J Supercomput (2016) 72:232–246
DOI 10.1007/s11227-015-1559-9

Anti-debugging scheme for protecting mobile apps on android platform

Haehyun Cho[1] · Jongsu Lim[1] · Hyunki Kim[1] · Jeong Hyun Yi[1]

Published online: 6 November 2015

Abstract The Android application package file, APK file, can be easily decompiled using Android reverse engineering tools. Thus, general apps can be easily transformed into malicious application through reverse engineering and analysis. These repacked apps could be uploaded in general android app market called Google Play Store and redistributed. To prevent theses malicious behaviors such as malicious code injection or code falsifications, many techniques and tools were developed. However, these techniques also can be analyzed using debuggers. Also, analyzed apps can be tampered easily. For example, when applying anti-analysis techniques to android apps using Dexprotector which is commercial tool for protecting android app, it can be seen that these techniques can also be analyzed using debugger. In this paper, to protect the android app from the attack using debugger, we propose anti-debugging techniques for native code debugging and managed code debugging of android apps.

Keywords Anti-reversing · Android APP protection · Detecting emulator · Anti-debugging

✉ Jeong Hyun Yi
jhyi@ssu.ac.kr

Haehyun Cho
haehyuncho@gmail.com

Jongsu Lim
jongsu253@gmail.com

Hyunki Kim
hitechnet92@gmail.com

[1] School of Computer Science and Engineering, Soongsil University, Seoul 156-743, Korea

1 Introduction

With the development of the smartphone market, the smartphone app market has also been marked with huge growth. However, the increased use of smart phone apps has also seen an exponential increase in malicious activity targeting smartphone apps. The result has been a myriad of things such as the spread of malicious code, leakage of personal information, heavy financial loss, etc.

The list of these security risks for Android apps are never ending, because of Android app's structural characteristics [7,14]. Android apps are fundamentally built with Java and formed in the APK (Android application package) file structure which is a self-signed app. The binary compiled with Java can be easily restored into the original source or similar to it, with a Java decompiler. Android apps are characterized by the easily decompiled Java language and by the easily extracted apps' APK files installed on an Android phone. Because of these characteristics, an attacker can easily extract the APK file of a normal app and inject an attack code so that, after repackaging it, the app is disguised as a brand new app and distributed. If a user installs and runs the tampered app, thinking it as a normal app, the malicious codes planted by the attacker can be executed.

Because Android apps are easy to reverse engineering and analyze, it can be easily tampered into malicious apps. Therefore, many methods to prevent tampering has been proposed. Among the proposed methods are codes obfuscation, execution code compression, control flow obfuscation, etc., [16,18]. However, there is the caveat that even apps that apply these methods still remain vulnerable to analysis with the use of a debugger. Therefore, in this paper targeted debugger is the IDA [11] which is the famous commercial debugger, we analyze the way the debugger operates for both native and managed code. In addition, using the results of the analysis, explain the methods of thwarting the debugger.

2 Backgrounds

In this section, we will examine how the debuggers used to analyze Android apps operate.

2.1 Analysis of the native code debugger operational method

Figure 1 shows the process of analyzing the Android app Native code. Remote debugging support tool must be running inside the Android device, because debugging is done by communicating with the tool. To run the remote debugging support tool in the Android device, one needs to use Android Debug Bridge (ADB) [1], a tool that enables the host computer to communicate with the Android device. ADB communicates with Android Debug Bridge Daemon (ADBD) which operates within the device to provide services such as debugging information, and remote shell. But to run the remote debugging support tool within the Android device, remote shell needs to be used. When running the remote debugging support tool using remote shell, the remote debugging support tool must be run with root authority. This is because in the case of

