J Supercomput (2016) 72:3629–3645
DOI 10.1007/s11227-016-1763-2

Mobile application tamper detection scheme using dynamic code injection against repackaging attacks

Haehyun Cho[1] · Jiwoong Bang[1] · Myeongju Ji[1] · Jeong Hyun Yi[1]

Published online: 2 June 2016

Abstract The Android platform, with a large market share from its inclusive openness, faces a big problem with repackaging attacks, because reverse engineering is made easy due to the signature method that allows self-sign and also due to application structure. A repackaging attack is a method of attack, where an attacker with malicious intent alters an application distributed on the market to then redistribute it. The attacker injects into the original application illegal advertisement or malicious code that extracts personal information, and then redistributes the app. To protect against such repackaging attacks, obfuscation methods and tampering detection schemes to prevent application analysis are being developed and applied to Android applications. However, through dynamic analysis, protection methods at the managed code can be rendered ineffective, and there is a need for a protection method that will address this. In this paper, we show that, using Dalvik monitor, protection methods at the managed code level can be dynamically analyzed. In addition, to prevent a tampered application from running, we propose a tampering detection scheme that uses a dynamic attestation platform. It consist of two phases; (1) *detection code injection*: inject tamper detecting code into an application and (2) *code attestation*: attest the injected code on the platform. The proposed scheme first uses the tamper detection method at the

✉ Jeong Hyun Yi
jhyi@ssu.ac.kr

Haehyun Cho
haehyuncho@gmail.com

Jiwoong Bang
jiwoongbang@gmail.com

Myeongju Ji
wlaudwn007@gmail.com

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

platform level to inspect execution codes executed in real time and to fundamentally intercept repackaged applications.

Keywords Android application protection · Tamper detection · Android platform

1 Introduction

Along with the growth of the mobile device market, the mobile application market, as well, is growing exponentially. Our expectations are that Android will continue to play an ever greater role in the mobile application market. Even with the dawn of the internet of things (IoT) [1] world in the near future, Android, which is founded on openness, has the potential to be established as a mainstay of this new development. However, despite the large market share of Android and its lofty predictions of future use, the large amount of malware affecting this operating system is increasing sharply [2].

The primary cause of these problems associated with Android protection is repackaging attacks where, after the attacker decompiles the APK file, they re-sign the application with their private key rather than with the developer's signature before distributing the application in the market [3,4]. Through repackaging attacks, the attacker can manipulate the application to inject advertisements or malicious code that extracts users' personal information or, when attacking a banking app, can create a financial accident to direct the money to the attacker's account when a user uses the account transfer service on the app [5].

Although obfuscation and tamper detection techniques [6] are being used in response to the repackaging attacks on Android apps, mostly using obfuscation techniques at the managed code level, this could be revealed and an analysis is possible practically anytime. Furthermore, on the platform level, when outputting instructions using the Dalvik virtual machine, the encrypted obfuscation code can easily be analyzed without secret key information. Thus, there is a need for research leading to the development of techniques capable of detecting when apps are tampered with, even under circumstances in which repackaging attacks and Dalvik monitoring could possibly occur anytime.

This prompted us to find ways to protect an application from repackaging attacks. This paper proposes a dynamic code attestation scheme with the ability to disable tampered applications to prevent them from running at the platform level. The proposed method assumes that the analysis is possible on all, and uses the Dalvik Monitor with the aim of preventing an attacker from tampering with the app. We have named our scheme *DexAttestor*.

The paper is organized as follows. In Sect. 2, we describe the backgrounds on the code obfuscation and the tamper detection techniques. In Sect. 3, we describe how to analyze Android applications using Dalvik Monitor. In Sect. 4, we propose the temper detection scheme based on dynamic code injection. We describe the architecture of the proposed system in Sect. 5. In Sect. 6, we explain the experimental results of the proposed scheme using the built prototype. In Sect. 7, we discuss on the advantages and the disadvantages of the proposed scheme. Finally, we draw the conclusions.

2 Background

2.1 Code obfuscation

In the case of Android, because applications that permit self-sign are extremely vulnerable to repackaging, applying code protection schemes previously mentioned, such as obfuscation and tampering detection schemes, is vital. Major protection tools for Android applications are Stringer [7], Allatori [8], DexProtector [9], DexGuard [10], etc.

Obfuscation is the representative code protection method for protecting algorithms or important code structures. Its purpose is to prevent analysis by making it difficult to use reverse engineering [11] to perform analyses. Obfuscation can be classified into categories, such as layout obfuscation, control flow obfuscation, and data obfuscation [12].

Layout obfuscation is a method that is used to complicate attempts to perform an analysis by changing identifiers predetermined during application development, such as class and variable and removing debugging information [13]. If these identifiers remain unchanged, it is easy to unravel the function and structure of an application using the identifiers of the class, method, and variables that have been labeled by their function and use.

