Wireless Pers Commun (2013) 73:1421–1437
DOI 10.1007/s11277-013-1258-x

Repackaging Attack on Android Banking Applications and Its Countermeasures

Jin-Hyuk Jung · Ju Young Kim · Hyeong-Chan Lee · Jeong Hyun Yi

Published online: 14 June 2013

Abstract Although anyone can easily publish Android applications (or apps) in an app marketplace according to an open policy, decompiling the apps is also easy due to the structural characteristics of the app building process, making them very vulnerable to forgery or modification attacks. In particular, users may suffer direct financial loss if this vulnerability is exploited in security-critical private and business applications, such as online banking. In this paper, some of the major Android-based smartphone banking apps in Korea being distributed on either the Android Market or the third party market were tested to verify whether a money transfer could be made to an unintended recipient. The experimental results with real Android banking apps showed that an attack of this kind is possible without having to illegally obtain any of the sender's personal information, such as the senders public key certificate, the password to their bank account, or their security card. In addition, the cause of this vulnerability is analyzed and some technical countermeasures are discussed.

Keywords Smartphone application vulnerability · Android app repackaging · Reverse engineering

1 Introduction

The number of Android-based smartphones is growing rapidly. As a relative latecomer to the smartphone market, one of the top priorities of Android-based smartphones is openness,

J.-H. Jung · J. Y. Kim · H.-C. Lee · J. H. Yi (✉)
School of Computer Science and Engineering, Soongsil University, Seoul, Korea
e-mail: jhyi@ssu.ac.kr

J.-H. Jung
e-mail: nemojjh@ssu.ac.kr

J. Y. Kim
e-mail: juyoungkim@ssu.ac.kr

H.-C. Lee
e-mail: iclear0708@ssu.ac.kr

so that it can quickly gain market share and be adopted by more companies. Accordingly, the Android Market the key market in which apps are distributed is also run in an open way. Anyone can register as an app developer, self-sign developed apps, and register them without having to go through a complicated procedure. In addition, Google allows for app markets run by third parties in an attempt to overcome the disadvantage of being a latecomer, which has led to rapid growth. However, this kind of open policy in which anyone can publish apps on the market has a fundamental weakness: apps containing malicious code can be published just as easily. In addition, Android apps can be easily decompiled due to their structural characteristics [7], which mean that apps modified in this way can be repackaged. Amid the current situation in which this kind of vulnerability exists, more and more Android-based financial service apps, including banking apps, are finding their way into user's hands.

We analyze whether it is possible to use repackaging attacks to forge or modify some of the most popular smartphone banking apps being distributed in South Koreas Android Market. Specifically, when a user requests a money transfer from a sender to an intended recipient in the money transfer service provided by a banking app, a forged app tries to transfer the money to an unintended attackers account instead. Whether this is actually possible is tested by publishing the forged app on the real Android Market. The experimental results showed that repackaged forged apps ran successfully for all seven of the banking apps tested. This means that attackers can successfully perform an attack with just the forged app, without needing to obtain any of the following: the sender's public key certificate, the account password, and the security card. Further, we analyze the root cause for this and investigate technical countermeasures for the vulnerability.

This paper is organized as follows. Section 2 takes a brief look at related work. Section 3 describes the results of a vulnerability analysis performed on actual banking apps. Section 4 describes the attack results of the forged app, which was actually published on the Android Market. Section 5 attempts to find countermeasures for the repackaging vulnerability. Finally, the conclusions are presented in Sect. 6.

2 Related Work

This section introduces the building and distribution processes of Android apps, repackaging vulnerability, and reverse engineering techniques for Android apps.

2.1 Android App Building

Android apps are written in Java. They are ultimately created in the form of Android Application Package (APK) files [1]. An APK is a packaged file that includes files needed for app execution. The types of files included in the app are as follows.

- *DalVik Executable (DEX) file*: The executable file resulting from compilation of the Java source code.
- *Manifest file:* A file containing app properties such as privileges, the app package file, and version.
- *eXtensible Markup Language (XML) file*: A file in which the user interface (UI) layout and values are defined.
- *Resource file*: A file containing resources required for app execution, such as images.

Figure 1 shows the series of steps for the building and packaging of files that make up the APK. First, the Java source code is compiled using the Javac compiler (included in

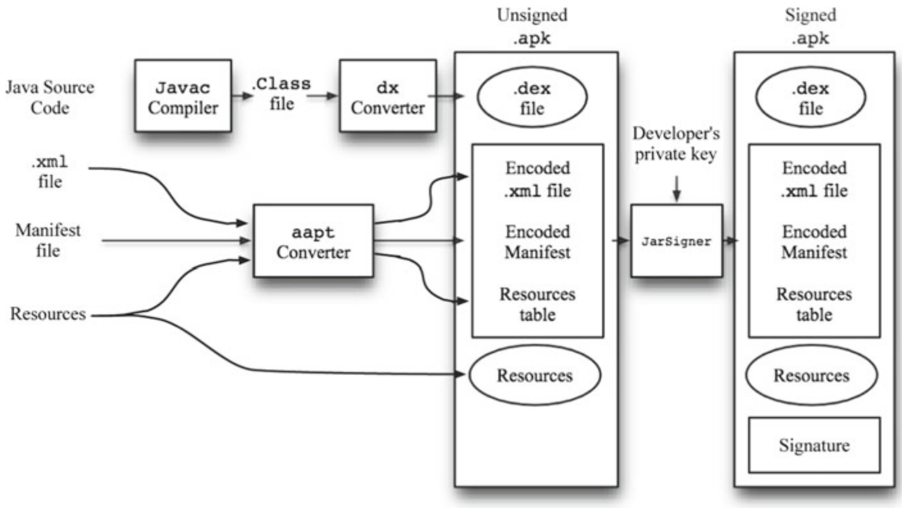


Fig. 1 Android app building process. The series of steps for the building and packaging of files are needed to make up the APK

the Java Development Kit) and outputted in the form of a class file that runs on the Java Virtual Machine (JVM). The resulting class file is converted to a DEX file using the dx converter included in the Android SDK [2]. The DEX file runs on the Dalvik Virtual Machine.

The manifest file, which is in the form of XML, and other XML files needed for app execution are encoded in binary form. After that, the DEX, XML, manifest, and resource files are packaged in an APK file, which is in ZIP format. The initially created APK file does not include the developer signature, which is needed in order to distribute it. The unsigned APK file can be self-signed with the developer's private key using Jarsigner [8]. The developer's signature and the public key are then added to the APK file, which completes the Android app building process.

2.2 Android App Distribution

The steps taken by developers to register and distribute their apps on the Android Market are as follows.

