature is FIRST, and the alias of the key which used the second signature is SECOND. These signatures are made using RSA algorithm. **Fig. 15** shows the file structure of doubleSigned.apk file. As we can see, FIRST.SF and FIRST.RSA are placed in META-INF1 while SECOND.SF and SECOND.RSA are placed in the META-INF folder.

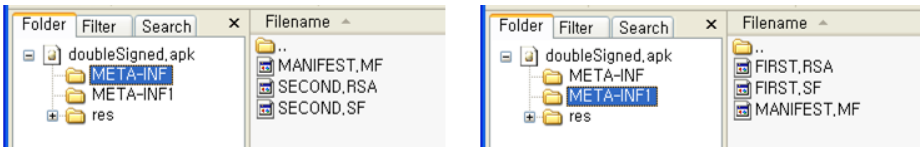

Fig. 15. Signature Files of doubleSigned.apk

Fig. 16 shows the results of a successful verification of the file doubleSigned.apk by utilizing the proposed algorithm.

```
P:Wa>jarsigner -verify doubleSigned.apk
Target[  META-INF1 ]
jar verified.
Target[  META-INF  ]
jar verified.
```

Fig. 16. Verification Result for dobleSigned.apk

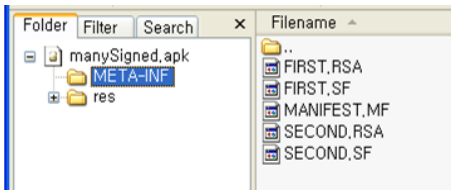

Fig. 17. Signature Files of manySigned.apk

(2) manySigned.apk: two signatures in a signature folder

This experiment shows the verification results of double-signed apps using original double signing method. **Fig. 17** shows the structure of manySigned.apk. This contains the two signatures FIRST and SECOND in the META-INF folder. **Fig. 18** shows the results as failing to validate because there are too many signatures.

```
P:Wa>jarsigner -verify manySigned.apk
Failed (Too many signature files)
```

Fig. 18. Verification Result for manySigned.apk

(3) fakeSigned.apk: falsified sinature files are injected

In this experiment, the second signature is falsified. **Fig. 19** shows the file structure of fakeSigned.apk. META-INF contains the original signature file while META-INF1 contains the fake signature. As seen from **Fig. 20**, the second is determined as not valid.

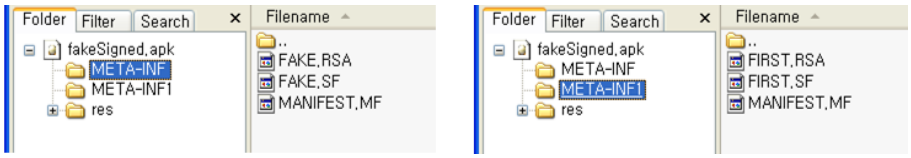

Fig. 19. File Structure of fakeSigned.apk

```
P:Wa>jarsigner -verify fakeSigned.apk
Target[  META-INF1 ]
jar verified.
Target[  META-INF  ]
jar is unsigned. (signatures missing or not parsable)
```

Fig. 20. Verification Result for fakeSigned.apk

6.3.2 Implementation on Android Platform

In Section 6.3.1, we upgraded jarsigner to show a variety of signature verification failures. However, the actual installation of the app on a device does not show a detailed error message to users. In this section, we implemented our proposed algorithm on the Android platform built on PC. To maintain consistency between the existing systems, the installation is simply stopped when signature verification fails. **Fig. 21** shows the successful installation results of an app that was double-signed by the proposed algorithm. **Fig. 22** shows the failed installations of abnormal double-signed apps. The left-side signature folder contains two signatures (see **Fig. 17**), and the right folder contains a fake signature (see **Fig. 19**).

6.4 Compatibility

The proposed signature generation procedure utilized the original Android signature procedure in order to prevent any new cryptographic vulnerability. The original signature procedure composed of several classes and methods. We didn't modify fundamental methods except one method in high level, for compatibility. The proposed signature is generated for entire files including previous signature files and the signature folder name remains the same as the previous. Therefore, the app signed by the proposed signature scheme can be verified by not only the proposed verification procedure but also the original signature verification procedure.