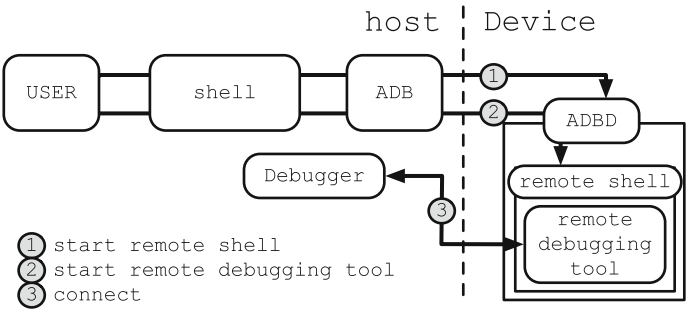

Fig. 1 The execution process of debugger and remote debugging support tools

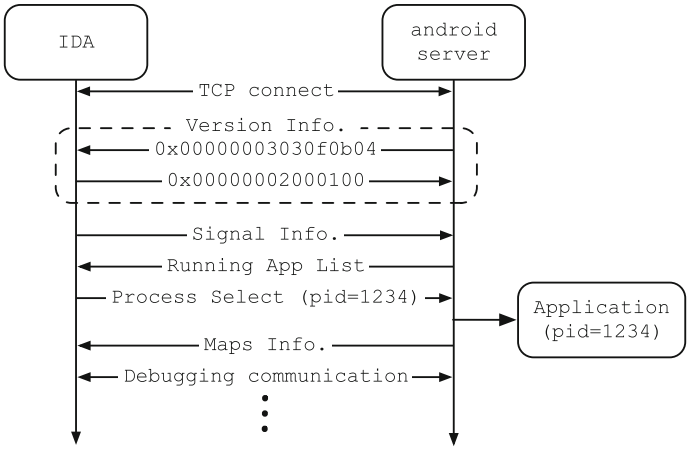

Fig. 2 The IDA and android_server communication process

Native debugging, the target is a process; therefore, the debugger's authority must be higher than the debuggee process's authority.

Figure 2 is the communication process of using debugger IDA to analyze the app's native code and the android server which is a remote debugging support tool, to change the process into a debuggee process. After IDA and android_server establish TCP connection (three-way handshake), verify the version information. In addition, IDA sends android_server signals occurred by the debuggee process which are needed to be verified by the debugger and receives a list of processes available for debugging from the android_server. Also, if the PID of the process targeted for debugging is sent to android_server, android_server will set up the process with the corresponding PID into a debugging process and send the binary code and maps information of the corresponding process to IDA.

2.2 Analysis of the JDWP debugger operational method

Dalvik virtual machine translates and executes the app's Dalvik byte code included in Dalvik Executable (DEX) file [5] which is an Android app execution file

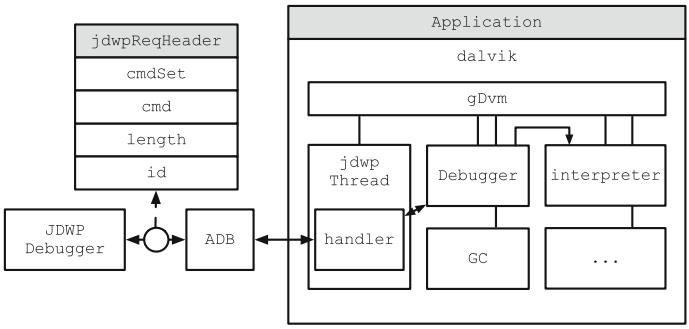

Fig. 3 The operational process of the JDWP debugger

format. The objective of Dalvik virtual machine being to run Android apps, it provides features related to execution such as interpreter and garbage collector [3,10]. Also, debugging feature is related to app execution, it is supported by the Dalvik virtual machine. Debugging of an Android app, as shown in Fig. 3, is done in a remote debugging environment. Dalvik virtual machine and debugger cannot communicate directly with each other but must communicate using ADB.

JDWP, an abbreviation for Java Debug Wire Protocol [12] and used during the ADB communication process, is a designated protocol for communication between the debugger and virtual machine in Java Platform Debug Architecture (JPDA) [13]. It is normally used in an environment for debugging Java applications but, because Dalvik virtual machine also supports JDWP, it is used in the communication process.

To deal with JDWP, Dalvik virtual machine provides a JDWP thread [19]. When the JDWP debugger sends a debugging command through JDWP protocol, the JDWP thread interprets JDWP protocol and executes the handler corresponding to the `cmdSet` and `cmd`. Android app debugging takes place following a process of the handler executing the debugging function and sending the executed results back to the JDWP thread through JDWP protocol.

The DvmGlobals structure was defined for the debugger to easily check the condition of the virtual machine, but the DvmGlobals structure stores not only information on the status of the virtual machine but also information produced by the debugger. The DvmGlobals structure is generated into a global variable `gDvm`, and the handler executed by the JDWP thread also verifies or alters the `gDvm` value to have debugging proceed.

3 Anti-debugging

In this section, based on the debugging process analyzed in the above Sect. 2, we explain debugger prevention techniques [4,9,17].

3.1 Native code debugger prevention method

Debugger detection method through hash value comparison If the debugger put into place a break pointer in the debuggee process, the code of the debuggee process that includes a break pointer will be altered. But when the code section of the procedure is running normally, due to it not being an altered section, the same hash value will always be generated when you find the hash value for the code section. However, because code is altered if you implement a break pointer, a different hash value is generated when finding the hash value using the same hash algorithm as the situation mentioned before. As a result, debuggers can be detected by checking to see if the hash value generated by the code section differs from the existing hash value.

Debugger detection method using timing check The timing check method is a detection method based on program runtime. It is common for code to be executed in a single step method when executing code in the debugger but, with the single step method, a wait time develops. By perceiving this wait time, we can detect the debugger. However, because code is not always executed in a single step method when code is executed in the debugger, this method must be used only after anticipating the wait time and the location conducive for executing code to use this method.