- *Developer registration*: Anyone can register as a developer for USD 25. The app developer makes a request for developer registration by sending his/her personal information and credit information to the market. The market will then check the information and approve the registration accordingly.
- *App registration*: The developer sends a self-signed app to the market and makes a request for its registration. The market will check that the app is signed and that the package name does not conflict with that of previously registered apps. If there are no problems, the app is registered in the market.
- *App distribution*: The registered app is immediately published to users and distributed to them at their request.
- *App installation and signature verification*: A check is done to see whether there are any installed apps having the same package name; if this is the case, the developer signature

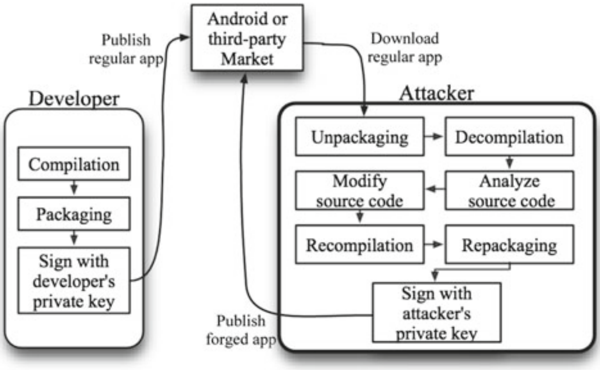


Fig. 2 Android app forgery process which exploits repackaging vulnerability

is checked to see whether it is in fact the same app. Depending on the result, the app is installed, or updated, or the installation is cancelled.

Android allows app distribution by means other than using Googles Android Market. The Android Market takes the form of an open market structure, allowing for the creation of third party markets, and distribution can be done via any distribution path, such as website downloading and side loading. This means that it is possible for developers to update their apps regardless of the distribution path; all that is required is that the updated app is self-signed with the same developer's private key. In Android, this is referred to as a "*seamless update*."

2.3 Repackaging Vulnerability

Security [3,18] is one of most important issues in various computing environments. Multiple security holes have been found in different Android components [15,20]. Among these, this paper focuses on the self-signed app repackaging problem. Reverse engineering techniques [6] can be used to create forged apps containing arbitrary code in the APK file. This vulnerability is caused by the structural characteristics of bytecode [13] that runs on a virtual machine (VM). Bytecode in object files contains character strings of class names, method names, member variable names, and so on. In addition, the methods are classified by logic, facilitating reverse engineering. Figure 2 shows the Android app forgery process, which exploits this vulnerability.

In this paper, this is referred to as "repackaging vulnerability." For DEX files, as with Java applications, object files contain code information, making them vulnerable to reverse engineering. Consequently, code similar to the original source code can be obtained by decompiling bytecode that is run on the Dalvik VM. The attacker analyzes the code obtained by reverse-engineering, inserts malicious code, and then recompiles it to create a forged DEX file with which he/she can repackage and self-sign the app with his/her private key and then distribute the forged app.

2.4 Repackaging Attack

Reverse engineering for analyzing Android apps is done primarily through decompilation of the DEX file, which can be decompiled into either Java code or smali code [16], depending on the technique used.

To obtain the Java code, tools such as undx [19] and dex2jar [5] can be used to convert the Dalvik VM bytecode into JVM bytecode and a Java decompiler can then be used to recover the Java code. The Java class files have to undergo optimization when they are converted by the dx converter during compilation for use with the Dalvik VM, resulting in the advantage of obtaining Java code that is written in a high-level language. ~~However, there is a shortcoming in that the decompilation result and the actual source code do not closely match~~. Furthermore, an identical development environment to that of the original source code must be set up in order to repackage, so this is not suitable for app modification purposes.

Obtaining the smali code, on the other hand, involves getting the code in a lower level language. In particular, it will be in the form of Dalvik VM assembly, which is mapped one-to-one with Dalvik VM bytecode [4]. Although analysis will take longer because it is in a low level language, there is the advantage that the development environment does not have to be set up to match that used for the original source code because the DEX file is created by directly converting the Dalvik VM instructions into Dalvik VM bytecode during repackaging. The latter is used in this study.

The reverse engineering process used when attacking the repackaging vulnerability using a decompiling technique is as follows.

- *Modification point search*: The activity information, UI layout, and app execution flow are gathered and the points at which code is inserted are selected. Logcat [11] can be used to gather activity names and obtain information on activities that are run during app execution. The OnCreate function of the activities can then be decompiled in order to obtain the UI information and XML information used in the UI.
- *Decompilation*: After extracting the DEX file in the APK file, a decompilation tool called baksmali [16] is used to generate the smali source code.
- *Code injection and modification*: Code containing arbitrary Dalvik VM instructions is inserted at the modification point or the existing code is modified.
- *Manifest change*: The package name is changed in the app manifest. When this is done, the app can be registered on the Android Market without conflicting with existing apps.
- *Self-signing*: The modified app is self-signed to complete the repackaging.

3 Attack on Android Banking Apps

This section describes the repackaging attack that was carried out based on analysis of the major banking apps in Korea. Table 1 lists the notations used in this paper.

3.1 Smartphone Banking Apps

Seven Android banking apps were examined in the experiments. Based on components in which processing requests and the resulting output are displayed, they can be classified into the following two types: Webview-based apps and Widget-based apps. Table 2 lists the features of the banking apps chosen for testing.

In general, money transfer using Android banking apps consist of the following four steps, as shown in Fig. 3.

Step 1 (Anti-virus Checking) This check is done before the banking apps are run to verify whether an anti-virus app is running. If no anti-virus app has yet been installed, the banking app invokes anti-virus installation and then executes the anti-virus app first. Figure 4 shows the sample code for checking whether an anti-virus app is installed.

Table 1 Symbols used in this paper are described

Symbol	Description
A	Attacker's name
S	Sender's name
R	Recipient's name
$H(x)$	Hash function on message x
ver	APK version
APK_{org}	Original APK file
APK_{mod}	Modified APK file
AC_A	Attacker's account number
AC_R	Recipient's account number
\$	Amount requested to be transferred
\$′	Amount allowed to be transferred after balance checking
PWD_S	Sender's account passwords
PSC_S	Sender's personal security card information
$Sign_A(x)$	Signature on message x using entity A's private key
ID_B	Bank identifier

Table 2 Analysis of android banking apps in Korea

Name	Type	APK version	APK hashing	Anti-virus checking	Obfuscation applied	Encryption applied
H-bank	Webview	2.12	No	Yes	No	Yes
I-bank	Widget	1.1.1	No	Yes	No	Yes
K-bank	Widget	1.8	No	Yes	Yes	Yes
N-bank	Widget	1.1	Yes	Yes	No	Yes
S-bank	Widget	2.6.6	No	Yes	No	Yes
SC-bank	Webview	1.5	No	Yes	No	Yes
W-bank	Widget	3.0.5	No	Yes	No	Yes

The features of the selected banking apps for testing are summarized

Step 2 (APK Integrity Checking) This is the core part of checking for forgery or modification of the banking apps. The procedure is as follows. The banking app calculates the hash value from the original APK file using a hash function, i.e., $h = H(APK_{org})$, and sends the APK integrity checking request (*AIC_REQ*) message containing h and APK version to the banking server. The banking server then retrieves the original APK hash value (i.e., h') with the APK version, which is kept in the banking server database. After checking whether $h = h'$, the banking server sends the APK integrity checking response (*AIC_RES*) message with the checking result to the banking app. Figure 5 shows a sample code for hashing the APK file using the SHA-256 hash function and sending the hash value to the designated URL using the APK integrity checking technique applied to N-bank.

