An Android Application Protection Scheme against Dynamic Reverse Engineering Attacks

Kyeonghwan Lim[1], Younsik Jeong[1], Seong-je Cho[1], Minkyu Park[2], and Sangchul Han[2*]
[1]*Dept. of Computer Science and Engineering, Dankook University*
Yongin, Gyeonggi 16890, South Korea
{limkh,jeongyousik,sjcho}@dankook.ac.kr
[2]*Dept. of Computer Engineering, Konkuk University*
Chungju, Chungbuk 27478, South Korea
{minkyup,schan}@kku.ac.kr

Abstract

Reverse engineering of Android applications is easy because the applications are written in the high level but simple bytecode language. Due to malicious reverse engineering attacks, many Android applications are tampered and repackaged into malicious applications. To protect Android applications from reverse engineering, many research studies have proposed and developed anti-reverse engineering techniques such as obfuscation, packing (packed executable), encryption, and anti-debugging. Obfuscation, packing and encryption are the defense techniques against static reverse engineering, which cannot prevent dynamic reverse engineering like memory dumping and runtime debugging. On the other hand, the existing defense techniques against dynamic reverse engineering have usually tried to protect applications by determining whether they are being executed on an emulation-based analysis environment and stopping their execution on the emulator. However, the protection techniques based on detecting the emulators become ineffective because attackers recently employ dynamic reverse engineering directly on real mobile devices. This paper presents a new protection scheme for Android applications against dynamic reverse engineering which can be applied on real mobile devices. Our scheme checks if a device on which the application is running is rooted and/or the application is being debugged. If so, our scheme stops the execution of the application. Our experiments demonstrate that the rooted/debugging environments detection techniques can be evaded by method hooking attacks and that the evasion attack, fortunately, can be detected by our scheme. One of the strength of our approach is that it is not implemented as a part of application's source code but a separate executable. It can be applied to applications whose source code is not available.

Keywords: Application Protection, Reverse Engineering, Evasion Attack, Android, Rooting

1 Introduction

Software reverse engineering is a process of analyzing the structure of program and its behavior to know more about how the program works and operates. Reverse engineering can be used for the purpose of learning how a program works. It can also be used for creating a new competitive program at a low cost by understanding the structure and behavior of an existing program. By decompiling or reverse engineering an executable, one can get the source code of a program, can know the technology working behind a program, and can design a new program by taking some ideas from the existing programs.

On the other hand, reverse engineering can be used to tamper or hack mobile applications to get illegal benefits such as bypassing authentication or payment [1]. Hacked or tamper mobile applications

Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications, 7:3 (September 2016), pp. 40-52
*Corresponding author: Department of Computer Engineering, Konkuk University, 268 Chungwondaero, Chungju-si, Chungbuk-do, 27478, Korea, Tel: +82-43-840-3605

that control medical devices [2] can even threaten human life. According to the report of Arxan Technologies, one of well-known application protection solution providers, 97 % of the top paid Android applications (or apps) and 80 % of the top free Android apps have been hacked [3]. The usual process of hacking apps is as follows: (1) Select a target app, (2) Reverse-engineer app contents, (3) Extract and steal confidential data, (4) Create a cracked, patched or tampered version of the app, or Use malware to patch/infect the app on other devices, and (5) Release/use the hacked app [3].

This leads to an increasing interest to shield program codes of Android apps against reverse engineering. Anti-reverse engineering can be helpful to prevent software piracy, to guard intellectual property, and to protect applications from malicious attacks. There is another motivation for anti-reverse engineering. App developers are interested in protecting their apps.

To prevent copyright infringement of Android apps, there are many research on obfuscation techniques and anti-static reverse engineering techniques [4, 5, 6, 7, 8]. However, these techniques do not consider dynamic reverse engineering such as memory dumping [9]. Although there are some techniques that prevent emulator-based dynamic reverse engineering [10, 11], they are little effective on real device-based dynamic reverse engineering techniques [12]. Another problem is that the existing anti-dynamic reverse engineering techniques are implemented as a part of application's source code. This implies that these techniques canoot be appliced to apps whose source codes are not available.

In this paper, we propose a scheme that protects Android apps against dynamic reverse engineering on real mobile devices. Our scheme includes (1) rooted/debugging environments detection techniques, (2) a call stack-based evasion detection technique that detects method hooking attacks (for evading rooted/debugging environments detection), and (3) a dynamic code loading technique that enables protection of compiled (packaged) executables. Our experiments demonstrate that the rooted/debugging environments detection techniques can be evaded by method hooking attacks and that the evasion attacks, fortunately, can be detected by our scheme. We also show that our scheme can be applied to any Android application package without its source code.

The rest of this paper is organized as follows. Section 2 overviews the existing anti-reverse engineering techniques. In Section 3, we explain the proposed scheme in detail. Section 4 presents our experimental results. Section 5 concludes this work.

2 Related Work

Many studies have been presented to protect software from reverse engineering attacks. The popular anti-reverse engineering techniques include obfuscation of program code, encryption of executables, anti-debugging, etc. Many techniques are available, however none of them provides perfect protection against reverse engineering.