Debugger detection method using signals All signals of the main process in debugging are first sent to the debugger, and transmission of the main debugging process signal is confirmed in the debugger. Because the process is terminated when a signal such as `SIGKILL` is transmitted to the process, signals such as `SIGKILL` will not be sent to the process while debugging. Thus, after generating signals such as `SIGINT` and confirming their transmission to the app, we can detect whether the debugging is operating or not.

Debugger detection method using TracerPID If a process is attached to the debugger process, the TracerPID status value is not 0, but the PID value of the debugger process which is attached to it. The TracerPID status value can be read from the `/proc/PID/Status` file; therefore, the debugger can be detected by determining whether the TracerPID status value in `/proc/PID/status` file is zero or not.

3.2 JDWP debugger prevention method

Dalvik virtual machine has been optimized to generate multiple virtual machine instances with minimal memory. The reasons for generating multiple instances are for it providing an advantage in protection and greater efficiency in execution [15]. In the case of the current standard Java runtime system, when a crash occurs in the virtual machine, either due to malicious intent or due to error, all applications run by the virtual machine are affected. However, for the Dalvik virtual machine, only the applications run by the virtual machine that had a crash are affected. Additionally, in the case of Dalvik virtual machine, it only needs to deal with the information related to the application which increases the efficiency in execution (Figs. 4, 5, 6).

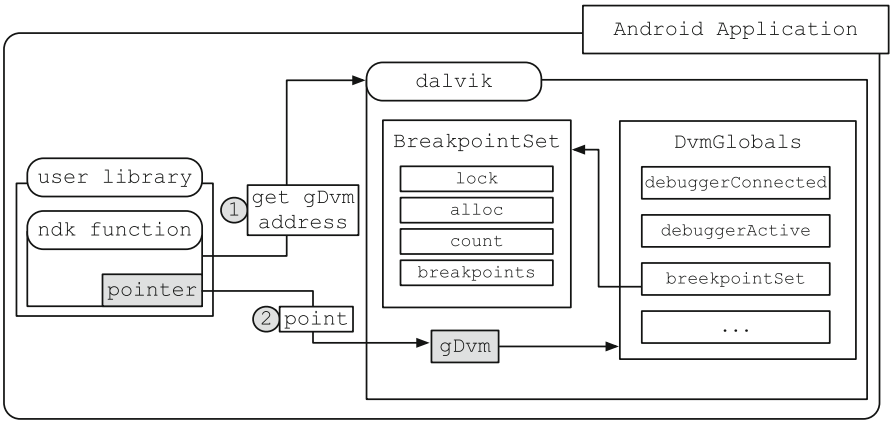

Fig. 4 User library can access libdvm.so

Fig. 5 The value that stores information on the debugger status

```
struct DvmGlobals {
    ...
    bool debuggerConnected;
    bool debuggerActive;
    ...
};
```

Fig. 6 BreakpointSet structure

```
struct BreakpointSet {
    pthread_mutex_t lock;
    int          alloc;
    int          count;
    Breakpoint* breakpoints;
};
```

Because of the abovementioned characteristics, the Dalvik virtual machine instance is included in library format for each Android app process. Because the Dalvik virtual machine instance is generated by an Android app process, even the library code written with NDK by the user can easily access the memory area used by the Dalvik virtual machine instance. This is because the two libraries both exist in the memory area of the same process.

The Dalvik virtual machine instance includes `gDvm`, which was explained while analyzing the debugging process of JDWP debugger, and, by referring to and altering the status value stored in the `gDvm`, JDWP debugger can be prevented [2]. So, in this section we will explain the method of preventing the debugger by accessing the Dalvik virtual machine instance included in the Android app process.

Detection method using the debugger status value The `debuggerConnected` member variable and `debuggerActive` member variable store the debugger execution status in the DvmGlobals structure. If the debugger is connected to the JDWP thread, the `debuggerConnected` variable is set to 1, and if the debugging function

is running, the `debuggerActive` variable is set to 1. By checking if the variables for `debuggerConnected` and `debuggerActive` are set to `1`, we can determine whether the debugger is connected or running.

Detection method using the number of breakpoints To control the execution flow, a debugger provides the function of setting a breakpoint in the execution code.

Information regarding the set breakpoint is stored in the `BreakpointSet` structure. The `BreakpointSet` structure stores the number of breakpoints set, the address of the locations where the breakpoints have been set, the original execution code, etc.

The number of set breakpoints is incorporated in the count member of the `BreakpointSet` structure and, by checking the count value, the debugger can be detected.

Tamper detecting using the breakpointset structure The `BreakpointSet` structure is dynamically allocated in the Dalvik virtual machine. Thus, since it is accessing a structure member using the pointer, if the `NULL` pointer is allocated to `BreakpointSet`, the process can be terminated due to erroneous memory access when setting the breakpoint, and the debugger can be thwarted.