Step 3 (Account Holder Checking) Once the integrity-checking step of the APK file is cleared, the banking app checks the account holder information. It sends the banking server the account holder checking request (*AHC_REQ*) message containing the recipient's account number (AC_R), the amount to be transferred (*\$*), and the sender's account passwords (PWD_S). If all

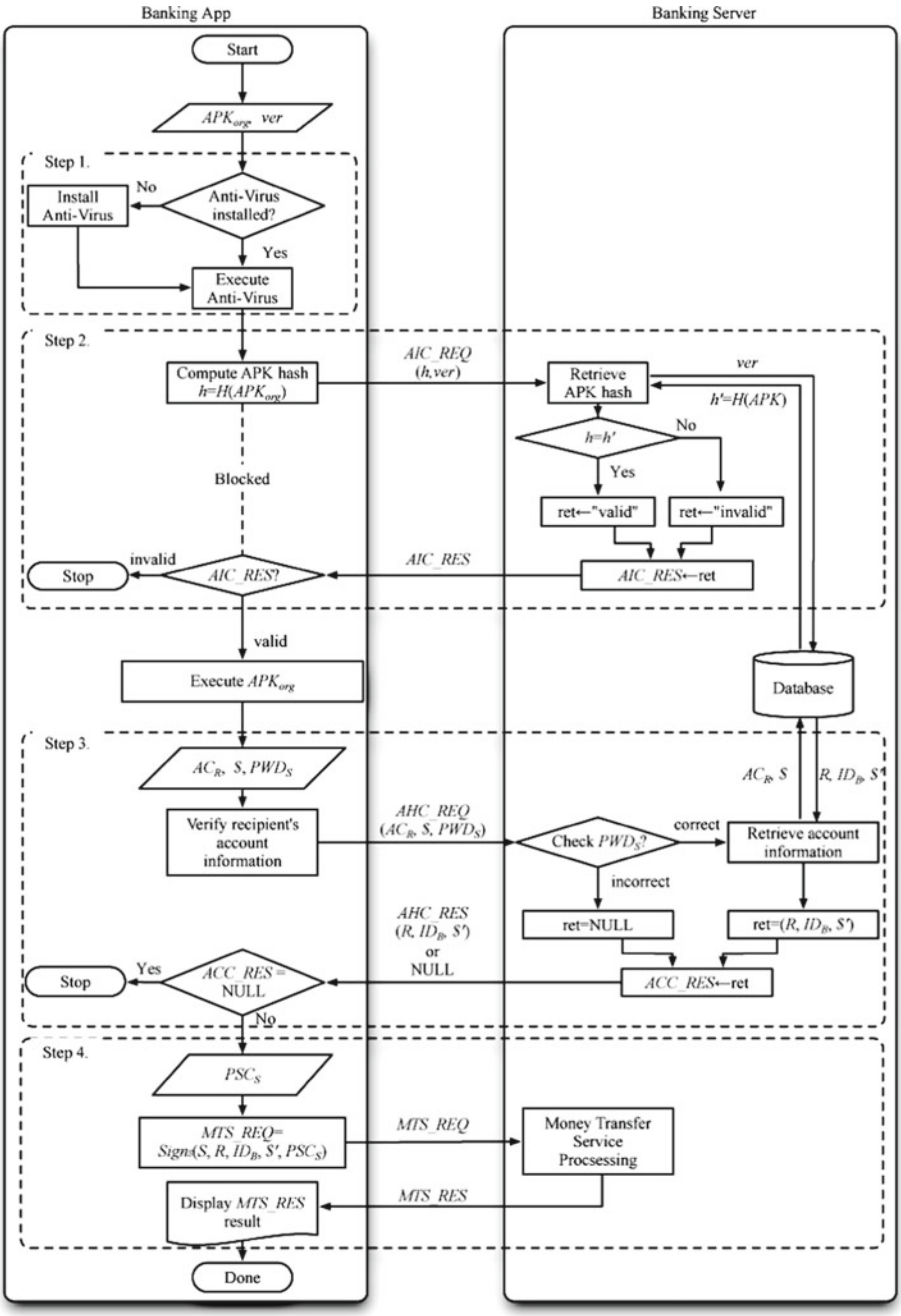
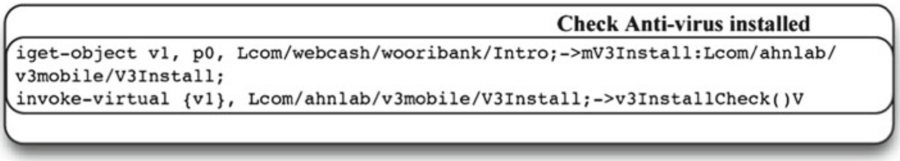


Fig. 3 Money transfer process in Android banking apps. The procedure consists of anti-virus checking, APK integrity checking, account holder checking, and money transfer service

of the account information is correct, the banking server sends the account holder checking response (*AHC_RES*) message to the recipient with the recipient's actual name (R), the recipient's bank identifier (ID_B), and the amount allowed to be transferred after balance



Check Anti-virus installed

```
iget-object v1, p0, Lcom/webcash/wooribank/Intro;->mV3Install:Lcom/ahnlab/
v3mobile/V3Install;
invoke-virtual {v1}, Lcom/ahnlab/v3mobile/V3Install;->v3InstallCheck()V
```

Fig. 4 Banking app routine for checking whether an anti-virus app is installed

checking ($\$'$). If the account information is incorrect, the banking server returns a NULL message.