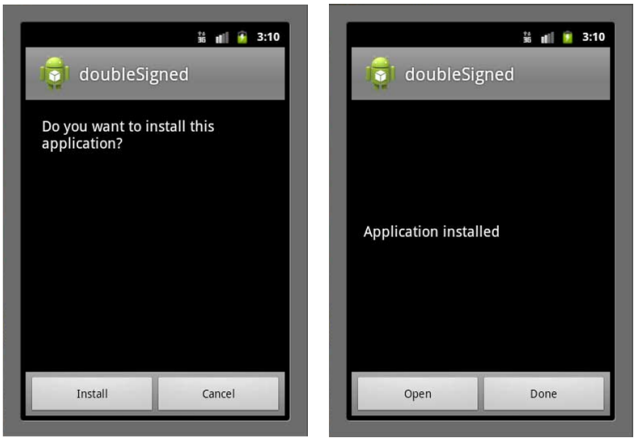

Fig. 21. Successful Installation of Double Signed App.

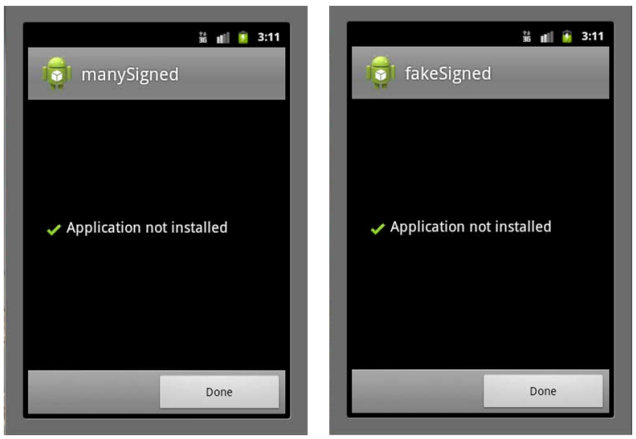

Fig. 22. Failed Installations of Double Signed Apps.

7. Conclusion and Application Method

In this study, we studied the vulnerabilities of signature management scheme and proposed a countermeasure for Android that is most widely targeted by attackers. If attackers utilize the vulnerabilities, they can more easily distribute malwares compared to the repackaging method which is currently the most popular attack method. When the proposed Android signataure/verification processes are applied, the distribution of the new type of mal-apps will be prevented in the future.

The majority of malicious Android apps are distributed through the black markets. In some markets, the developer's signature is removed and replaced by the market's signature. However, in order to determine the path of distribution for malicious apps and allow for accountability, both the market's and developer's signatures are needed. T. Cho, et al. proposed a code-signing process by which the distribution of malware can be determined by tracking either the market's signature or the developer's signature. This approach can be applied manually, without changing the existing system of double signatures. Furthermore, the scheme cannot prevent the signature folder from being abused as a malware warehouse. Our method when applied to the Android signature management scheme is a consistent way to generate and verify the signatures of the developer and market. Even if the hierarchical management system of markets and apps is formed in the future, our method can be applied to generate and verify multiple signatures.

References

[1] http://www.gartner.com/it/page.jsp?id=1421013, 2010.9.
[2] http://www.gartner.com/newsroom/id/2482816.
[3] A.P.Felt, M.Finifter, E.Chin, D.Wagner, "A Survey of Mobile Malware in the Wild," in *Proc. of ACM Workshop on Security and Privacy in Mobile Devices*, vol.17, pp.3-14, Oct, 2011. Article (CrossRef Link)
[4] Y.Zhou, X.Jiang, "Dissecting Android Malware: Characterization and Evolution," in *Proc. of IEEE Symposium on Security and Privacy*, pp.95-109, May, 2012. Article (CrossRef Link)
[5] AnserverBot, http://www.csc.ncsu.edu/faculty/jiang/AnserverBot/
[6] ADRD, http://www.f-secure.com/weblog/archives/00002100.html/
[7] Pjapps, http://www.symantec.com/security_response/writeup.jsp?docid=2011-022303-3344-99/
[8] Basebridge, http://www.ubergizmo.com/2011/05/basebridge-new-android-malware/