ProGuard [5] is an obfuscation tool integrated in the Android Software Development Kit. The tool transforms original identifiers for packages, classes, and methods into obscured ones. It can also find out unused classes and dead codes, and remove them automatically and manually.

Patrick Schulz [6] presented several code obfuscation methods on the Android platform such as identifier mangling, string obfuscation, dynamic code loading, dead code management, and self-modifying code. Identifier mangling neutralizes the information of original identifiers in order to make them not to be easily analyzed by a reverse engineer. String obfuscation transforms an original string into another string. Dynamic code loading method employs two components: an encrypted executable (encrypted dex file) and a decryption stub. The decryption stub performs three main steps. The first step fetches the encrypted executable into memory by extracting the encrypted dex file from an internal data structure or downloading it from a remote server. The second step decrypts the encrypted dex file and restores the

Table 1: Rooting detection methods presented in [16]

Detection methods	Description
D1: Check installed packages	check whether rooting-related apps are installed.
D2: Check installed files	check whether binary files (such as `busybox` and `su`) that usually appear in rooted devices are installed.
D3: Check the BUILD tag	check whether Android image on the device is a stock image or a custom image built by third-party developers.
D4: Check system properties	check the value of system properties, `ro.debuggable` and `ro.secure`, that allow root-privileged shell.
D5: Check directory permission	check whether write permission is given to some directories that should have read-only permission.
D6: Check processes, services and tasks	check whether apps with root privilege are running.
D7: Check Rooting Traits using shell commands	check rooting traits in a separate process by using shell commands such as `su` and `ps`

original dex file. After obtaining the original dex file, the final step loads the original dex file into the Dalvik Virtual Machine (DVM) and executes it.

Sudipta Ghosh et al. [13] proposed a code obfuscation technique which makes program codes more confusing for reverse engineers on the Android platform. Their technique focuses on increasing the complexity of an app's control flow. This can make it harder to obtain the business logic embedded inside the application. A shortcoming of their work is that experts in reverse engineering can break the proposed obfuscation technique.

Junfeng Xu et al. [14] proposed debug state and debug environment detection for protecting Android applications from dynamic debugging. Debug state detection is detecting debugger such as GDB, IDA and strace by examining parent process, reading process status, or using `ptrace`. Debug environment detection is detecting an emulator environment by testing emulator specific attribute values and features. However, these techniques cannot defend attacks like memory dumping using Lime [15]. Since memory dumping attack necessarily requires root privilege, we need to detect a rooted environment to prevent dynamic reverse engineering.

San-Tsai Sun et al. [16] analyzed several existing rooting methods and presented rooting detection methods based on the trait of rooted Android devices. The rooting detection methods are shown in Table 1. Since these rooting detection methods usually invoke Android APIs, however, rooting detection can be evaded by hooking the APIs using a framework such as Xposed [17]. Hence, we need a technique that can detect evasion attack (method hooking) on Android devices.

3 Protecting Android App against Dynamic Reverse Engineering

Most anti-reverse engineering techniques protect Android applications from dynamic analysis by detecting rooted and/or debugging environments. However, those techniques can be easily evaded by evasion attacks such as method hooking. In this section, we propose an Android application protection scheme against dynamic reverse engineering. This scheme includes rooted/debugging environments detection technique, call stack-based evasion attack (method hooking) detection technique, and dynamic code loading technique. Our scheme is implemented as a stub DEX file and the file is added to APKs (An-

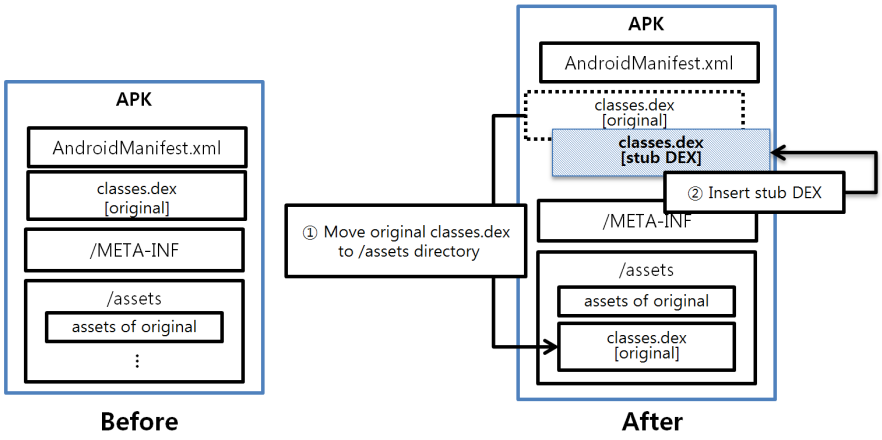

Figure 1: The structure of APK before/after applying our scheme

droid Application Packages), that is, APKs are repackaged. When an application is launched, the stub DEX tries to detect rooted/debugging environments and evasion attacks. If not found, it loads the original DEX file (`classes.dex`) dynamically. Since our scheme is not implemented as a part of application's source codes, it can be applied to APKs that are already packaged.