4 Reverse engineering using debugger

In this section, we apply prevention methods on Android app to deal with reverse engineering using Dexprotector [6], a tool that applies prevention methods, and refer to the necessity of debugger prevention method, through reverse engineering using debugger.

DexProtector is a tool used to prevent Android reverse engineering. The reverse engineering prevention method used by DexProtector includes using dynamic key, automatic application of the tamper detection function when applying an encryption method, concealment of the tamper detection routine, concealment of the obfuscation routine, concealment of the encrypted data, running a tamper detection routine while the app is loading, etc. DexProtector's methods are applied as shown in Fig. 7.

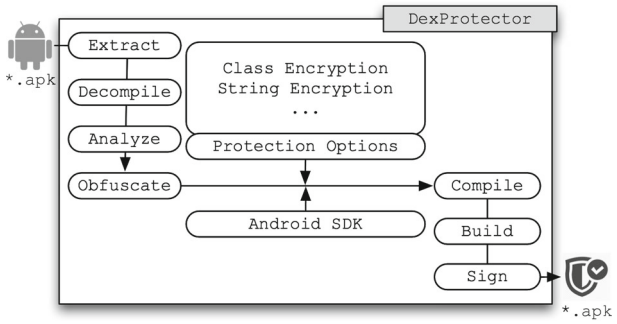

Fig. 7 The process of applying DexProtector to an android app

```
D/dalvikvm( 1459): DexOpt: --- BEGIN 'new.apk' (bootstrap=0) ---
D/dalvikvm( 1471): DexOpt: load 10ms, verify+opt 6ms, 103460 bytes
D/dalvikvm( 1459): DexOpt: --- END 'new.apk' (success) ---
D/dalvikvm( 1459): DEX prep '/data/data/com.example.msec/app_dex/new.apk': unzip
 in 0ms, rewrite 171ms
D/dalvikvm( 1459): DexOpt: --- BEGIN 'new.apk' (bootstrap=0) ---
D/dalvikvm( 1473): DexOpt: load 3ms, verify+opt 2ms, 77108 bytes
D/dalvikvm( 1459): DexOpt: --- END 'new.apk' (success) ---
D/dalvikvm( 1459): DEX prep '/data/data/com.example.msec/app_dex/new.apk': unzip
 in 0ms, rewrite 84ms
```

Fig. 8 The running log of an app that applies DexProtector

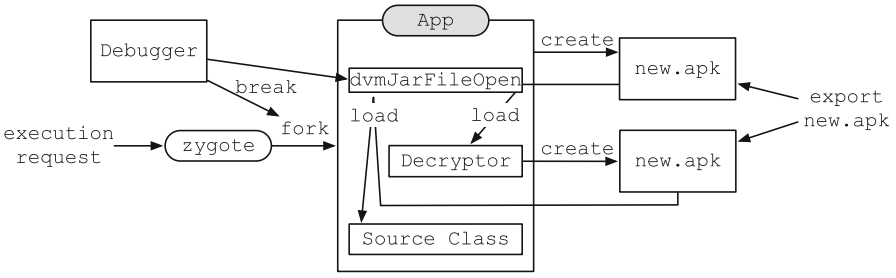

Fig. 9 The process of reverse engineering an app that applies DexProtector

Figure 8, showing the log that results from running an app with DexProtector, is the log that results when loading the file containing the Dalvik byte code onto memory in the Dalvik virtual machine. Thus, we can see that the `new.apk` file that contains the Dalvik byte code is loaded twice onto memory and then used. However, because the `new.apk` file is deleted after it is used, we cannot confirm the code.

The way to acquire a `new.apk` file is to extract it before it gets deleted. Attempts at repackaging proved that one cannot acquire the `new.apk` file when modifying the original app when the tamper detection routine is applied. Also, in the case of the JDWP debugger, because the JDWP debugger prevention method is applied, it cannot be used. While debugging is possible with a native debugger, due to the differences in the execution of debuggers, the `new.apk` file cannot be extracted.

The reason for this is because DexProtector uses application class which bypasses the functions applied for preventing reverse engineering and runs the original app. Application class is the very first class called when the Android app first opens. Resultantly, the native debugger that attempts debugging an app that is running cannot debug the application class.

Figure 9 is the process of acquiring the `new.apk` from an app that uses class encryption, one of the app protection functions of DexProtector.

Characteristic of the Android platform, Android apps run by forking the Zygote process [8]. The Zygote process is a process that loads the class, resource, and library used in the app beforehand to have Android apps run faster. Therefore, Zygote process is forked and only replaces the process image, when a Android app runs. This characteristic is used to debug the Zygote process, when the Android app process forks, debugging is possible.