Control flow obfuscation, a method that alters the execution flow of an application, uses complex conditional statements, repetitive statements, dummy code, etc., to disrupt the flow of content to hinder analysis [14]. In addition, JNI could be utilized to frustrate an attempt to analyze the control flow by invoking the Java function in the native code.

Data obfuscation protects data by encoding the stored character string using methods, such as restructuring of the storage type and access method. When static analysis is used, this scheme will prevent the exact value of each data value from being determined. Other than the above-mentioned methods, an API concealment obfuscation scheme, which uses a feature, such as Java reflection [15], also exists. Instead of directly calling the API used in the program from the attacker, this scheme only calls information that can be dynamically gathered and, thus, hinders analysis.

Yet, another obfuscation scheme encrypts the entire class much like string encoding [11]. Once the class is encoded and stored in data or in another storage location, the application performs a decryption routine before being executed. When class encryption is used, it is not possible to obtain any information regarding the encrypted class using the static analysis. Alternatively, packers are used to encrypt the entire DEX file and load it dynamically similar to a class encryption scheme; therefore, this approach can also be used to prevent an adversary from obtaining bytecode.

2.2 Tamper detection

The tamper detection method, similar to obfuscation, protects program code for the purpose of preventing tampering attacks. Unlike obfuscation schemes, however, the tamper detection method can actively respond to tampering attacks on the app. The

application itself determines whether the entire application has been tampered with and, based on the result, decides whether to run. This method is applied by injecting a routine capable of carrying out tamper detection using methods such as those that involve inspecting the integrity of the execution file of the intended program. When an attacker carries out a tampering attack, the above-mentioned routine runs when the attacked program is executed and detects that the program has been tampered with.

The tamper detection method typically functions using the hash value of the execution code, and this value is either stored in the data area within the execution file or on the server. If it is stored and run on the server when the tamper detection routine runs, the hash value of the original code is sent from the server. The routine then runs the received hash value and compares it with the calculated hash value to determine whether the application has been tampered with. A program in which tampering has been detected will terminate itself.

Another method functions by receiving a tamper detection code instead of the hash value from the server [16]. This method prevents analysis by separating the tamper detection code from the app. If the mobile device sends a verification request, the tamper detection code is generated in the server and sent to the device, which then executes the code and calculates the hash value. The detailed structure of the process is shown in Fig. 1.

Protection of the application with the use of the tamper detection method makes it essential to protect the tamper detection routine from an attacker to prevent the routine from being found. If it is possible to easily expose the tamper detection routine, it is not very difficult for an attacker to bypass the tamper detection method. Thus, rather than

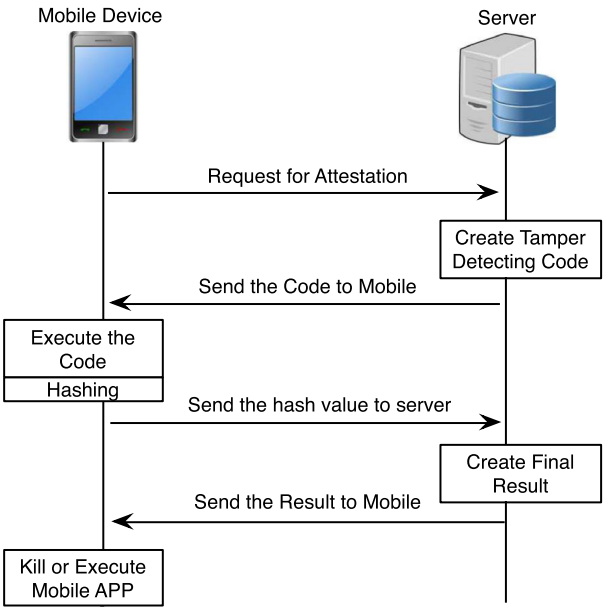

Fig. 1 Example of a tamper detection method

the tamper detection method being used independently, it is common for the method to be used along with the various code protection methods introduced earlier.

2.3 Anti-analysis schemes

Several anti-analysis schemes have been developed to counter both static and dynamic analyzing methods in Android systems. Reverse engineers or hackers attempt to unbox an APK file statically to acquire byte code or metadata to analyze an application. In general, manifest file cheating [17], forgery ZIP file encryption [18], or method concealment [19] schemes could be harnessed to prevent the static analysis. These schemes prevent adversaries from obtaining the bytecode of an application and other information by modifying metadata, which only has an impact on static analysis tools, such as apktool, but do not affect its execution.

Typically, an app is analyzed dynamically using the IDA, GDB, and NetBeans debuggers. Dynamic analysis using debuggers offers definite advantages compared with static analysis as mentioned before. However, the use of debuggers to analyze an application requires root permission, which is a constraint. Thus, an application for which anti-rooting schemes have been applied cannot be analyzed by debuggers before these schemes are subverted. Some anti-debugging schemes [20] are designed to thwart analysis using debuggers.