3.1 Stub DEX and Dynamic Code Loading

The proposed scheme is implemented as an Android executable file, called *stub DEX*. The main tasks of the stub DEX are rooting traits detection, debugging environment detection, call stack analysis and dynamic code loading. The structure of APK before applying our scheme is shown in Figure 1. Usually an APK contains `AndroidManifest.xml` (describing information such as the application's name, version and access rights), `classes.dex` (an executable file), `META-INF` directory and `assets` directory. While applying our scheme, the stub DEX is added to an APK. To execute the stub DEX first when the APK is launched, we set the value of attribute `android:name` of `<application>` element in `AndroidManifest.xml` to the application class of the stub DEX (class `Application` or a subclass of `Application`). The original executable file `classes.dex` is moved to `assets` directory and the stub DEX is named as `classes.dex`. Then, the APK is repackaged and distributed in the market. The structure of the APK after applying our scheme is also shown in Figure 1.

If the repackaged APK is launched in a user's smartphone, the stub DEX is executed first. It performs rooting traits detection and debugging environment detection. In the middle of these tasks, the stub DEX analyzes its call stack to examine whether methods are hooked, i.e. whether any method call other than the stub DEX's methods appears in the call stack. If so, the stub DEX terminates the app assuming it is an evasion attack. If not, the tasks complete and return the results of detection. Depending on the results, the stub DEX terminates the app or loads the original executable file `assets/classes.dex`. If rooting traits or debugging environment is not detected, the stub DEX creates a class loader for the original executable file using `DexClassLoader()`. Then it replaces its class loader with the new class loader using `makeApplication()`. The original executable file invokes `onCreate()` method of its start component. This process is illustrated in Figure 2.

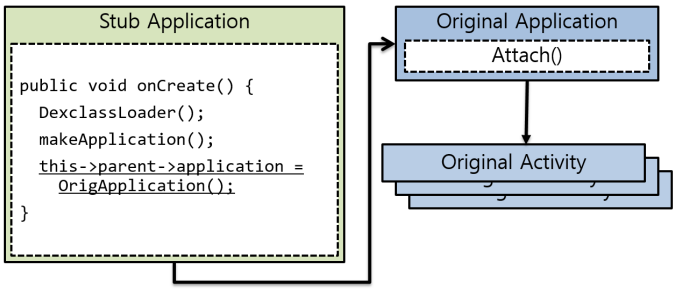

Figure 2: Altering execution flow using dynamic code loading

Table 2: Checklist and stub DEX's methods

Checklist	stub DEX's method
custom image flashing	`detectTestKeys()`
change of system properties	`checkForSystemProperies()`
installed `su` binary	`checkForBinary()`
new commands from `busybox`	`checkForBusybox()`
filesystem attributes	`checkForFilesystem()`
rooting-related apps	`detectRootManagementApps()`
running root process	`detectRootProcess()`

3.2 Rooting Detection Technique

This section presents the design of rooting tratis detection technique. We adopt the detection methods described in [16]. Based on these methods, we produce a checklist of rooting traits and design several functions that check each of them. In our checklist, we refine D2 of [16] into two items: checking `su` binary and checking commands from busybox. And we do not include D7 of [16] in our checklist because D7 can be covered by other methods. We implement the functions utilizing an open source software [18]. The checklist and functions are shown in Table 2.

Method `detectTestKeys()` checks custom image flashing. It reads `/system/build.prop` and examines whether the value of `ro.build.tags` is "release-key". If not, it assumes that the device is rooted. Method `checkForSystemProperties()` checks the value of system properties. It invokes `Runtime().exec("getprop")` to obtain system properties. Then it examines the value of `ro.secure` and `ro.debuggable`. Method `checkForBinary()` tries to find `su` binary file. It invokes `exist()` method to check whether `su` binary is installed in directories such as `/system/(x)bin, /data/local/(x)bin`, ..., etc. Method `checkForBusybox()` checks if additional commands provided by busybox exist or not. It invokes `Runtime.exec()` to execute commands such as `find`, `tail` and `lsof`. Method `checkForFilesystem()` checks filesystem attributes. It invokes `mount` command to examine whether any read-only filesystem has 'write' attribute. Method `detectRootManagementApps()` invokes `getPackageInfo()` to get the list of installed apps, and compares it with a list of well-known rooting-related apps. Method `detectRootProcess()` invokes `Runtime.exec("ps")` to get the list of currently running processes, and checks processes with root privilege. If such a process is not a child of `zygote`, we assume the device is rooted.