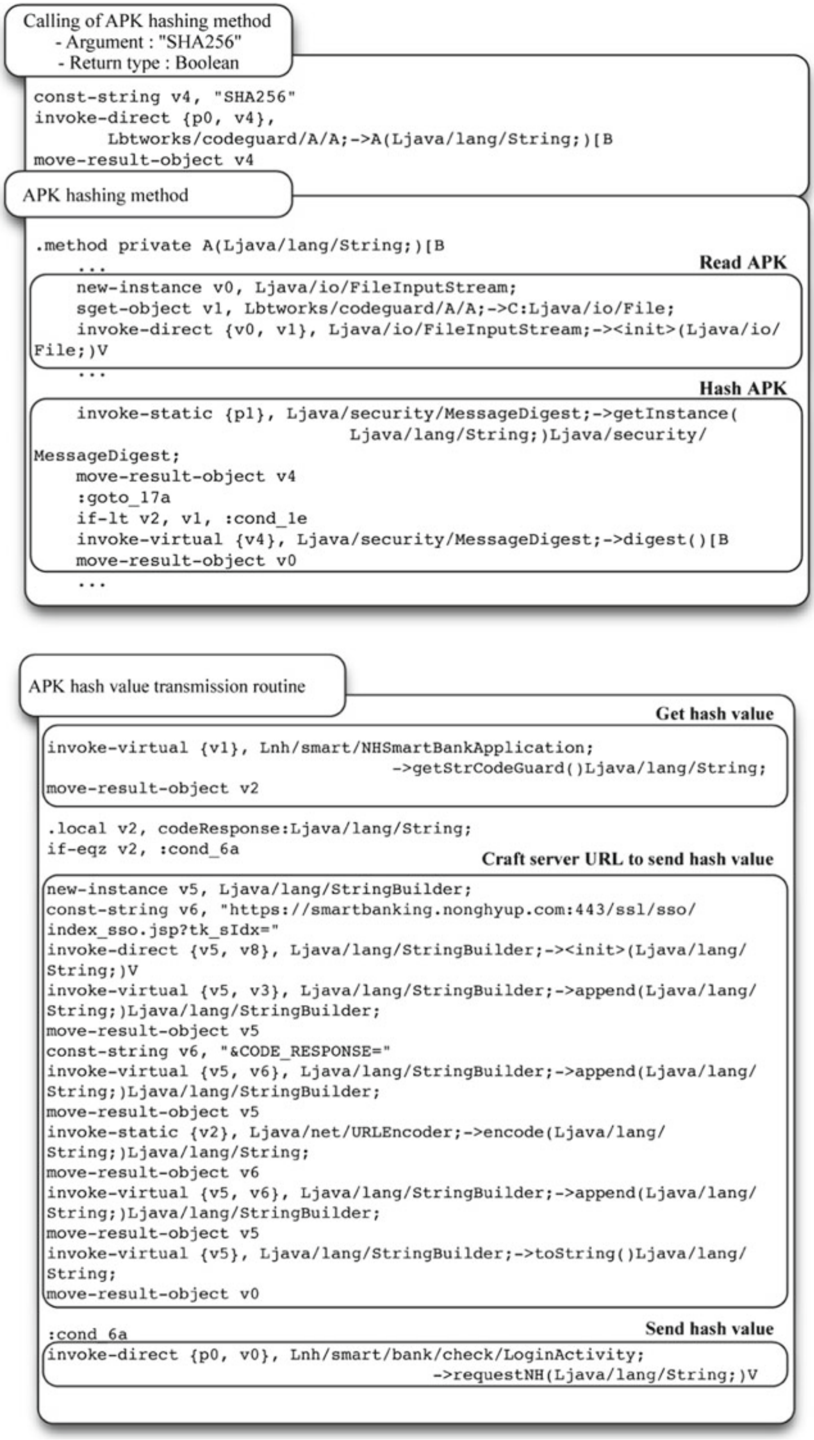
Step 4 (Money Transfer Service) This is the actual money transfer step. The banking app constructs a money transfer service request (*MTS_REQ*) message with the sender's name (*S*), recipient's name (*R*), recipient bank identifier (ID_B), amount ($\$'$), and the personal security card number (PSC_S). The sender then signs (*MTS_REQ*) with its private key. Upon receiving the signed (*MTS_REQ*) message, the banking server processes the money transfer service. The detailed process involved in this step is beyond the scope of this paper.

3.2 Repackaging Attack on Banking Apps

When attacking a banking app by exploiting a repackaging vulnerability, the attackers may attempt direct attacks that may cause financial loss for the user, such as unauthorized money transfers and stealing passwords. This section describes a particular type of attack in which a money transfer is made not to the intended recipient, but to the attacker, using a forged app.

The attack is carried out as follows (see Fig. 6). The anti-virus checking routine can be easily skipped. In the APK integrity-checking step, the forged app sends the hash value of the original APK file, not that of the forged APK file. It simply cheats the banking server. Regardless of what the (*AIC_RES*) message returns, the forged app proceeds with the forged APK and not the original APK. To obtain the intended recipient's name and the attacker's name, the forged app sends the (*AHC_REQ*) message to the banking server twice: one with the intended recipient account number (AC_R) and the other with the attacker account number (AC_A). Owing to space constraints, we have skipped the step involving *AHC_REQ*(AC_R, \$, PWD_S) in Fig. 6. In the final step, the forged app requests the money transfer service with the attacker's account information using the sender's original security card information and the sender's private key. When receiving the result from the banking server, it ignores that result containing the attacker's account information. Instead, it displays the forged result with the intended recipient's account information.

After decompiling, the OnCreate method is analyzed in the activity class in order to obtain the UI information. The attacker can discover that the OnClick handler for the Button class (among the UI components) processes the money transfer service request and that the money transfer request confirmation details are outputted using TextView. The attacker can change the corresponding code to make it such that money is transferred to the attacker's account when the user uses the money transfer service, while making it appear to the user as if money was correctly transferred to the intended recipient. The recipient's information is shown to the user, although the forgery app actually does not check the account of the recipient that the user entered, but that of the attacker, as shown in Step 4. The user proceeds to the next step without suspicion, and information is also shown confirming that the money was transferred



```
const-string v4, "SHA256"
invoke-direct {p0, v4},
       Lbtworks/codeguard/A/A;->A(Ljava/lang/String;)[B
move-result-object v4
```

```
.method private A(Ljava/lang/String;)[B
    ...
```

```
    new-instance v0, Ljava/io/FileInputStream;
    sget-object v1, Lbtworks/codeguard/A/A;->C:Ljava/io/File;
    invoke-direct {v0, v1}, Ljava/io/FileInputStream;-><init>(Ljava/io/
File;)V
    ...
```

```
    invoke-static {p1}, Ljava/security/MessageDigest;->getInstance(
                             Ljava/lang/String;)Ljava/security/
MessageDigest;
    move-result-object v4
    :goto_17a
    if-lt v2, v1, :cond_1e
    invoke-virtual {v4}, Ljava/security/MessageDigest;->digest()[B
    move-result-object v0
    ...
```

```
invoke-virtual {v1}, Lnh/smart/NHSmartBankApplication;
                             ->getStrCodeGuard()Ljava/lang/String;
move-result-object v2

.local v2, codeResponse:Ljava/lang/String;
if-eqz v2, :cond_6a
```

```
new-instance v5, Ljava/lang/StringBuilder;
const-string v6, "https://smartbanking.nonghyup.com:443/ssl/sso/
index_sso.jsp?tk_sIdx="
invoke-direct {v5, v8}, Ljava/lang/StringBuilder;-><init>(Ljava/lang/
String;)V
invoke-virtual {v5, v3}, Ljava/lang/StringBuilder;->append(Ljava/lang/
String;)Ljava/lang/StringBuilder;
move-result-object v5
const-string v6, "&CODE_RESPONSE="
invoke-virtual {v5, v6}, Ljava/lang/StringBuilder;->append(Ljava/lang/
String;)Ljava/lang/StringBuilder;
move-result-object v5
invoke-static {v2}, Ljava/net/URLEncoder;->encode(Ljava/lang/
String;)Ljava/lang/String;
move-result-object v6
invoke-virtual {v5, v6}, Ljava/lang/StringBuilder;->append(Ljava/lang/
String;)Ljava/lang/StringBuilder;
move-result-object v5
invoke-virtual {v5}, Ljava/lang/StringBuilder;->toString()Ljava/lang/
String;
move-result-object v0

:cond_6a
```

```
invoke-direct {p0, v0}, Lnh/smart/bank/check/LoginActivity;
                             ->requestNH(Ljava/lang/String;)V
```