In addition to the above schemes, emulator detection schemes could be used to prevent analysis [21], because there are a number of analysis tools based on an emulator. An emulator can be detected using its IMEI number, kernel message, thread scheduling, and so on. In the case of a QEMU emulator, it has a uniform IMEI number and kernel log containing the QEMU string. In addition, because its thread scheduling policy is in numerical order, we can expect a multi-thread program to achieve definite results. Therefore, identifying emulators based on their features can render an application unanalyzable against such emulator-based analysis tools.

3 Android application analysis

3.1 Analytical methods

Compiled execution codes for Android applications can be separated into managed code and native code. Managed code is compiled in Java, such that the generated code is executed not in the CPU but in the Dalvik virtual machine. Native code refers to code in languages, such as C and C++, i.e., it is machine code that can be recognized by the CPU. Managed code [22] includes information about the Android API that is used and, compared to native code, contains a large amount of information needed for analysis; thus, it becomes the main target for the analysis during a repackaging attack.

Methods for application analysis can be largely grouped into static analysis and dynamic analysis. The static analysis of an Android application is executed by disassembling the managed code (Smali codes) using tools, such as ApkTool [23] and BakSmali [24]. Static analysis cannot access the value of encrypted class variables, such as the code, character, or string and, if the code of an app is partially sepa-

rated onto somewhere, such as the server, analysis is impossible. On the other hand, dynamic analysis can extract important data for analysis purposes that is hard to understand using static analysis, and this form of analysis is easier to implement than static analysis [25].

3.2 Analysis using the Dalvik Monitor

The Dalvik virtual machine, which executes managed code on the Android platform, can be modified to create a platform for dynamic analysis as a part of the Android open source project (AOSP) [26]. The Dalvik Monitor, which processes and outputs the information needed for analysis based on bytecode, is executed in Dalvik virtual machine. This enables anyone to conduct a dynamic analysis anytime without knowing the secret key information of an app, using obfuscation techniques based on class encryption and code concealment, such as string encryption. This is because the code and data executed in the Dalvik virtual machine can be dynamically monitored in a form similar to Smali with information, such as string.

As shown in Fig. 2, we can extract all the execution codes running in real time and observe the registers of the virtual machine and the value of each of the variables or parameters using the Dalvik Monitor. In addition, the instructions of the encrypted classes can also be extracted, including the encrypted strings. The fact that it is possible to monitor obfuscation methods at the level of managed code in the Dalvik Monitor using these functions and suggests that the use of this monitor to perform analyses would be easy.

In this work, we used the Dalvik Monitor to extract encrypted bytecode [27], which we did by implementing the monitor to execute a simple synthetic application of which the class is encrypted on the Dalvik Monitor. Selected obfuscation tools for encryption are DexGuard and DexProtector. Figure 3 shows source code written for the class encryption test.

Figure 4a, b shows the outputs of analyses using the Dalvik Monitor. These outputs are protected by DexGuard and DexProtector, respectively. As you can see from the test results, although the class was encrypted, an attacker would have no problem

```
|1|invoke-static args=1 @0x04bb {v4, v0, v0, v0, v0}
|1|[fp=0x6d455bd8] Lexample/dalvik/monitor;->a(Ljava/lang/String;)Z
|1|paramCnt : 1, args 1
|1|param1 Ljava/lang/String;   -> This is an example of dalvik monitor

|1|sget-object v4, Lexample/dalvik/monitor;->test:Ljava/lang/String; sfield@0x0001
|1|+ SGET 'test'=0x42984628

|1|const-string v0 string@0x000051fc, string = example string

|1|const/4 v1, #0x01

|1|move v2, v1          (v1=0x00000001)

|1|return v2

|1|retval=0x00000001
|1|return to Lexample/dalvik/monitor;->b(Landroid/content/Context;)Z [fp=0x6d455c24]
```

Fig. 2 Output of Dalvik Monitor

Fig. 3 Source code for class encryption

```
public class SecretClass {
  static String SecretMessage = "MSEC";
  public String getMessage() {
    String Message = "Hello";
    Message=Message.concat(SecretMessage);
    return Message;
  }
}
```

```
    |1|return to
       Ljava/lang/String;->
       concat(Ljava/lang/String;)Ljava/lang/String;
  |1|return-object v7
  |1|retStr = HelloMSEC
  |1|retval = 0x742488290
  |1|return to Lo/槲;->鷭()Ljava/lang/String;
|1|move-result-object v1(v1=0x42488290)
|1|return-object v1
|1|retStr = HelloMSEC
```

(a)

```
    |1|return to
       Ljava/lang/String;->
       concat(Ljava/lang/String;)Ljava/lang/String;
  |1|return-object v7
  |1|retStr = HelloMSEC
  |1|retval = 0x7424d1a88 (leaving native)
  |1|return from native
     Lcom/dexprotector/ha;->lldab…
     to Lcom/ddg/msec/SecretClass;->getMessage()…
|1|move-result-object v0(v0=0x424d1a88)
|1|check-cast v0,class@0x000a
|1|return-object v0
|1|retStr = HelloMSEC
```

(b)

Fig. 4 **a** Extracted code encrypted by DexGuard. **b** Extracted code encrypted by DexProtector

analyzing the structure and function of the encrypted class using the output class without having information on the secret key.