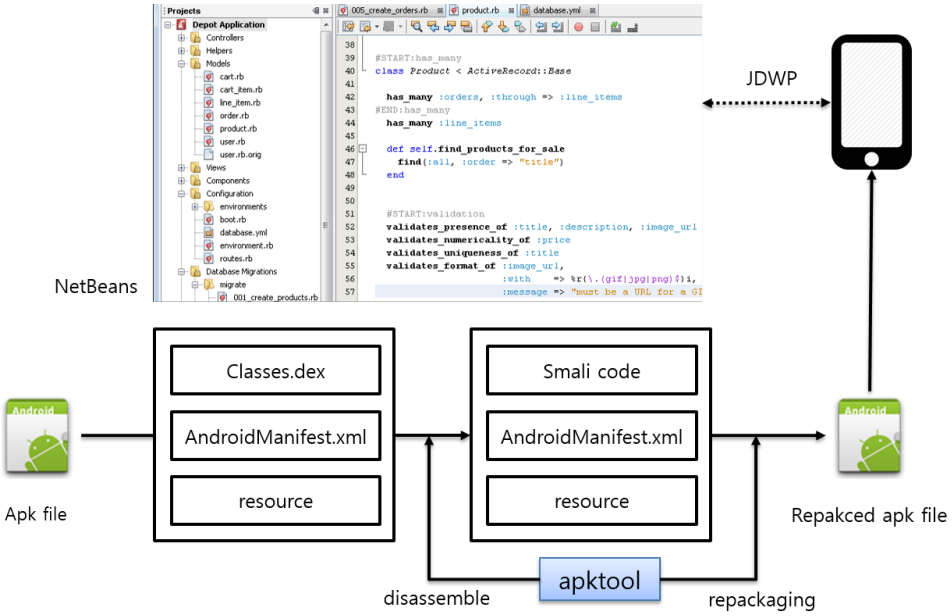

Figure 3: Dynamic debugging of Android app using NetBeans

3.3 Debugging Detection Technique

Android apps in the market are usually built in release mode, and the flag for debugging is removed in release mode. To enable dynamic debugging, attackers need to activate the flag using tools like apktool [19]. The flag is activated by inserting `android:debuggable=true` into `AndroidManifest.xml`. Java-level debugging of Android application is usually conducted using NetBeans [20]. The debugging environment is illustrated in Figure 3.

We can detect a debugging environment by examining the value of the flag. By using an Android service `PackageManager`, the stub DEX obtains `ApplicationInfo` class. `ApplicationInfo.flags` field of the class contains the values of flags of the currently running app. The stub DEX checks whether the field contains FLAG_DEBUGGABLE, which comes from `android:debuggable`. If so, it terminates the app assuming an attacker is debugging. This task is implemented as `isDebuggable()` method.

3.4 Call Stack-based Evasion Attack Detection Technique

A call stack is a memory space for execution of methods of Java applications. When a method is invoked, some memory in the call stack is allocated for the method. The memory usually stores local variables, parameter variables, temporary data of operations and so on. By investigating the call stack, we can identify caller and callee mehods and obtain a chain of method calls. Since Android app is a kind of Java application, we can get the information on method calls of Android apps and notice method hooking (evasion attacks) by comparing method names.

Table 3: Checklist and stub DEX's methods

Checklist	stub DEX's method
flag for debugging	`isDebuggable()`

The stub DEX generates an exception in functions that we want to protect from evasion attacks. We obtain the contents of the call stack by invoking `getStackTrace()` method. `getStackTrace()` returns an array of objects of `StackTraceElement` that contain the class and method names of each called method.

Evasion attacks usually hook system calls or library functions, and modify their parameters or return values to change the execution flow. In order to hook library functions on Android, many attackers employ Xposed [17]. Xposed is a well-known framework which can alter the behavior of Android system or apps. It extends Zygote to load a library for hooking, and every app launched by Zygote inherits the library. The library provides a method, `findAndHookMethod()`, that hooks a specified method and registers callback methods. Everytime the hooked method is called, `handleHookedMethod()` is called. This method invokes registered callback methods, which can change the arguments of the call, invoke other methods and modify the results.

```
[+] call stack : detectResult
skaiser.wrapper.root_detect->call_stack
skaiser.wrapper.root_detect->detectResult
skaiser.wrapper.root_detect->detectRootManagementApps
skaiser.wrapper.root_detect->isRooted
skaiser.wrapper.TestApplication->doRootCheck
skaiser.wrapper.TestApplication->onCreate
```

Figure 4: Call stack of the stub DEX when checking rooting-related apps

Figure 4 shows the call stack of the stub DEX when it invokes `detectRootManagementApps()` to check rooting-related apps. When the app is launched, `onCreate()` is called first. Then it calls `doRootCheck()`, which creates a `root_detect` object. This object calls `isRooted()`, which calls in turn `detectRootManagementApps()`. This method obtains a list of rooting-related apps and calls `detectResult()` to compare it with the apps installed in the device. Note that `call_stack()` is a method that generates an exception object and examines the contents of the call stack contained in it.

```
[+] call stack : detectResult
skaiser.wrapper.root_detect->call_stack
skaiser.wrapper.root_detect->detectResult
skaiser.wrapper.root_detect->detectRootManagementApps
de.robv.android.xposed.XposedBridge->invokeOriginalMethodNative
de.robv.android.xposed.XposedBridge->handleHookedMethod
skaiser.wrapper.root_detect->detectRootManagementApps
skaiser.wrapper.root_detect->isRooted
skaiser.wrapper.TestApplication->doRootCheck
skaiser.wrapper.TestApplication->onCreate
```

Figure 5: Call stack of the stub DEX when `detectRootManagementApp()` is hooked

Figure 5 shows the call stack of the stub DEX when the rooting detection method is hooked. An attacker might hook `detectRootManagementApps()` and modify the list of rooting-related apps. The method calls from `onCreate()` to `detectRootManagementApps()` are the same as in Figure 4. In Figure 5, there are two methods that are not declared in the stub DEX, `handleHookedMethod()` and `invokeOriginalMethodNative()`. These methods belong to Xposed and are able to modify the list of rooting-related apps.