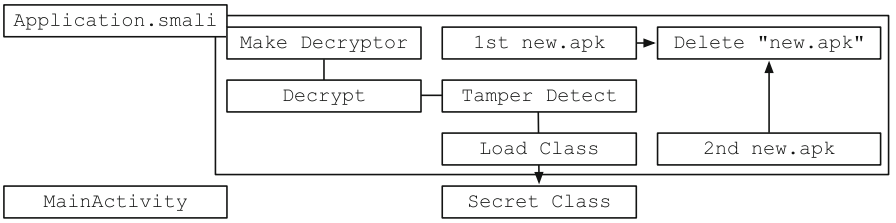

Fig. 10 The restoration process of the original code of an app that applies DexProtector

By loading the file that contains Dalvik byte code after the Android app process forks, the process image is replaced. After the image is replaced, the first thing that is executed is application class. As a result, the execution process of the application class can be analyzed.

In the application class, you can see that the `new.apk` file is used by loading it twice on memory However, to load the file containing Dalvik byte code onto memory, you must call `dvmJarFileOpen`, a function provided by the Dalvik virtual machine. Thus, by setting a breakpoint in the `dvmJarFileOpen` function, the `new.apk` file, generated before the `dvmJarFileOpen` function is executed, can be extracted

Figure 10 shows the restoration process of the original code discovered through reverse engineering of an app that applies DexProtector. First, the app with DexProtector combines the byte codes to create the decryptor into a `new.apk` file. In addition, the generated decryptor is loaded onto memory and decrypts the encrypted original code `classes.dex` stored in the asset folder. The decrypted original code is generated into a `new.apk` file. Also, by loading the decrypted original code into the memory, the original app is run and the recently created `new.apk` file is deleted.

5 Proposed scheme

5.1 Preemptive virtual connection and attachment

A characteristic of the debugging process of the native code is that the program that executes the actual debugging function and the program that provides the interface concerning debugging are divided. Therefore, the two programs need to communicate with each other, and a break in communication disallows debugging from proceeding (Figs. 11, 12).

Incorporating the information mentioned above, we attempted to establish a virtual connection utilizing the protocol used in communication between the remote debugging support tool android_server and the user interface tool IDA. After establishing TCP connection between the Android app and android_server, by sending the version information message `(0x00000003030f 0b04)` from android_server and sending the reply message `(0x0000000200 0100)`, the virtual connection status of the Android app and android_server can be maintained. Because of this, the actual IDA cannot communicate with android_server and, as a result, the debugging process cannot be carried out. Thus, by virtually connecting the Android app with android_server, the native debugger can be prevented.

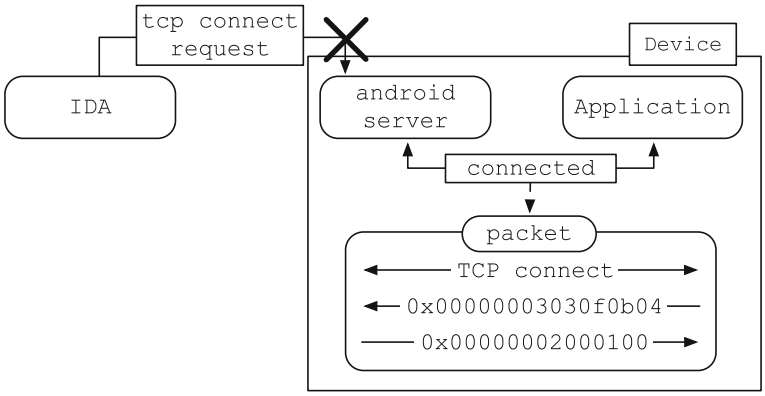

Fig. 11 Prevention of IDA connection using virtual connection

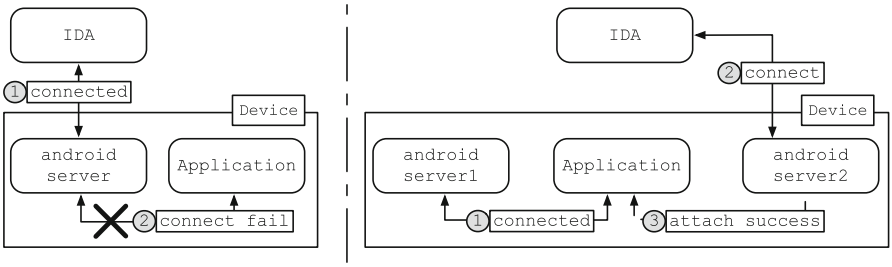

Fig. 12 The unusual situation that can arise during the virtual connection

However, the method that virtually connects the Android app with android_server is only a temporary measure. If the IDA and android_server are connected first, conversely the Android app cannot communicate with android_server. Also, even if the Android app connected first with android_server, this cannot prevent debugging attempts through communication with a different android_server. The reason being that, in the case of virtual connection, it establishes communication between the Android app and android_server process for the purpose of preventing communication between the android_server process and IDA.