Using the Dalvik Monitor in this way enables anyone to easily even analyze applications that apply complex obfuscation and tamper detection schemes. Therefore, there is a limit to the level of protection that can be provided by protection methods at the managed code level, leaving applications vulnerable to becoming targets of a tampering attack at any time. This indicates that we need application tampering detection methods at the platform level to protect users from repackaging attacks.

4 Proposed scheme

Until this point, we have confirmed that using protection schemes at the application level does not allow us to properly protect an application from tampering attacks. This

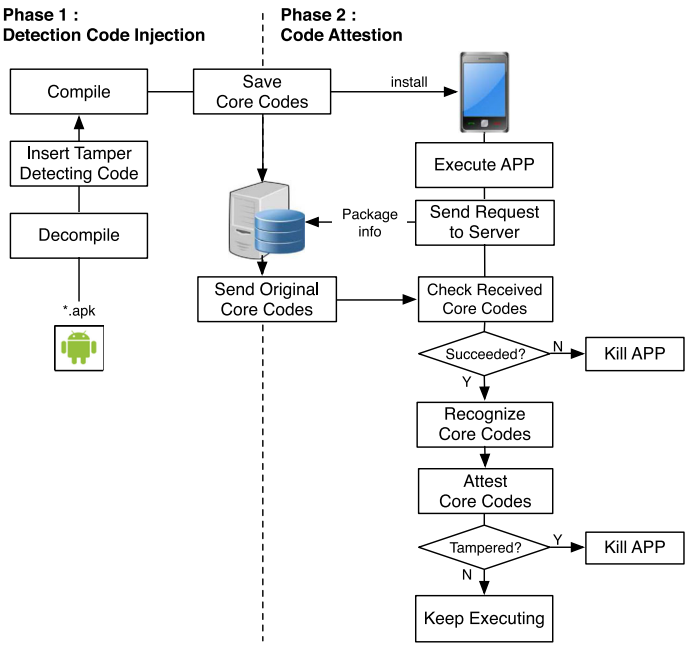

Fig. 5 Structure of proposed scheme

led us to propose DexAttestor as an Android application tamper detection method that uses a dynamic verification platform to protect applications and provide a safe running environment.

The proposed scheme is largely composed of two parts. During the preliminary preparation stage, core code is injected, with the purpose of detecting tampering in the existing managed code. This is followed by the key part of the proposed scheme, namely, the attestation stage, which occurs on the mobile device when the application begins to run.

In this work, the section that injects tamper detection code is referred to as the detection code injector, whereas the section that regulates code attestation is referred to as the code attestor. The injected tamper detection code is simply a code that determines whether there has been tampering based on the hash value of the Dex file, i.e., the execution file of the Dalvik virtual machine. In this work, the injected tamper detection code is referred to as core code.

The attestation stage is carried out using a method of inspecting instructions executed in the Dalvik virtual machine by units of instruction. The attested code inspects the hash value of the running application and checks its integrity, whereupon the proposed scheme, at the platform level, verifies that the code and data that authenticate the integrity of the application have not been tampered with. The structure of the entire tamper detection technique is shown in Fig. 5.

When an application undergoing tamper verification is run, application information is sent to the server, whereas information pertaining to the core code and the method that includes the core code are received from the server. When core code is executed, the

instructions of the core code are compared to the received instructions of the original code and inspected. During the inspection, if another instruction, which differs from the original one, is executed or the parameters of the instruction are different, the code attestor decides that the application has been tampered with and terminates it. Applications that inspect the entire length of the original code verify the integrity of the tamper investigation routine and are, thus, deemed safe, in which case, the Dalvik virtual machine switches to execution mode and continues to run the app.

5 Architectural design

The architecture of the proposed scheme for dynamic application integrity verification is shown in Fig. 6. The Dalvik virtual machine (VM), which implements the proposed scheme, has two kinds of modes, execution mode and attestation mode. The application is launched in attestation mode. When a code attestor module finishes attesting the core code, the mode of the Dalvik VM is changed to execution mode.

The code attestor included in the Dalvik VM has an attestation interpreter that executes Dalvik instructions and compares them with the original core code received simultaneously from the server for tamper detecting.

The code attestor is composed of data transfer, execution trigger, attestation interpreter, and crypto modules. The attestation server has data transfer, database, and crypto modules.