To defend such evasion attacks, the stub DEX investigates its call stack in the middle of rooting/debugging detection methods. By comparing the methods in the call stack with the list of methods declared in the stub DEX (including the methods listed in Table 2 and Table 3), the stub DEX detects method hooking or evasion attack, and terminates the app.

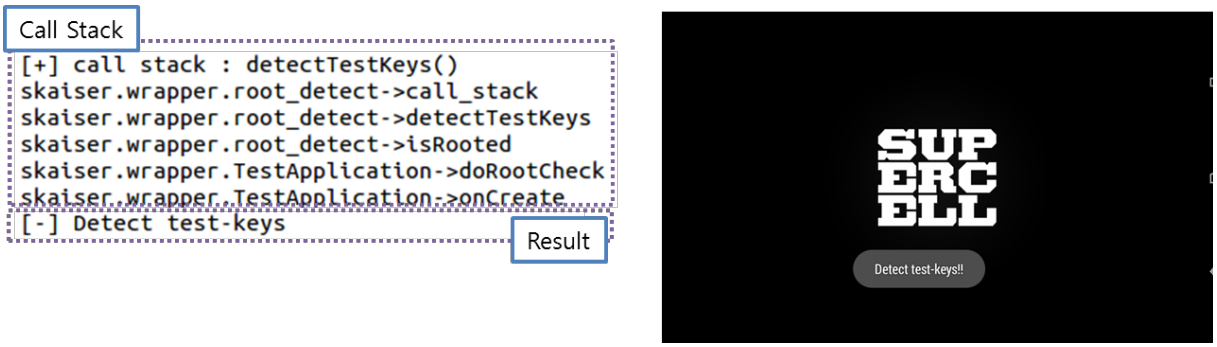

Figure 6: Stack trace and detection result for custom image flashing

4 Experiments

We implement a prototype of the proposed scheme and apply it to an Android application. The test app is 'Clash Of Clans' and is downloaded from Google Play. We add a stub DEX (named as `classes.dex`), move the original `classes.dex` file to directory `assets`, and repackage the APK. The added stub DEX performs rooted/debugging environments detection, evasion (hooking) detection and dynamic loading of the original `classes.dex`.

The experiments are conducted on a real smartphone Nexus 4 (Android 4.4 Kitkat, Linux kernel 3.5 version). We setup a rooted environment for each check list in Table 2 and launch the repackaged test app to examine whether the rooted environment is detected. We also setup a debugging environment as shown in Figure 3 and test whether the debugging environment is detected. Next, we build and install Xposed to the test system to try to evade rooting/debugging detection. By comparing the call stacks of the stub DEX, we show that our scheme can detect evasion attacks.

4.1 Evasion Attacks using Xposed

First, we examine the call stack of the stub DEX on the rooted device with Xposed deactivated. The device is rooted by flashing Cyanogenmod [21] custom image. Figure 6 shows the stack trace of the stub DEX and the detection result when the repackaged test app is launched. Every stack trace of our stub DEX includes method calls `onCreate()`, `doRootCheck()`, and `isRooted()`. When `onCreate()` in our stub DEX is invoked, it calls `doRootCheck()` and `doRootCheck()` creates `root_detect` object. The object contains the methods (listed in Table 2) that detect each rooting or debugging trait. These methods are called one after another by method `isRooted()` of the `root_detect` object. For example, in Figure 6, `isRooted()` calls `detectTestKeys()` to examine the value of `ro.build.tags`. In Figure 6, `detectTestKeys()` detects custom image flashing and outputs the log "[-] Detect test-keys". It also pops a toast message and terminates the app.

Next, we examine the call stack of the stub DEX with Xposed activated. Figure 7 shows the call stack under Xposed's evasion attack. The Xposed's method `handleHookedMethod()` and `invokeOriginal MethodNative()` are in the call stack. These methods modify the return value of `detectTestKeys()`. As a result, `isRooted()` cannot detect custom image flashing. The return value of other rooting trait detection methods is modified in a similar way.

In debugging detection test, we set "`android:debuggable=true`" flag in `AndroidManifest.xml` of the test app and repackage it. We setup a debugging environment as shown in Figure 3 and launch the app. The stub DEX creates `root_detect` object and invokes its method `isDebuggable()`. This method checks the value of `android:debuggable` flag and outputs the log "[-] detect debug mode!!".

