Multi-partitioned Bytecode Wrapping Scheme for Minimizing Code Exposure on Android

Yongjin Park, Taeyong Park, Jeong Hyun Yi

School of Software, Soongsil University, South Korea
{absolujin, taeyong88}@gmail.com, jhyi@ssu.ac.kr

Abstract

To date, the Android has an overwhelming share of the global smartphone market and an immense number of users. However, owing to their structural problems, Android applications are more vulnerable to reverse engineering attacks compared to other mobile applications. Attacks on Android applications not only infringe on the intellectual property rights of application developers, but they can lead to the theft of private user data and financial damage. Developers have applied several techniques to protect Android applications from these attacks. One method is the packing technique. However, because the packing technique is a protection technology for the entire code, the core code is continually exposed in dynamic memory until the process terminates. In other words, if the attacker succeeds in dynamic memory analysis, the attacker may easily obtain the logic of the core code. Therefore, in this paper, we propose a scheme that makes reverse engineering analysis difficult. This is achieved by wrapping and dropping the original bytecode after separating the original bytecode, thereby resolving the problem of exposing all the original bytecodes. The proposed scheme was applied to a sample application, and the security and performance were compared with those of existing techniques.

Keywords: Android, Packing, Mobile application security

1 Introduction

The smartphone operating system, such as Google's Android and Apple's iOS, can provide a variety of services by installing user-desired applications and basic communication functions, such as phone and Internet. These smartphones are increasingly used not only for personal information storage, but also for personal services, such as financial services. Among smartphone operating systems, the Android has the largest number of users (82.8%) [24]. However, it is subject to various attacks because its application structural problems and the way its applications are distributed.

The Android application is developed with Java and compiled in a bytecode format to enable decompiling back to the Java source code level. However, this process is vulnerable to reverse engineering attacks. Moreover, to distribute Android applications, user sign-in is required. It is thereby possible for the attacker to redistribute by self-signing with a private key [7]. Attackers exploit these vulnerabilities to inject malicious code into an application and distribute it, causing indirect harm to both developers and users. To protect Android applications that are vulnerable to such attacks, some security vendors, including DexProtector [20], Bangle [18], and Ijiami [21], have applied a packing technique that encrypts and protects the code.

Packing is a technique used by malicious code to prevent static analysis in personal computer (PC) environments [11]. Packers, such as UPX [23], increase static analysis resistance by executable compression using file format or loader characteristics [13]. However, the existing packing technique loads the entire original bytecode into memory at the time of loading the application. Thus, in a dynamic analysis environment, such as DECAF [9], DexHunter [12] and AppSpear [3], the original bytecode is extracted and analyzed. Therefore, existing packers cannot effectively protect the original bytecode. Recent PC based code protection studies [8, 10] have evolved packing into a form that reduces code exposure, preventing attackers from accessing the code, or protecting the code location. To effectively protect the code on mobile devices, it is necessary to protect the code from exposure to the attacker.

In this paper, we propose a scheme to effectively increase the resistance to dynamic analysis by minimizing the exposure of original bytecode in memory. To address the limitation of existing packers, which expose all original bytecode, the proposed scheme wraps and drops the original bytecode. Specifically, the original bytecode is wrapped into several individual codes. The wrapped codes are unwrapped only when the code is called and loaded into memory. After using the code, it is removed from memory through dropping to minimize the exposure

*Corresponding Author: Jeong Hyun Yi; E-mail: jhyi@ssu.ac.kr
DOI: 10.3966/160792642018081904022

time.

The remainder of this paper is organized as follows. Section 2 introduces mobile application repackaging attacks and techniques for preventing them. Section 3 describes the proposed multi-partitioned bytecode wrapping scheme. In Section 4, features of the proposed scheme are compared with those of existing packers. In Section 5, the experimental results are presented. Finally, our conclusions are provided in Section 6.

2 Background

In this section, we examine repackaging attacks against Android applications (apps) and techniques to protect the apps from them.

2.1 Repackaging Attack

The repackaging attack by malicious Android app producers [2] adds malware code to normal apps and redistributes them. These attacks are possible because of the process of building the app and its structural problems. The process of building an Android app is described as follows. When compiling the app in the Java language, a Dalvik executable (DEX) file is created that consists of several class files. The DEX file consists of a Dalvik bytecode that runs in the Dalvik virtual machine (VM). The result of adding a developer's signature to this DEX file and distributing it is an Android app is called an Android application package (APK) [16].

In this app-building process, it is apparent that the code containing the actual behavior is the Dalvik bytecode. As shown in Figure 1, this Dalvik bytecode is easily disassembled into an intermediate code form called Smali [2] by using an open-source tool, such as Apktool [17]. Smali code is more human-readable than native code; thus, it is easy for an attacker to insert malicious code in the proper place after analyzing the code. Therefore, the attacker can self-sign the modified application and redistribute it by repackaging it.