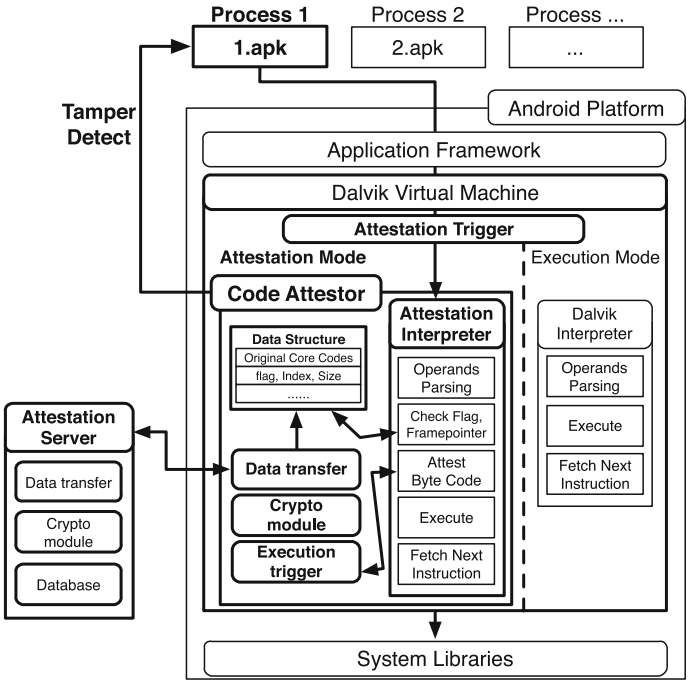

Fig. 6 Architecture of DexAttestor

5.1 Detection code injector

Android applications comprise multiple activities. An activity facilitates interaction between the user and the application in the Android system by providing a visible display, such as a dial or web browser [28]. In general, the activity first executed when the application runs are known as the main activity. The tamper detection code proposed in this paper is injected into the onCreate method of the main activity. This is because onCreate method, as the section that initializes each activity of the Android app, is the template method of the Android framework that cannot be deleted or have its name changed [28]. Thus, if the tamper detection code is injected into the onCreate method, we can prevent an attack that bypasses the execution of core code. As tamper detection code can be defined in various ways, we have deemed the definition of the code and the determination of its structure to be beyond the scope of this paper.

5.2 Code attestor

The code attestor that operates the overall tamper detection process and regulates code attestation does not exist as a specific module; instead, it exists with all of the Dalvik virtual machine (VM) codes. This signals that the Dalvik VM has made the transition from its original state to a modified state in which the code attestation function has been added to several areas, including the interpreter.

Application attestation occurs through the server and the modified Dalvik VM that loads the proposed platform level code attestation function. The code attestor can be classified as being in either attestation mode in which the application is attested or execution mode in which only the functions for execution are run.

In attestation mode, the proposed system operates as follows. Before the application runs and the interpreter that executes the Dex file is run, the Dalvik VM, from the list of applications that need default tamper detection, verifies the package name of the application that is running and determines whether the application should undergo tamper detection. Before running the interpreter, the verified application requests the core code from the server that called and executed data transfer and receives the relevant information of the core code. The received data include the method that contains codes along with the core code and class information. If the process of receiving the data is not complete, the application will not run.

When the code attestor has finished receiving the data, the application is run and, based on the method according to which the data are received and class information, wait for the execution of the method that includes the core code. In the Dalvik VM, the instruction that calls the method is carried out using the Invoke instruction. This instruction, as an operand, must use the parameter to be transmitted to the calling method and the descriptor, method name, and the descriptor of the calling class. This enables us to use the received data to verify whether the core method has been called. The moment the method that includes the core code is called, the length of the core code is stored in the global structure of the Dalvik virtual machine, and the flag that determines that code attestation must be executed is modified. After verifying the flag that indicates that attestation has begun, a task for core code attestation is additionally

executed in all the sections that carry out commands. Code attestation, after operands parsing has occurred, is carried out by comparing the instruction, the hexadecimal value of the operands, such as a register number, and the character string that converted the string object into C string with the originals.

5.3 Attestation interpreter

The Dex file is executed by interpreting the Dalvik commands into machine language compatible with the CPU architecture of the mobile device. The type and number of operands differ according to the command type of the Dalvik instruction, each of which uses a different method to parse operands. An operand consists of Dalvik registers and an index for accessing data, such as the value of a constant or string. The code attestor detects tampering by comparing the values of the instructions and operands after the operands have been parsed when the instructions of the dex file are loaded into memory and executed with the original values. For example, the data needed to inspect the instruction that is `const/4 v0, 0x0` will be made of `0x12`, `0x00` (Fig. 7).

In the case that a string object is accessed, an attestation of the entire string value is needed. This is because when object or field data is accessed, it is accessed using the index of that data such that, in the case that data has been modified, we cannot determine the integrity of the data using only commands and operands. Especially, regarding string objects, they often contain important information, such as an important hash value. Thus, we need a process that attests string content by converting the string object stored in the dex file into a C string. In this paper, we use the ASCII code value of the converted C string stored as a hexadecimal and inspects it for attesting the string.