Fig. 5 APK integrity checking routine used in the N-bank app. It hashes the APK file using the SHA-256 hash function and sends the hash value to the designated URL

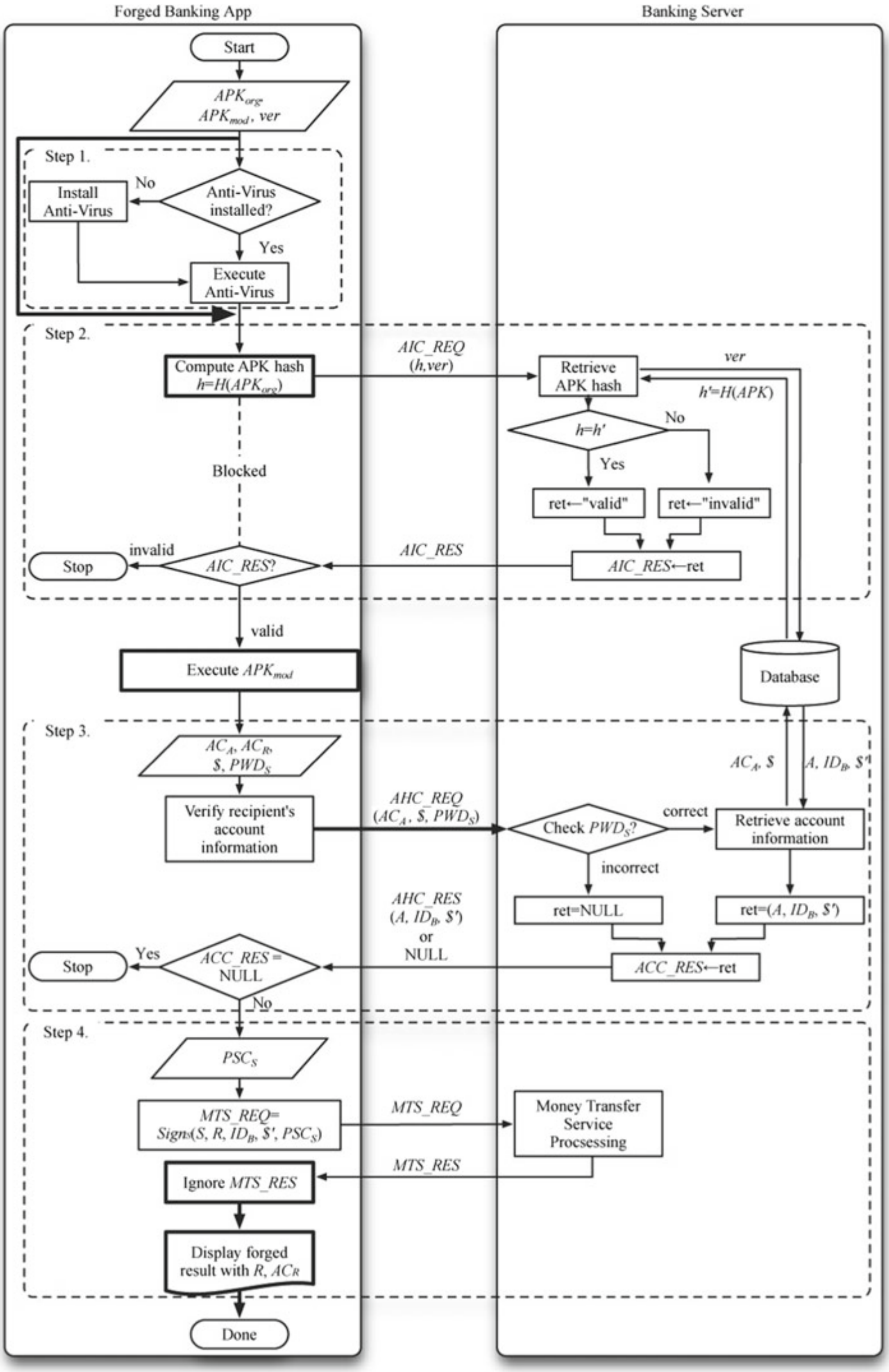
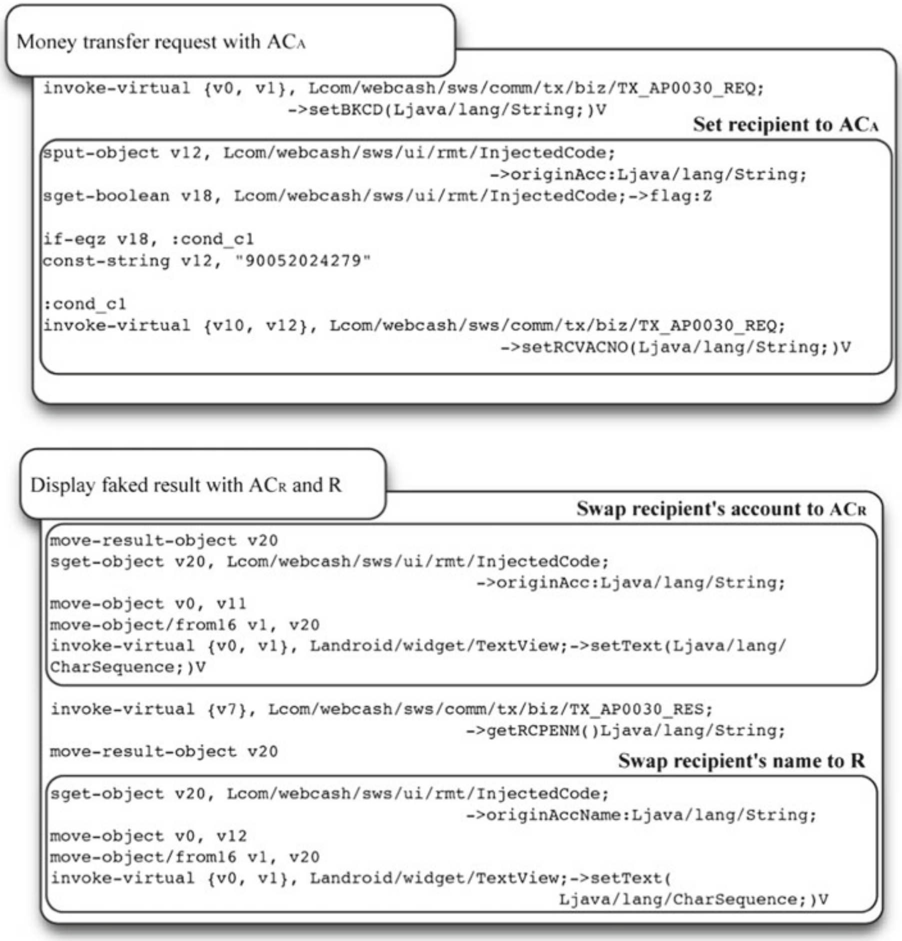