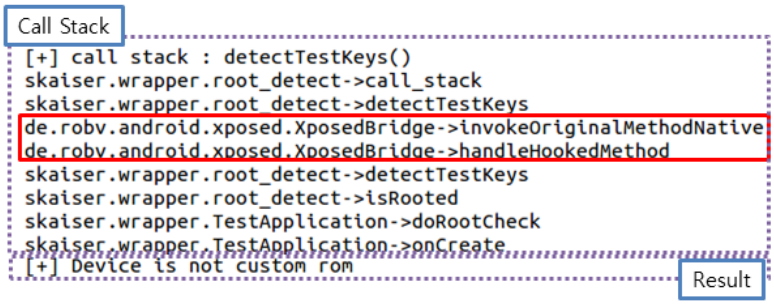

Figure 7: Stack trace when rooting detection is evaded

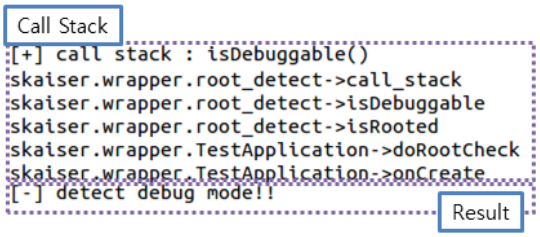

Figure 8: Stack trace when a debugging environment is detected

Figure 8 shows the call stack when a debugging environment is detected.

Figure 9 shows the call stack when debugging detection is evaded using Xposed. Similar to rooting detection evasion, when `isDebuggable()` is called, Xposed's hooking methods are invoked. They modify the return value of `isDebuggable()` and the detection fails. Figure 10 is a screen shot of NetBeans that is debugging the test app.

4.2 Verification of Call Stack based Detection of Evasion Attack

We see that when attackers perform evasion attack using tools like Xposed, the existing rooting and debugging detection techniques do not work as expected. In this section, we verify the effectiveness of the proposed scheme against evasion attacks.