Consequently, to protect apps from the above mentioned method, we induce the android_server connected to the Android app to debug the Android app. In other words, as in Fig. 13, the Android app sends the debugging command to android_server, and android_server becomes a process that debugs the Android app. Because the Android app and android_server are connected, IDA cannot connect with android_server. Also, because the android_server connected to the Android app is debugging the Android app, even with the use of a different android_server process, the Android app cannot be debugged.

Nevertheless, the method used above, also, cannot be used if an android_ server connected with IDA is debugging the Android app. This is the reason why the method proposed in this paper is like Fig. 14.

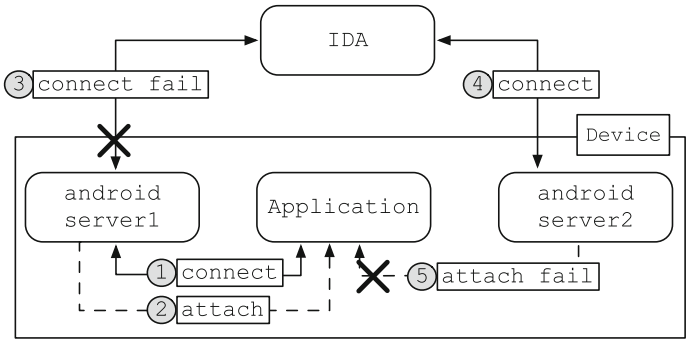

Fig. 13 Debugger prevention through preemptive attach

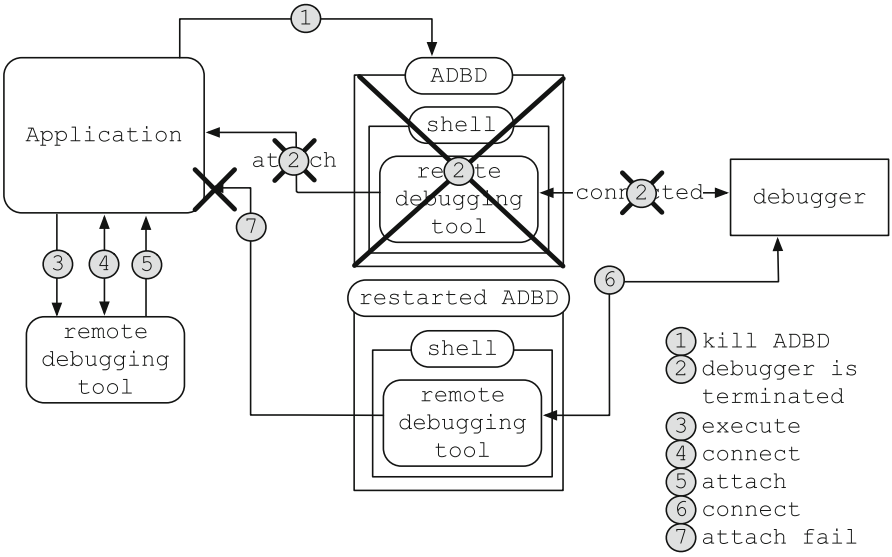

Fig. 14 Debugging prevention through preemptive attach

For special purpose programs like the debugger in Android to be executed within the Android device, as characteristic of the Android environment, the shell process provided by the abdb process is used. As a result, in the case that android_server is executed, android_server is the child process of shell and shell process becomes the child process of abdb. Therefore, through the termination of shell process or abdb process, android_server will be terminated. In addition, for android_server to debug the process, it must be executed with root authority. The su command that changes UID is not provided in the typical Android environment, but in the Android environment that runs android_server, the su command has to be provided. The su command provided in this environment can acquire root authority just by calling it. Thus, it is possible for an Android app to acquire root authority.

Using these characteristics, even if an android_server connected with IDA debugs the Android app, the Android app can acquire root authority with the su command and

terminate the abdb process or shell process to terminate the android_server connected with IDA. Using this process, we gain an opportunity for the android_server executed by the Android app to debug the Android app.

As a result, to protect the native code from debuggers, we terminate the adbd process to kill the shell and the shell's child process which is the child processes of adbd process. After, executing the android_server included in the app and connecting it with the app, the android_server is used for debugging.

5.2 Hooking communication function

The detection methods used in the existing JDWP debugging prevention method have the drawback that they must be executed while the debugger is connected to function. Furthermore, because the prevention methods facilitate access for the section that accesses memory with the `NULL` pointer to prevent errors induced by faulty memory access, an error message for Segmentation Fault is sent to the user.

Therefore, in this section we propose a method that, by hooking the function called to process the message that JDWP thread sends to JDWP protocol, terminates the Android app process when the JDWP thread is run (Figs. 15, 16).