[9] DroidKungFu, http://www.f-secure.com/weblog/archives/00002259.html
[10] Jifake, http://www.dataprotectioncenter.com/antivirus/quickheal/malicious-qr-code-used-for-spreading-android-malware/
[11] Zitmo, http://www.securelist.com/en/blog/208193760/New_ZitMo_for_Android_and_Blackberry.
[12] Y.Zhou, X.Zhang, X.Jiang, V.W. Freeh, "Taming information-stealing smartphone applications (on Android)," In *Proc. of TRUST'11*, pp.93-107, 2011. Article (CrossRef Link)
[13] P.Hornyack, S.Han, J.Jung, S.E.Schechter, D.Wetherall, "These aren't the droids you're looking for: retrofitting Android to protect data from imperious applications," in *Proc. of the 18th ACM Conference on Computer and Communications Security*, pp.639-652, 2011. Article (CrossRef Link)
[14] W. Enck, M. Ongtang, P. McDaniel, "On Lightweight Mobile Phone Application Certification," in *Proc. of the 16th ACM Conference on Computer and Communications Security*, 2009. Article (CrossRef Link)
[15] M.Ongtang, S.E.McLaughlin, W.Enck, P.D. McDaniel, "Semantically rich application-centric security in Android," in *Proc. of the 25th Annual Computer Security Applications Conference*, pp.340-349, 2009. Article (CrossRef Link)
[16] M. Lange, S. Liebergeld, A. Lackorzynski, A. Warg, M. Peter, "L4Android: A Generic Operating System Framework for Secure Smartphones," ACM, 2011. Article (CrossRef Link)
[17] M.C.Grace, W.Zhou, X.Jiang, A.R.Sadeghi, "Unsafe exposure analysis of mobile in-app advertisements," in *Proc. of WiSec'12 ACM,* pp.101-112, 2012. Article (CrossRef Link)
[18] S.Shekhar, M.Dietz and D.S.Wallach, "AdSplit: Separating smartphone advertising from applications," In Proc. of USENIX 2012. Article (CrossRef Link)
[19] KISA, "A Survey on the trends of domestic and foreign smartphone application black markets and A Study on the code signing technology for domestic smartphone applications markets," 2011.
[20] Taenam Cho, Seung-Hyun Seo, Nammee Moon, "Double Code-Signing for Enhanced Android Application Security," *Information - An International Interdisciplinary Journal,* vol.15, no.5, pp.1913-1926, 2012. Article (CrossRef Link)
[21] Android, http://en.wikipedia.org/wiki/Android.
[22] AndroidManifest.xml, http://developer.android.com/guide/topics/manifest/manifest-intro.html
[23] Application Signing, http://developer.android.com/tools/publishing/app-signing.html.
[24] eclipse, http://www.eclipse.org/.
[25] JDK, http://javadoc.ankerl.com/.
[26] ADT, http://developer.android.com/tools/sdk/eclipse-adt.html.
[27] keytool, http://docs.oracle.com/javase/6/docs/technotes/tools/solaris/keytool.html.
[28] jarsigner - JAR Signing and Verification Tool, http://docs.oracle.com/javase/6/docs/technotes/tools/windows/jarsigner.html.
[29] zipalign, http://developer.android.com/tools/help/zipalign.html.
[30] NIST, "Digital Signature Standard," *FIPS PUB* vol. 186, no. 3, 2009.
[31] R. Rivest, A. Shamir, L. Adleman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems," *Communications of the ACM,* vol. 21, no. 2, pp. 120−126, 1978. Article (CrossRef Link)
[32] NSA, "Secure Hash Standard," *FIPS PUB*, vol. 108, no. 1, 1993.
[33] S. Josefsson, "The Base16, Base32, and Base64 Data Encodings," *RFC 4686*, 2006.
[34] D. Cooper, et al., "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile," *RFC 5280*, 2008.
[35] Cheol Jeon, Yookun Chom, Jiman Hong, "Detecting Collaborative Privacy Information Leaks on Android Applications," in *Proc. of Korea Computer Congress*, vol. 39, no. 1, pp. 92-94, 2012. Article (CrossRef Link)

Taenam Cho received her B.S., M.S.and Ph.D. in Computer Science from Ewha Womans University, Seoul, Korea. She was a senior engineer of Electronics and Telecommunications Research Institute, Korea. She is an associate professor in Department of Information Security in Woosuk University, Korea. Her major interests are cryptographic protocols, network security and smartphone security.

Seung-Hyun Seo received her B.S., M.S.and Ph.D. in Computer Science from Ewha Womans University, Seoul, Korea. She is an assistant professor at Korea University since 2015. She was a post-doctoral researcher of Computer Science at Purdue University since 2012, a senior researcher of KISA (Korea Internet and Security Agency) since 2010 and a researcher for 3 years in FSA (Financial Security Agency), Korea. She received her B.S.(2000), M.S.(2002), and Ph.D.(2006) from Ewha Womans University in Korea. Her main research interests include cryptographic protocols, mobile security, secure cloud computing, and malicious code analysis.