Fig. 6 Cheating money transfer with forged apps which request the service with the attacker's account information


Money transfer request with AC_A
Set recipient to AC_A
Display faked result with AC_R and R
Swap recipient's account to AC_R
Swap recipient's name to R

```
invoke-virtual {v0, v1}, Lcom/webcash/sws/comm/tx/biz/TX_AP0030_REQ;
                      ->setBKCD(Ljava/lang/String;)V
sput-object v12, Lcom/webcash/sws/ui/rmt/InjectedCode;
                                        ->originAcc:Ljava/lang/String;
sget-boolean v18, Lcom/webcash/sws/ui/rmt/InjectedCode;->flag:Z

if-eqz v18, :cond_c1
const-string v12, "90052024279"

:cond_c1
invoke-virtual {v10, v12}, Lcom/webcash/sws/comm/tx/biz/TX_AP0030_REQ;
                                        ->setRCVACNO(Ljava/lang/String;)V
```

```
move-result-object v20
sget-object v20, Lcom/webcash/sws/ui/rmt/InjectedCode;
                                        ->originAcc:Ljava/lang/String;
move-object v0, v11
move-object/from16 v1, v20
invoke-virtual {v0, v1}, Landroid/widget/TextView;->setText(Ljava/lang/
CharSequence;)V

invoke-virtual {v7}, Lcom/webcash/sws/comm/tx/biz/TX_AP0030_RES;
                                       ->getRCPENM()Ljava/lang/String;
move-result-object v20
sget-object v20, Lcom/webcash/sws/ui/rmt/InjectedCode;
                                       ->originAccName:Ljava/lang/String;
move-object v0, v12
move-object/from16 v1, v20
invoke-virtual {v0, v1}, Landroid/widget/TextView;->setText(
                                                Ljava/lang/CharSequence;)V
```

Fig. 7 Modified code injected in the forged banking app

to the intended account in the last step showing the results of the money transfer, fooling the user. Figure 7 shows the modified code in the actual banking app.

The InjectedCode class inserted by the attacker swaps the account number, the bank name, and the recipient name. A swap routine is inserted into a routine that has finished the analysis in order to forge the money transfer service. The code modified as above is compiled using the smali tool to create a DEX file and the package name is modified in the manifest file in order to avoid package name conflict when registering on the market. The created DEX file and the manifest file are repackaged and self-signed with the attacker's private key to complete the repackaging attack for the banking app.

4 Experimental Results

In this section, the attack technique proposed in Sect. 3 is applied to an actual banking app and the subsequent experimental results are described.

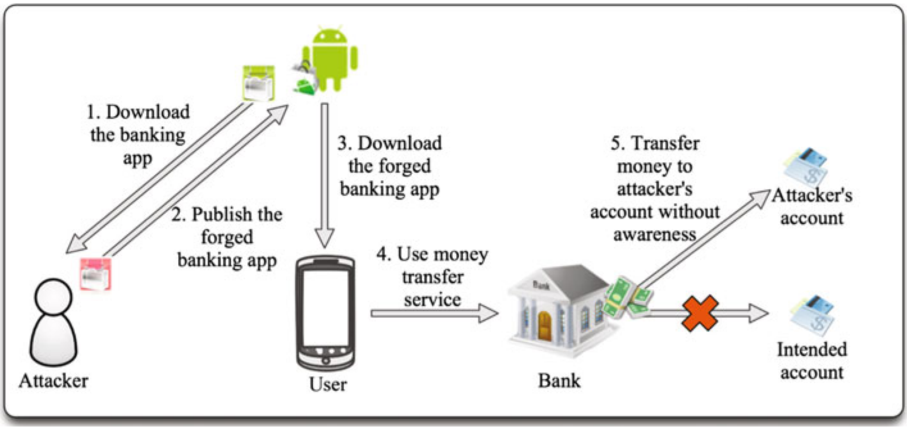

1. Download the banking app
2. Publish the forged banking app
3. Download the forged banking app
4. Use money transfer service
5. Transfer money to attacker's account without awareness
Attacker's account
Intended account
Attacker
User
Bank

Fig. 8 Attack scenario. An attacker distributes a forged app so that the remittance is made not to the intended recipient, but to the attacker himself

4.1 Attack Scenario

Figure 8 shows an attack scenario in which an attacker distributes a forged app so that when the user enters the recipient's account in a money transfer, the remittance is made not to the intended recipient, but to the attacker himself without being recognized by the user.

4.2 Experimental Setup

The user equipment setup used for the experiment was Nexus One (Snap-dragon 1 GHz CPU; 512 MB RAM; Android 2.3.6). Logcat, which comes with the Android SDK, was used to inspect the activity classes to be forged. The smali and baksmali tools were used to forge the banking app. Jarsigner was used to self-sign the forged app.

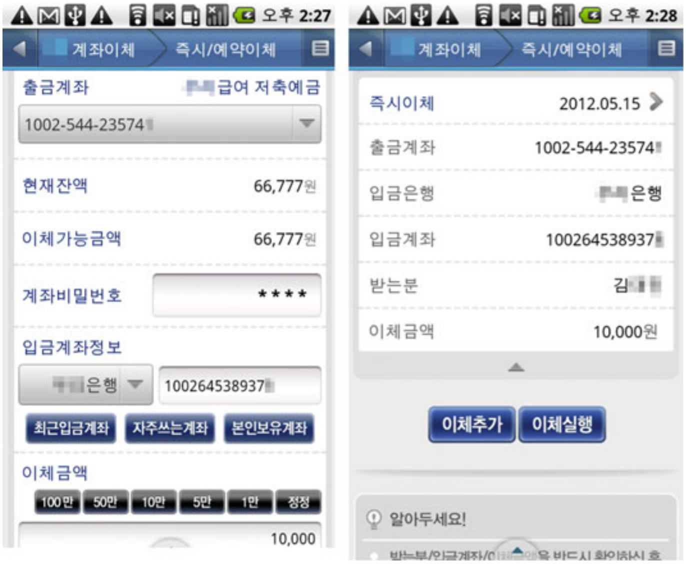
4.3 Experiment Results

Figure 9 shows the execution screen of the forged W-bank app. The user is currently requesting that a money transfer be made to the recipient (i.e., Kim).

Figure 10 shows that the money transfer was made to the recipient, as requested (i.e., Kim).