The JDWP thread, to process communication with the debugger, uses two methods: ADB communication method and socket communication method. These two communication methods share the same procedures that facilitate communication such as connection with the debugger, data transmission/reception, transmitted/received message translation, etc. However, because the two methods differ in the way they process the procedures, an interface for the function that processes the communication procedure is provided using `JdwpTransport` structure.

While the `JdwpTransport` structure provides a function pointer for the function that processes the communication procedure, in the JDWP thread, when the ADB communication method is used, the designated function to process the ADB communication method is registered in the `JdwpTransport` structure, and when the socket communication method is used, the function defined for processing the socket

```
struct JdwpTransport {
    bool (*startup)(struct JdwpState* state, const
      JdwpStartupParams* pParams);
    bool (*accept)(struct JdwpState* state);
    bool (*establish)(struct JdwpState* state);
    void (*close)(struct JdwpState* state);
    void (*shutdown)(struct JdwpState* state);
    void (*free)(struct JdwpState* state);
    bool (*isConnected)(struct JdwpState* state);
    bool (*awaitingHandshake)(struct JdwpState* state);
    bool (*processIncoming)(struct JdwpState* state);
    bool (*sendRequest)(struct JdwpState* state, ExpandBuf* pReq);
    bool (*sendBufferedRequest)(struct JdwpState* state,
        const struct iovec* iov, int iovcnt);
};
```

Fig. 15 JdwpTransport structure

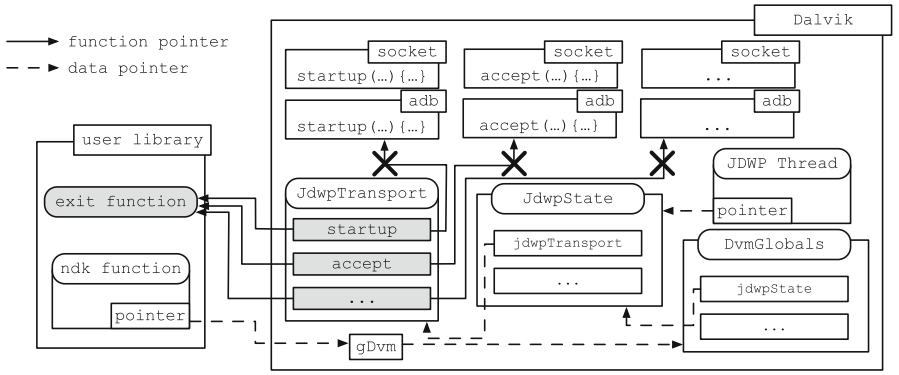

Fig. 16 JdwpTransport structure hooking process

method is registered in the `JdwpTransport` structure. Furthermore, to process the communication procedure, the JDWP thread calls the function pointer registered in the `JdwpTransport` structure.

Because the explained `JdwpTransport` structure uses the function pointer, a problem arises where the user-defined function is called when the JDWP thread processes the communication procedure if the user-defined function is registered in the `JdwpTransport` structure.

If, using the fact that the user-defined function can be registered in the `JdwpTransport` structure, a function that includes a code that terminates process is registered in the `JdwpTransport` structure, when the JDWP thread processes communication, instead of the function to process the existing communication, the user-defined function is called. Thus, if the user-defined function registered in the `JdwpTransport` structure accepts function pointer, during the procedure where the JDWP thread accepts from the debugger, the function registered by the user is called. At this time, if a termination code is included in the function registered by the user, the Android app will be terminated.

The method of registering in the `JdwpTransport` structure the function containing the termination code has the benefit that, unlike the currently used methods, it runs a normal termination code to terminate the process so that an error message is not sent to the user.

6 Experimental result

These are the test results from building an application program that applies the proposed method and running tests on it. Also, these experiments are conducted with IDA Pro 6.7.

Figure 17 is an attempt to, after running an application program that applies the proposed Preemptive Virtual Connection method, connect with the remote debugging support tool using IDA as the user interface tool. We can see that a warning message appeared.