The frame pointer, as the pointer that indicates the stack frame of the methods managed by the Dalvik VM, always indicates the frame of the method being executed. Core code attestation is carried out using this frame pointer on only the code of the

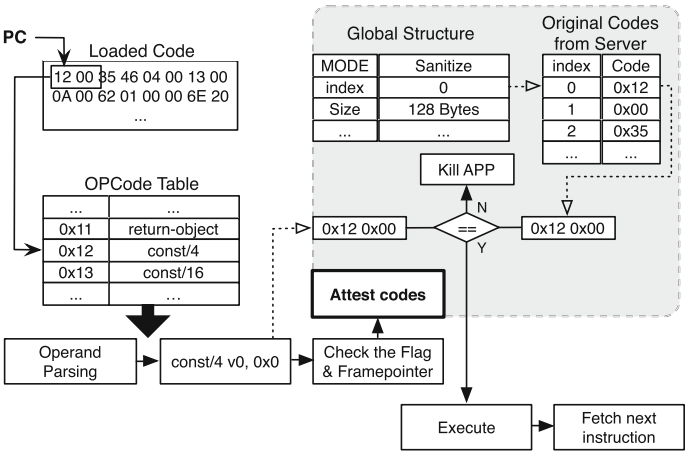

Fig. 7 Attestation interpreter built into the Dalvik interpreter

target method. Other methods called by the core method, and Android framework codes are not inspected.

Once attestation of the codes requiring verification is complete, the Dalvik VM does not need any further verification process. This is because the previously determined application tamper detection routine is injected into the core code, and the integrity of the entire application can now be demonstrated with just the attestation of the tamper routine. Therefore, once the task of verification has been completed, the proposed system operates by simply executing those functions that run the application, such as the Dalvik VM, in the existing Android platform. On the other hand, during attestation, if the application is determined to having been tampered with, the log is printed out and the application is terminated.

5.4 Data transfer and crypto modules

There is a need for a module that manages data transfer to and from the server. In addition, a module that encrypts data transmitted through the data transfer module is required. Thus, these modules were implemented in code attestor to send or receive the data securely.

5.5 Attestation server

The main function of the server is, based on the package information received from the mobile device, to send the original core code of that package from the database. In our proof-of-concept prototype, when the original core code is sent, the integrity of the data is ensured using HMAC (keyed-hash message authentication code) [29]. The brief description of the protocol between the server and the mobile platform is as follows. To request the core code, DexAttestor sends an identifier of the core code (ID_{cc}) which contains application-specific information and/or device unique id. The attestation server retrieves the corresponding core code (CC) and then generates a hash value $HMAC(K_s, CC)$ using the shared secret key (K_s). Then, the generated $HMAC(K_s, CC)$ is sent to the platform along with CC. DexAttestor on the platform side checks the integrity of the received data ($CC||HMAC(K_s, CC)$). We assume that the secret key is pre-shared. The detailed methods for key distribution are beyond the scope of this paper.

6 Experimental results

6.1 Implementation

The DexAttestor prototype proposed in this paper was created as follows. The detection code injector was generated in the Eclipse IDE [30] development environment using Java. The code attestor and attestation interpreter were developed by modifying the Android version 4.4 Dalvik virtual machine. Experiments were run by porting the modified platform onto a Nexus 5 device.

The data transfer module used a TCP socket, and base64 was used for the encoding of transmitted data. The crypto module was developed using OpenSSL [31]—one of the native libraries used by Android.

Apache was used as Linux OS middleware. Using PHP5 as the server language and MySQL for the database, a LAMP (Linux Apache PHP MySQL) [32] server was implemented.

6.2 Target application

The target application for the experiment was a W-bank application version 3.3.3 that was specifically structured to bypass the tamper detection routine. Using the same attack scenario as experimented in [5], the application was repackaged such that funds would not be transferred to the intended recipient's account but to that of the attacker instead. We presumed that the attacker knew that the tamper detection code as proposed in this paper had been injected into the repackaged application and, in response, built an application to bypass this.

6.3 Result

The experiment was carried out by largely following two approaches. The target application was first run on an established Android device, after which it was run on an Android device to which DexAttestor was applied, as proposed in this paper.

Figure 8 shows the result of running the target application on the established Android platform. As usual, the repackaging attack succeeded, and only standard operating procedures can be observed.

Figure 9 shows the result of running the same application on an Android device with DexAttestor built in. Results showed that the original core code and executed code differed for the target application built to bypass the tamper detection routine, causing the application to be terminated along with a message alerting the user of tampering.