However, as shown in Fig. 11, the money transfer was not made to the intended recipient (i.e., Kim), but to the attacker (i.e., Jung) instead. Besides the W-bank app, all of the seven banking apps were tested under the same attack scenario and the attack was found to be successful for all of them.

It was also found that anti-virus products used by the tested Android banking apps could not detect forgery. Even if the signature DB of the anti-virus app was to be updated to detect forgery, the anti-virus installation routine can be skipped and easily bypassed in the forged app. In addition, banking apps use APK integrity checking by sending APK hash information to the server. This is also part of the code included in the app, so it was confirmed that the attacker can still make the forged app run as intended by sending the original APK hash value, rather than the hash value of the forged app, or by circumventing the check routine altogether. Accordingly, the security solutions used in today's banking apps do all of their processing



오후 2:27

계좌이체 즉시/예약이체

출금계좌	[redacted]급여 저축예금
	1002-544-23574[redacted]
현재잔액	66,777원
이체가능금액	66,777원
계좌비밀번호	****
입금계좌정보	
[redacted]은행	100264538937[redacted]

최근입금계좌 자주쓰는계좌 본인보유계좌

이체금액

100만 50만 10만 5만 1만 정정

10,000

오후 2:28

계좌이체 즉시/예약이체

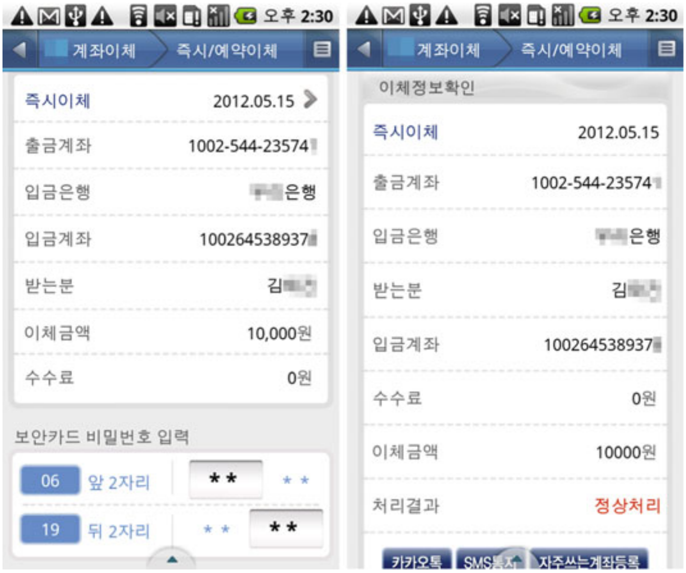
즉시이체	2012.05.15
출금계좌	1002-544-23574[redacted]
입금은행	[redacted]은행
입금계좌	100264538937[redacted]
받는분	김[redacted]
이체금액	10,000원

이체추가 이체실행

알아두세요!

받는분/입금계좌/이[illegible]을 반드시 확인하신 후

Fig. 9 Money transfer request made to the recipient (i.e., Kim) using the forged W-bank app



오후 2:30

계좌이체 즉시/예약이체

즉시이체	2012.05.15
출금계좌	1002-544-23574[redacted]
입금은행	[redacted]은행
입금계좌	100264538937[redacted]
받는분	김[redacted]
이체금액	10,000원
수수료	0원

보안카드 비밀번호 입력

06	앞 2자리	**	* *
19	뒤 2자리	* *	**

오후 2:30

계좌이체 즉시/예약이체

이체정보확인

즉시이체	2012.05.15
출금계좌	1002-544-23574[redacted]
입금은행	[redacted]은행
받는분	김[redacted]
입금계좌	100264538937[redacted]
수수료	0원
이체금액	10000원
처리결과	정상처리

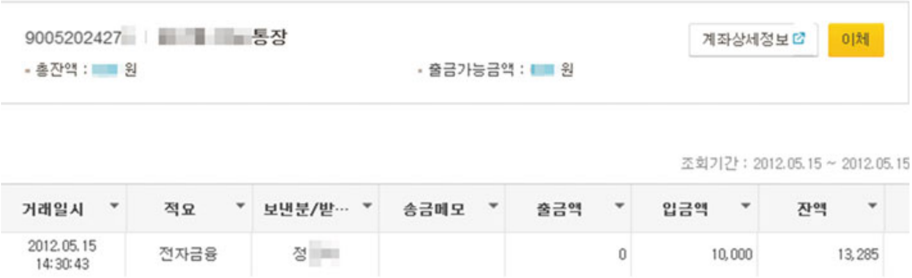
카카오톡 SMS[illegible] 자주쓰는계좌등록

Fig. 10 Money transfer response made to the recipient (i.e., Kim) using the forged W-bank app

at the application layer; therefore, they fundamentally cannot prevent repackaging attacks, whose exploit is based on making changes to the app itself.

5 Countermeasures

This section describes countermeasures against repackaging attacks.



9005202427[redacted] | [redacted]통장

- 총잔액 : [redacted] 원
- 출금가능금액 : [redacted] 원

조회기간 : 2012.05.15 ~ 2012.05.15

거래일시	적요	보낸분/받…	송금메모	출금액	입금액	잔액
2012.05.15 14:30:43	전자금융	정[redacted]		0	10,000	13,285

Fig. 11 Attacker’s account information. The actual money transfer was made to the attacker (i.e., Jung)

5.1 Self-Signing Restriction

Self-signing policy prevention can be considered as a countermeasure against repackaging attacks. The easiest way to do this is to change app signing from self-signing to signed-by-market and prevent app distribution without the market’s signature. Although this fundamentally eliminates repackaging attacks, it would violate Android’s open policy and the seamless update advantage would be lost. In addition, this method is not very practical because all software that has already been distributed to smartphone devices would have to be replaced with software that does not support self-signing. This restriction creates a requirement for a technique for maintaining compatibility with the existing system while addressing the vulnerability. One such technique is the multi-signature-based app signing scheme [9], which can minimize changes to the existing system while meeting the seamless update requirement.

5.2 Code Obfuscation

Obfuscation is a technique used for making reverse engineering of source code or machine code more difficult. Although it cannot make reverse engineering completely impossible, it can prevent exposure of logic or code by obstructing analysis. Obfuscation comes in two types according to the type of target:source code obfuscation and binary code obfuscation. Moreover, it is classified into layout obfuscation, data obfuscation, and control obfuscation, according to the specific techniques used. Proguard [14] is the default obfuscation tool provided by Android and is used for Java source obfuscation. That is, a DEX file is not directly obfuscated, but rather the class file is obfuscated and then a DEX file is created using the dx converter. Proguard supports layout obfuscation and changes class names, method names, and member variable names into meaningless, randomized character strings in order to obfuscate the code for class names and method names, etc. However, there is no change before and after obfuscation with regard to software control logic, so the attacker can discover the roles played by classes and methods by analyzing the software logic; the analysis just takes time in this case. In the example of K-bank app (see Table 2), although their banking app was obfuscated using Proguard, it unfortunately did not protect against our attack. To overcome this weakness, a logic obfuscation technique such as control obfuscation is necessary, but control obfuscation generally has the shortcoming of lowering code execution performance. Therefore, it is necessary to develop obfuscation techniques that do not impair performance in the limited smartphone environment while increasing the level of obfuscation.