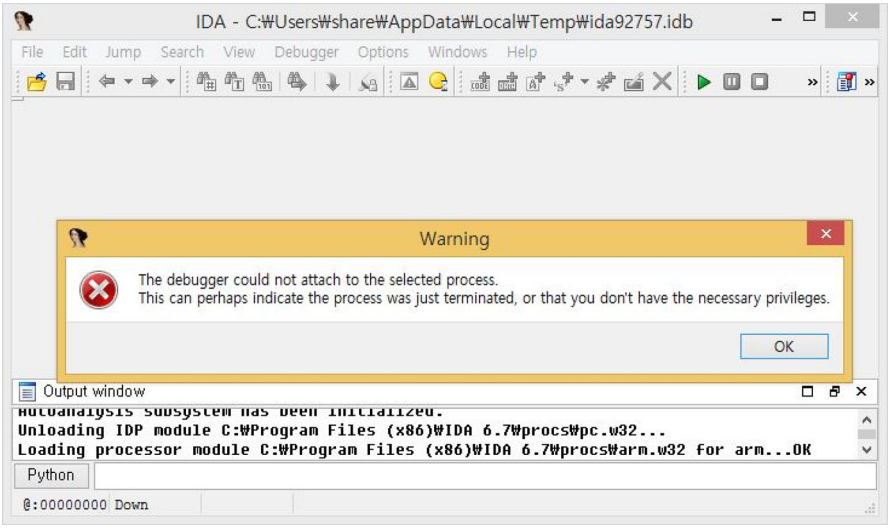

Fig. 17 Result from proposed IDA attachment prevention scheme

```
V/HOOKLOG ( 1505): I'm calling the exit(0)
I/WindowState(  249): WIN DEATH: Window{41e25db8 com.example.dalvik/com.example.
dalvik.MainActivity paused=false}
I/ActivityManager(  249): Process com.example.dalvik (pid 1505) has died.
W/ActivityManager(  249): Force removing ActivityRecord{41bc6388 com.example.dal
vik/.MainActivity}: app died, no saved state
D/Zygote  (   89): Process 1505 exited cleanly (255)
```

Fig. 18 Result from proposed hooking communication function scheme

Figure 18 is the resulting log of an attempt, after running the application program that applies the proposed Hooking Communication Function method, to establish JDWP debugging connecting using IDA as the user interface tool. `I'm calling the exit(0)` is a message that alerts that the hooked function has been called. We can see that no error messages have appeared and that the program has terminated normally.

7 Conclusion

In this paper, we proposed debugging prevention methods to prevent reverse engineering on android apps. To prevent Native debugger, we connect remote debugging support tool with the target app, then the tool preemptively connects with the target app, therefore, preventing the native debugger by occupying the target app first. Through experiments, we verified the feasibility of the proposed scheme.

While analyzing the debugging process of the Dalvik virtual machine, we realized that the function pointer that has been exposed to the global variable is used in the

debugging process. Thus, by having the method proposed to prevent against the JDWP debugger hook the function pointer to cause the application program to be terminated if debugging occurs, we ensured the validity of the method we proposed.

Acknowledgments This research was supported by Global Research Laboratory (GRL) program through the National Research Foundation of Korea (NRF-2014K1A1A2043029).

References

1. Android debug bridge. http://developer.android.com/tools/help/adb.html
2. Android reverse engineering and defenses. https://bluebox.com/technical/bluebox-berlinsides-presentationbluebox-berlinsides-presentation/
3. Bornstein D (2008) Dalvik vm internals. In: Google I/O developer conference, vol 23, pp 17–30
4. Cesare S (1999) Linux anti-debugging techniques (fooling the debugger). Security focus
5. Dex file. https://source.android.com/devices/tech/dalvik/dex-format.html
6. Dexprotector by licel. http://dexprotector.com/
7. Enck W, Octeau D, McDaniel P, Chaudhuri S (2011) A study of android application security. In: USENIX security symposium, vol 2, p 2
8. Fengsheng Y (2011) Android internals: system
9. Gagnon MN, Taylor S, Ghosh AK (2007) Software protection through anti-debugging. IEEE Secur Priv 5(3):82–84
10. Huang J (2012) Understanding the dalvik virtual machine. Google Technology User Groups, Taipei
11. Ida pro disassembler and debugger. https://www.hex-rays.com/products/ida/. Accessed 26 Mar 2015
12. Java debug wire protocol. http://docs.oracle.com/javase/7/docs/technotes/guides/jpda/jdwp-spec.html. Accessed 25 Mar 2015
13. Java platform debugger architecture. http://docs.oracle.com/javase/7/docs/technotes/guides/jpda. Accessed 25 Mar 2015
14. Jung JH, Kim JY, Lee HC, Yi JH (2013) Repackaging attack on android banking applications and its countermeasures. Wirel Pers Commun 73(4):1421–1437
15. Khan S, Khan S, Banuri H, Nauman M, Alam M (2009) Analysis of dalvik virtual machine and class path library. Tech. rep. Security Engineering Research Group, Institute of Management Sciences, Peshawar
16. Lee C, Jeong YS, Cho SJ (2013) A method to protect android applications against reverse engineering. J Secur Eng 10(1):41–50
17. Schallner M (2006) Beginners guide to basic linux anti anti debugging techniques. Code-Break Mag, Secur Anti-Secur Attack Def 1(2):3–10
18. Schulz P (2012) Code protection in android. Rheinische Friedrich-Wilhelms-Universitgt Bonn, Institute of Computer Science, Bonn
19. Selvakumar G (2012) Constructing an environment and providing a performance assessment of androids dalvik virtual machine on x86 and arm. Ph.D. thesis, University of Kansas