Fig. 8 Target application

Tag	Text
dalvikvm	msec|Index : 118, Running code : 1, Original code : 1
dalvikvm	msec|Index : 119, Running code : 2, Original code : 2
dalvikvm	msec|Index : 120, Running code : 0, Original code : 0
dalvikvm	msec|Index : 121, Running code : 0, Original code : 0
dalvikvm	msec|Index : 122, Running code : f, Original code : f
dalvikvm	msec|Index : 123, Running code : 8, Original code : 8
dalvikvm	msec|Index : 124, Running code : 1, Original code : 1
dalvikvm	msec|Index : 125, Running code : 0, Original code : 0
dalvikvm	msec|Index : 126, Running code : 2, Original code : 2
dalvikvm	msec|Index : 127, Running code : 9, Original code : 5
dalvikvm	msec|TAMPER DETECT!! This APP is modified.

Fig. 9 Target application is killed by DexAttestor

The core code of the tamper detection scheme proposed in this paper refers to the section of code that cannot be altered by an attacker when running the app. If the application was manipulated to bypass this code section, the instructions are changed, and differed from the original code. Thus, a message alerting to tampering would appear, and related processes would be terminated in Dalvik. If you print out and inspect the entire code attestation process and all of the results using the Android log system, you would be able to identify that they are the results of operating in the platform. This tamper detection method of using code attestation at the platform level supplements the existing weak protection techniques at the application program level. Moreover, it provides users with a secure execution environment.

7 Discussion

DexAttestor has both advantages and disadvantages. The disadvantages of DexAttestor are the following: (1) when the app is launched, the network must be available; (2) the proposed scheme needs the attestation server to verify the app; (3) the proposed scheme could be adopted on the smartphone only by the mobile device manufacturers; and (4) launching the app results in performance overhead to launch the app.

As a complement to the first and second disadvantage, based on costs and usability, wearable devices, such as a smart watch, could be used instead of the attestation server. More recently, mobile device manufacturers have released smart watches that are used in combination with a smart phone and are developing wearable devices. Those smart watches that have been released have storage, a CPU, and memory, and can be connected to a smart phone using Bluetooth. Therefore, a wearable device can perform the role of the server.

7.1 Performance overhead

We selected 50 Android applications, to which the tamper detection scheme was not applied, from various categories of Google Play to evaluate the performance overhead of DexAttestor. All tests are conducted on the Nexus 5 with DexAttestor and in the same environment. Thus, in this evaluation, we assume that the transaction time for receiving the core codes is the same; thus, it is not considered. The size of the core code is about 24 KB.

The overhead was determined by checking the time required to load the MainActivity for each application, because DexAttestor starts running before the MainActivity starts. Figure 10 shows the result of the evaluation. In the case of DexAttestor, the average time for loading the MainActivity is 6,308 ms, whereas the average time without DexAttestor is 110 ms.

Furthermore, the overhead caused by DexAttestor varies according to the size and number of threads of an application. As a consequence, the result of the evaluation showed that about 6200 ms is needed to operate DexAttestor on average.

This overhead impose on every time when the target application is launched. To reduce this overhead, we can employ a policy that decreasing the number of tamper detection using DexAttestor. For example, if we conduct the tamper detection just one time after the application is installed, we can avoid the overhead. However, it would give the possibility to adversaries for the repackaging attack, since applications can be upgraded at runtime. Therefore, the core code should be updated with respect to the any type of updates of an application to inspect updated code and the tamper detection should be conducted.

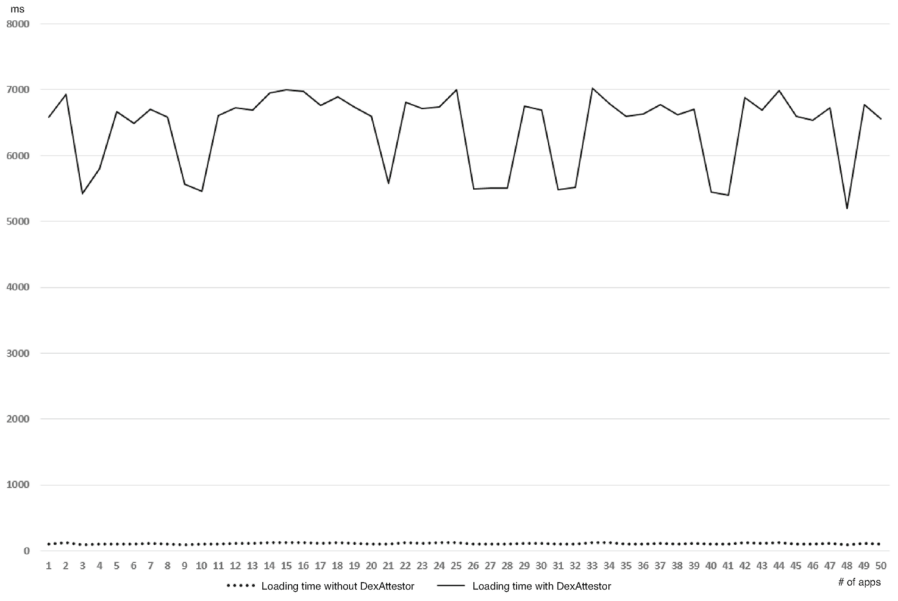

Fig. 10 Performance overhead of DexAttestor

8 Conclusion

Although various schemes to prevent Android application tampering attacks and analyses, such as obfuscation, are being combined and used, they are inadequate in responding to attacks on the platform, such as at Dalvik monitoring, and, instead, lead to a performance degradation of the app.