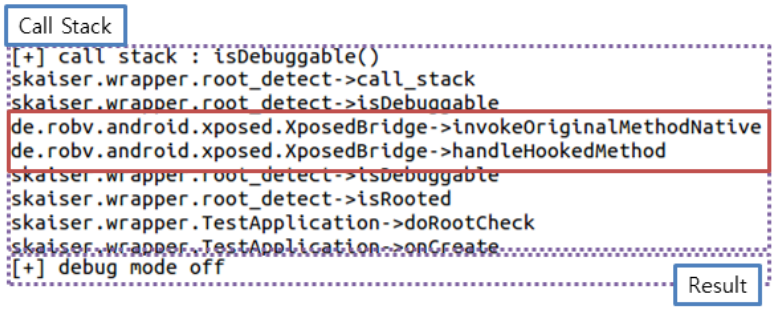

Figure 9: Stack trace when debugging detection is evaded

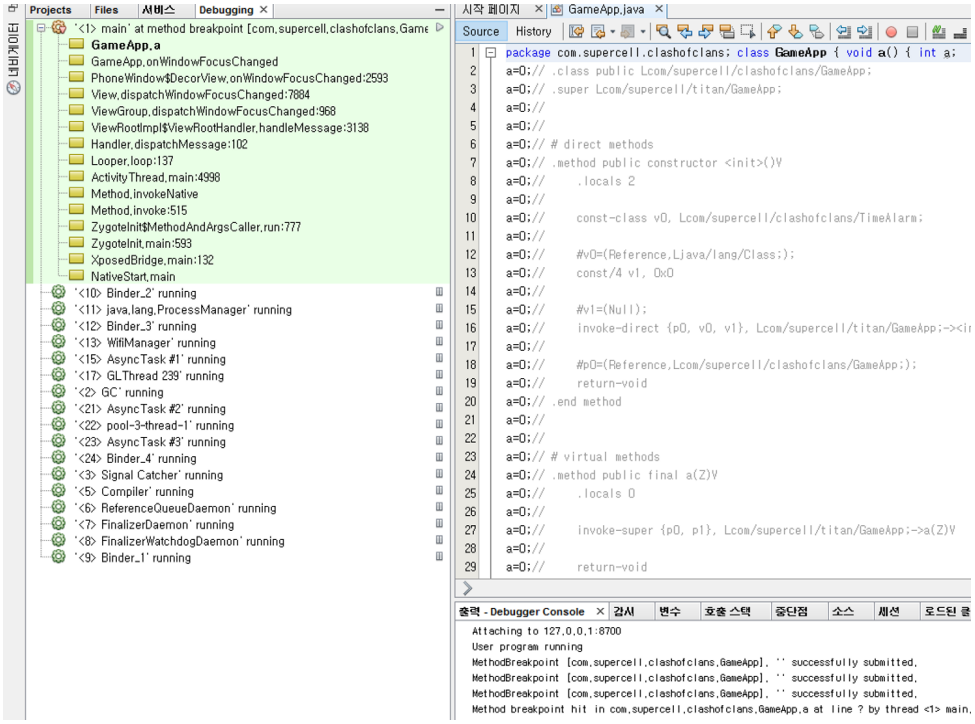

Figure 10: Debugging the test app using NetBeans

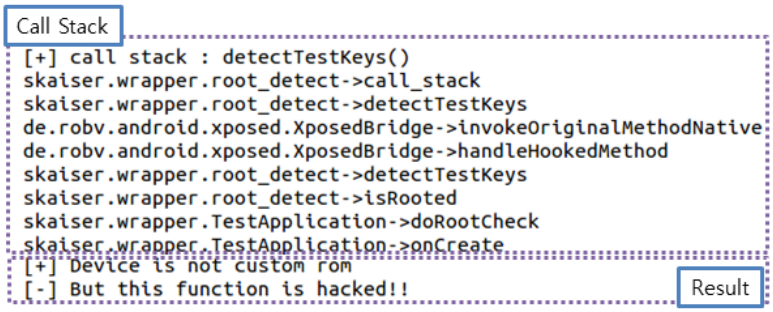

Figure 11: Detection of evasion attack in a rooted environment

Figure 11 shows that the proposed scheme protects the test app against evasion attacks in a rooted environment. Due to the evasion attack, the existing rooting trait detection method (`detectTestKeys()`) fails to detect custom image flashing (Notice the log "[+] Device is not custom rom"). However, the hooking methods of Xposed (`invokeOriginalMethodNative()` and `handleHookedMethod()`) appear in the call stack. Our call stack-based evasion detection technique succeeds in detecting Xposed's method hooking (Notice the log "[-] But this function is hacked!!"). Figure 12 shows similar results of detecting evasion attack that bypasses debugging environment detection. This time, `isDebuggable()` is hooked. The debugging mode detection method fails to detect the debugging flag, but the proposed scheme can find this hooking.

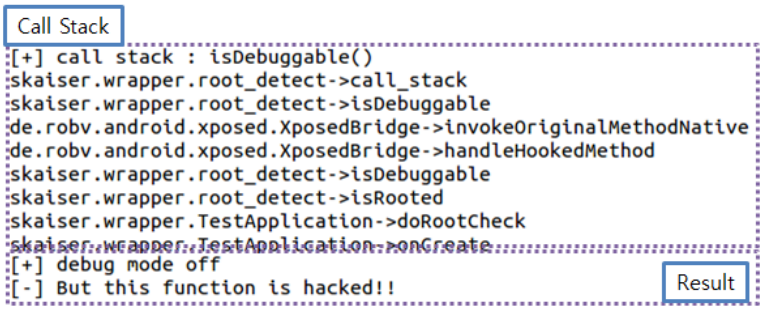

Figure 12: Detection of evasion attack in a debugging environment

4.3 Overhead Measurement

In the proposed scheme, the size of stub DEX is 21.7KB. We measure the time overhead of the proposed scheme. The total overhead is a sum of time for detecting a rooted or debugging environment; determining whether the method is hooked; and loading the original DEX file. The measured total overhead is 1.96 sec.

5 Conclusion

Since Android applications are written in Java programming language and Java bytecodes can be easily decompiled using reverse engineering tools, they are exposed to static and/or dynamic reverse engineering attacks. To protect Android application against static reverse engineering, there are many research work on obfuscation, dynamic code loading, packing, and so on. Recently, however, real device-based dynamic reverse engineering attack is presented that can dump application code and reverse engineer it. This attack necessarily requires root privilege on real devices.

In this paper, we propose an Android application protection scheme. We implement recent rooted/debugging environment detection techniques in our scheme. Since these techniques can be evaded using Xposed framwork, as shown in this paper, we also implement call stack-based evasion detection technique that can detect method hooking enabled by Xposed framework. One of the strong point of our scheme is that it can be applied to any compiled (packaged) Android application by using dynamic code loading technique.

Acknowledgments

This research was supported by (1) the MSIP (Ministry of Science, ICT and Future Planning), Korea, under the ITRC (Information Technology Research Center) support program (IITP-2016-R0992-16-1012) supervised by the IITP (Institute for Information & communications Technology Promotion), and (2) Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT and Future Planning (No. 2015R1A2A1A15053738).

References

[1] S. W. Park and J. H. Yi, “Multiple device login attacks and countermeasures of mobile voip apps on android,” *Journal of Internet Services and Information Security (JISIS)*, vol. 4, no. 4, pp. 115–126, November 2014.

[2] J. Ninglekhu, R. Krishnan, E. John, and M. Panday, "Securing implantable cardioverter defibrillators using smartphones," *Journal of Internet Services and Information Security (JISIS)*, vol. 5, no. 2, pp. 47–64, May 2015.

[3] "State of mobile app security – apps under attack," Research Report: Special Focus on Financial, Retail/Merchant and Healthcare/ Medical Apps, Volume 3, November 2014, https://www.arxan.com/wp-content/uploads/assets1/pdf/State_of_Mobile_App_Security_2014_final.pdf [Online; Accessed on September 10, 2016].

[4] W. Zhou, Y. Zhou, X. Jiang, and P. Ning, "Detecting repackaged smartphone applications in third-party android marketplaces," in *Proc. of the 2th ACM conference on Data and Application Security and Privacy (CODASPY'12), San Antonio, Texas, USA*. ACM, February 2012, pp. 317–326.