5.3 Code Attestation

The Trusted Platform Module (TPM) [17] is one of the strongest security measures available for smartphone apps that deal with sensitive data, such as banking apps. It is one of hardware-based platform security solutions [10,12]. Secure booting from the boot loader to the OS kernel and library module loading is made possible by TPM. Privilege isolation between apps is also made possible, along with remote attestation, which remotely checks platform integrity. In particular, remote attestation is a useful technique for checking whether the user app has been forged before financial data is exchanged between the banking server and the user app, such as in the banking app example. In addition, static integrity checking of binary execution code can be done on a smartphone and malicious forged apps can also be identified at execution time using remote attestation. Therefore, even though there would be increased costs from additional hardware, there is a need for seriously reviewing the introduction of a TPM-based app integrity verification technique.

6 Conclusion

Malicious software that can cause leakage of personal information and/or financial loss is rapidly on the rise. For example, security threats against smartphone banking apps can directly cause damage to users. In this paper, an experiment was done on the feasibility of exploiting repackaging attacks on banking apps provided by the major banks in South Korea, which are distributed on the Android Market. The forged banking apps were tested and it was confirmed that when the user uses the money transfer service, money is not transferred to the account of the intended recipient, but to that of the attacker. It was also found that security techniques currently in use to prevent this were completely ineffective. The reason for this is that they all run in the application layer, which means they can be bypassed when repackaging is done. Accordingly, binary code obfuscation and hardware-based code attestation were proposed as countermeasures for tackling the root cause of the problem.

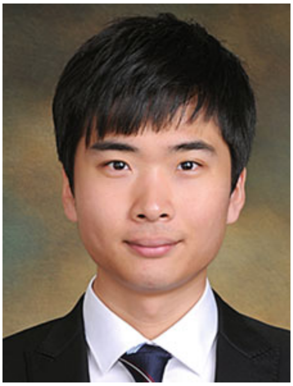
Acknowledgements This research was supported by the MSIP (Ministry of Science, ICT & Future Planning), Korea, under the ITRC (Information Technology Research Center) support program (NIPA-2013-H0301-13-1003) supervised by the NIPA (National IT Industry Promotion Agency).

References

1. Android, A. P. K. http://developer.android.com/guide/market/expansion-files.html.
2. Android Developers SDK. http://developer.android.com/sdk/.
3. Bringer, J., & Chabanne, H. (2012). Embedding edit distance to enable private keyword search. *Human-centric Computing and Information Sciences*, *2*(1), 2.
4. Dalvik VM Bytecode. http://source.android.com/tech/dalvik/instruction-formats.html.
5. Dex2jar. http://code.google.com/p/dex2jar/.
6. Eilam, E. (2011). *Reversing: Secrets of reverse engineering*. New York: Wiley.
7. Enck, W., Octeau, D., McDaniel, P., & Chaudhuri, S. (2011). A Study of android application security. In: *Proceedings of the 20th USENIX security, symposium*, pp. 21–21.
8. Jarsigner. http://docs.oracle.com/javase/1.3/docs/tooldocs/win32/jarsigner.html.
9. Lee, H. C., Jung, J. H., & Yi, J. H. Multi-signature based integrity checking scheme for detecting modified applications on android. *Information Journal* (To appear).

10. Li, T., Yu, F., Lin, Y., Kong, X., & Yu, Y. (2010). Trusted computing dynamic attestation using a static analysis based behaviour model. *Journal of Convergence*, *2*(1), 61–68.
11. Logcat. http://developer.android.com/guide/developing/tools/logcat.html.
12. Malakuti, S., Aksit, M., & Bockisch, C. (2011). Runtime verification in distributed computing. *Journal of Convergence*, *2*(1), 1–10.
13. Miecznikowski, J., & Hendren, L. (2002). Decompiling java bytecode: Problems, traps and pitfalls. In: R. Nigel Horspool (Ed.), *Compiler construction*, (pp. 153–184). Berlin: Springer.
14. Proguard. http://proguard.sourceforge.net/.
15. Shabtai, A., Fledel, Y., Kanonov, U., Elovici, Y., Dolev, S., & Glezer, C. (2010). Google android: A comprehensive security assessment. *IEEE Security and Privacy*, *8*(2), 35–44.
16. An Assembler(smali) and disassembler(baksmali) for androids dex format. http://code.google.com/p/smali/.
17. Trusted Computing Group (TCG). (2011). TPM main specification level 2 spec. ver. 1.2, Rev. 16.
18. Tseng, F. H., Chou, L. D., & Chao, H. C. (2011). A survey of black hole attacks in wireless mobile ad hoc networks. *Human-Centric Computing and Information Sciences*, *1*(1), 1–16.
19. UNDX. http://sourceforge.net/projects/undx/.
20. Vidas, T., Votipka, D., & Christin, N. (2011). All your droid are belong to Us: A survey of current android attacks. In: *Proceedings of the 5th USENIX workshop on offensive technologies*, pp. 10–10.

Author Biographies



Jin-Hyuk Jung received his B.S. degrees and M.S. degrees in Computer Science from Soongsil University, Seoul, Korea, in 2011 and 2013, respectively. He is a Ph. D. student in School of Computer Science and Engineering, Soongsil University. His research interests include mobile application security and mobile platform security.

Ju Young Kim received his B.S. degrees in Computer Science from Soongsil University, Seoul, Korea, in 2012. He is currently working for Samsung Electronics.

Hyeong-Chan Lee is a member of research staff in National Security Research Institute (NSRI), Korea. He received his B.S. and M.S. degrees in Computer Science from Soongsil University, Seoul, Korea, in 2010 and 2012, respectively. His research interests include mobile application security and mobile platform security.

Jeong Hyun Yi is an Assistant Professor in the School of Computer Science and Engineering at Soongsil University, Seoul, Korea. He received the B.S. and M.S. degrees in computer science from Soongsil University, Seoul, Korea, in 1993 and 1995, respectively, and the Ph.D. degree in information and computer science from the University of California, Irvine, in 2005. He was a Principal Researcher at Samsung Advanced Institute of Technology, Korea, from 2005 to 2008, and a member of research staff at Electronics and Telecommunications Research Institute (ETRI), Korea, from 1995 to 2001. Between 2000 and 2001, he was a guest researcher at National Institute of Standards and Technology (NIST), Maryland, U.S. His research interests include mobile security and privacy, network security, cloud computing security, and applied cryptography. Some of his notable research contributions include Certificate Management Protocol (CMP) for Korean PKI Standards and integration of Korea PKI and U.S. Federal PKI.