In this paper, we proposed a tamper detection scheme that uses platform-level core code attestation to address the weaknesses of existing managed code obfuscation methods and tamper detection techniques. We showed that the analysis of encrypted application was possible without secret key information using the Dalvik Monitor.

Unlike existing security methods that remain vulnerable to analysis and bypassing, the proposed scheme, as a security service executed on the Android platform, can fundamentally stop the execution of applications that have been tampered with through repackaging. In addition, tampering in applications can now be detected without using complicated functions, such as string encryption or class encryption that can decrease efficiency.

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

References

1. Kopetz H (2011) Internet of things. In: Real-time systems. Springer, Berlin, pp 307–323
2. Wang X, Yang Y, Zeng Y, Tang C, Shi J, Xu K (2015) A novel hybrid mobile malware detection system integrating anomaly detection with misuse detection. In: Proceedings of the 6th international workshop on mobile cloud computing and services. ACM, pp 15–22
3. Arp D, Spreitzenbarth M, Hubner M, Gascon H, Rieck K. Drebin: efficient and explainable detection of android malware in your pocket. In: Proc. of 17th network and distributed system security symposium, NDSS, vol 14
4. Enck W, Octeau D, McDaniel P, Chaudhuri S (2011) A study of android application security. In: USENIX security symposium, vol 2, p 2
5. 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
6. Aucsmith D (1996) Tamper resistant software: an implementation. In: Information hiding. Springer, Berlin, pp 317–333
7. Stringer. https://jfxstore.com/stringer/
8. Allatori. http://www.allatori.com/
9. Dexprotector. https://dexprotector.com/
10. Dexguard. https://www.guardsquare.com/dexguard
11. Schulz P (2012) Code protection in android. Insititute of Computer Science, Rheinische Friedrich-Wilhelms-Universitgt, Bonn
12. Collberg C, Thomborson C, Low D (1997) A taxonomy of obfuscating transformations. Tech. rep., Department of Computer Science, The University of Auckland, New Zealand
13. Brzozowski M, Yarmolik VN (2007) Obfuscation as intellectual rights protection in VHDL language. In: 6th international conference on computer information systems and industrial management applications, CISIM'07. IEEE, pp 337–340
14. Low D (1998) Java control flow obfuscation. Ph.D. thesis, Citeseer
15. Forman IR, Forman N, Ibm JV (2004) Java reflection in action
16. Piao Y, Jung JH, Yi JH (2016) Server-based code obfuscation scheme for apk tamper detection. Secur Commun Netw 9(6):457–467
17. Android reverse engineering and defenses. https://bluebox.com/wp-content/uploads/2013/05/AndroidREnDefenses201305.pdf

18. Fake encryption sample. https://github.com/blueboxsecurity/DalvikBytecodeTampering
19. Apvrille A (2013) Playing hide and seek with Dalvik executables. In: Hack. Lu, October (2013)
20. Cho H, Lim J, Kim H, Yi JH (2016) Anti-debugging scheme for protecting mobile apps on android platform. J Supercomput 72(1):232–246
21. Petsas T, Voyatzis G, Athanasopoulos E, Polychronakis M, Ioannidis S (2014) Rage against the virtual machine: hindering dynamic analysis of android malware. In: Proceedings of the seventh European workshop on system security. ACM, p 5
22. Alliance OH (2011) Android overview. Open Handset Alliance, USA
23. Apktool. http://ibotpeaches.github.io/Apktool/
24. Baksmali. https://github.com/JesusFreke/smali
25. Yan LK, Yin H (2012) Droidscope: seamlessly reconstructing the OS and Dalvik semantic views for dynamic android malware analysis. In: USENIX security symposium, pp 569–584
26. Android open source project. https://source.android.com/
27. Yi JH, Cho H, Bang J, Ji M (2015) Application code analysis apparatus and method for code analysis using the same. KR Patent 101557455
28. Developers A (2009) Android activity
29. Bellare M, Canetti R, Krawczyk H (1996) Keying hash functions for message authentication. In: Advances in cryptology, CRYPTO'96. Springer, Berlin, pp 1–15
30. Eclipse. https://eclipse.org/
31. Viega J, Messier M, Chandra P (2002) Network security with openSSL: cryptography for secure communications. O'Reilly Media Inc., Sebastopol
32. Ware B et al (2002) Open source development with LAMP: using Linux, Apache, MySQL and PHP. Addison-Wesley Longman Publishing Co., Inc., Boston