[5] E. Lafortune *et al.*, "Proguard," 2004, http://proguard.sourceforge.net [Online; Accessed on September 10, 2016].

[6] P. Schulz, "Code protection in android," June 2012, http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.469.6113&rep=rep1&type=pdf [Online; Accessed on September 10, 2016].

[7] "Bangcle," http://www.bangcle.com [Online; Accessed on September 10, 2016].

[8] "Liapp," https://liapp.lockincomp.com/ [Online; Accessed on September 10, 2016].

[9] D. Kim, J. Kwak, and J. Ryou, "Dwroiddump: Executable code extraction from android applications for malware analysis," *International Journal of Distributed Sensor Networks*, vol. 2015, February 2015.

[10] T. Petsas, G. Voyatzis, E. Athanasopoulos, M. Polychronakis, and S. Ioannidis, "Rage against the virtual machine: hindering dynamic analysis of android malware," in *Proc. of the 7th European Workshop on System Security (EuroSec'14), Amsterdam, Netherlands*. ACM, April 2014, pp. 5:1–5:6.

[11] T. Strazzere, "Dex education 201: anti-emulation," 2013.

[12] W. Yang, Y. Zhang, J. Li, J. Shu, B. Li, W. Hu, and D. Gu, "Appspear: Bytecode decrypting and dex reassembling for packed android malware," in *Proc. of the 18th International Symposium on Research in Attacks, Intrusions, and Defenses (RAID'15), Kyoto, Japan*, ser. Lecture Notes in Computer Science, vol. 9404. Springer International Publishing, December 2015, pp. 359–381.

[13] S. Ghosh, S. Tandan, and K. Lahre, "Shielding android application against reverse engineering," *International Journal of Engineering Research and Technology*, vol. 2, no. 6, pp. 2635–2643, June 2013.

[14] J. Xu, L. Zhang, Y. Sun, D. Lin, and Y. Mao, "Toward a secure android software protection system," in *Proc. of the 2015 IEEE International Conference on Computer and Information Technology; Ubiquitous Computing and Communications; Dependable, Autonomic and Secure Computing; Pervasive Intelligence and Computing (CIT'15/IUCC'15/DASC'15/PICOM'15), Liverpool, United Kingdom*. IEEE, October 2015, pp. 2068–2074.

[15] "Lime," https://github.com/504ensicsLabs/LiME/tree/master/doc [Online; Accessed on September 10, 2016].

[16] S.-T. Sun, A. Cuadros, and K. Beznosov, "Android rooting: Methods, detection, and evasion," in *Proc. of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM'15), Denver, Colorado, USA*. ACM, October 2015, pp. 3–14.

[17] Rovo89, "[framework only!] xposed - rom modding without modifying apks (2.6.1)," 2012, http://forum.xda-developers.com/xposed/frameworkxposed-rom-modding-modifying-t1574401 [Online; Accessed on September 10, 2016].

[18] M. Scott, "Rootbeer," https://github.com/scottyab/rootbeer [Online; Accessed on September 10, 2016].

[19] "Apktool," http://ibotpeaches.github.io/Apktool/ [Online; Accessed on September 10, 2016].

[20] "Netbeans," https://netbeans.org/ [Online; Accessed on September 10, 2016].

[21] "Cyanogenmod," https://download.cyanogenmod.org [Online; Accessed on September 10, 2016].

Author Biography

Kyeonghwan Lim received the B.E. degree in Dept. of Software Science from Dankook University, Korea, in 2015. He is currently a master student at Dept. of Computer Science and Engineering in Dankook University, Korea. His research interests include computer system security, mobile security.

Younsik Jeong received the B.E. degree in computer engineering from Dankook University, Korea, in 2012 and the M.E. degree in computer science and engineering from the Dankook University, Korea, in 2013. He is a Ph.D. student in Computer science and engineering at the Dankook University. His current research interests include computer security and mobile security.

Seong-je Cho received the B.E., the M.E. and the Ph.D. in Computer Engineering from Seoul National University in 1989, 1991 and 1996 respectively. He joined the faculty of Dankook University, Korea in 1997. He was a visiting scholar at Department of EECS, University of California, Irvine, USA in 2001, and at Department of Electrical and Computer Engineering, University of Cincinnati, USA in 2009 respectively. He is a Professor in Department of Computer Science and Engineering (Graduate school) and Department of Software Science (Undergraduate school), Dankook University, Korea. His current research interests include computer security, smartphone security, operating systems, and software protection.

Minkyu Park received the B.E., M.E., and Ph.D. degree in Computer Engineering from Seoul National University in 1991, 1993, and 2005, respectively. He is now a professor in Konkuk University, Rep. of Korea. His research interests include operating systems, real-time scheduling, embedded software, computer system security, and HCI. He has authored and co-authored several journals and conference papers.

Sangchul Han received his B.S. degree in Computer Science from Yonsei University in 1998. He received his M.E. and Ph.D. degrees in Computer Engineering from Seoul National University in 2000 and 2007, respectively. He is now an associate professor of Department of Computer Engineering at Konkuk University. His research interests include real-time scheduling, software protection, and computer security.