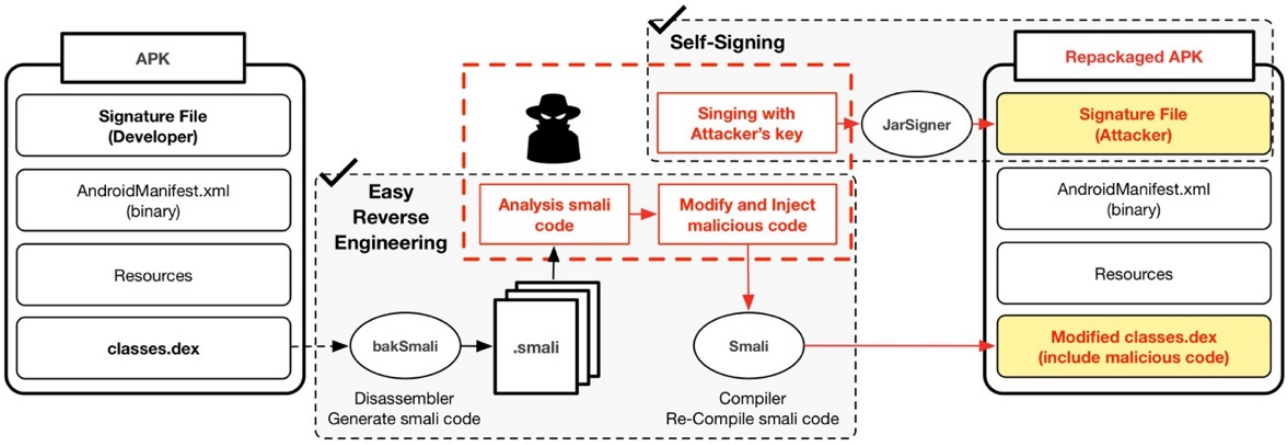

Figure 1. Repackaging attack and structural problem vulnerability

To counter this repackaging attack, developers have applied various techniques, such as obfuscation and packing, to Android apps. Obfuscation and packing are described below.

2.2 Obfuscation

Obfuscation is a scheme that protects the Java bytecode, which is vulnerable to reverse engineering [5, 15]. This obfuscation can be categorized as respective layer, control-flow, and data obfuscation. Layer obfuscation is the most basic obfuscation technique. It cannot recover original information by omitting or deleting specific information of original code. A typical example is an identifier obfuscation that replaces an identifier with a simple character, thereby removing information that is easy to perceive from the identifier. Control-flow obfuscation makes analysis difficult by complicating the process without changing the results of the original code. Typical examples are adding dead code that is not actually executed, dummy code that does not affect execution results, or a loop statement that is broken into multiple jump statements (such as *goto*). Data obfuscation is an obfuscation technique that encrypts the data or code, such as important strings in the executable code. Accordingly, the original information cannot be known during static analysis, making it impossible to grasp important parts. A typical example is string encryption, which is used to decrypt the original string when the string is called.

2.3 Packing

The packing technique was originally applied to malicious code in a PC environment to hinder detection by anti-virus software [11]. Recently, it has been actively used to protect mobile apps. The packing technique differs in the implementation details according to each tool. Nonetheless, the principles of packing and unpacking remain basically the same, as shown in Figure 2 and Figure 3. The process of

applying packing is the following. First, the original bytecode is packed by encryption and placed inside the app. Then, the code is added and unpacked, and the entry point of the app is changed to the unpacked code. This ensures that the unpacked code, which is specified as the entry point, is executed first when the app is executed. Finally, after the unpacked bytecode is loaded into memory, the original bytecode is called [14].

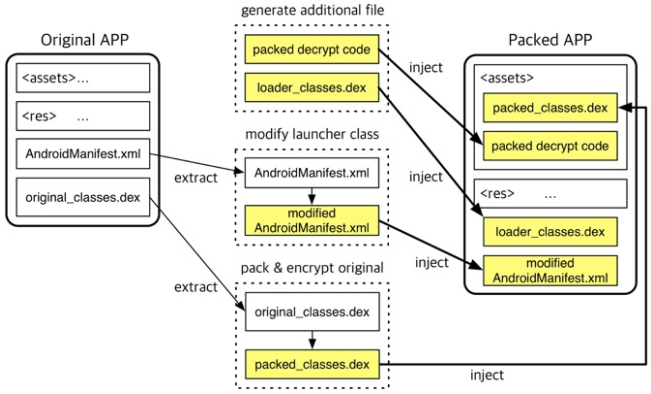

Figure 2. Packing process for Android apps

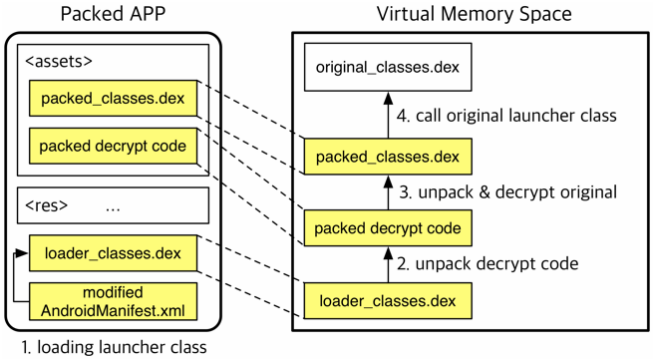

Figure 3. Unpacking process for Android apps

By analyzing the basic operations of the existing packing techniques [18, 21] that are applied to Android apps, it is evident that they share processes. First, they pack the original *classes.dex* and add the launcher class. Then, they modify the AndroidManifest.xml file to change the execution flow to the launcher class and add the unpacked related files. Furthermore, unique implementations of existing packers exist at each stage, which makes it easier to identify a specific packer during reverse engineering. In this regard, representative features of each packer can be identified in the launcher class for changing the program execution flow and adding files for unpacking.

For example, DexProtector divides the original bytecode into several packing codes. In unpacking, after integrity is secured through tamper detection, the decryption key is dynamically generated using the application signature information. Then, unpacking the entire original bytecode is performed [1]. Bangcle packs the original bytecode into one and places it in the asset path. Next, when the app is executed, the packed code is copied to the temporary path, unpacked, and the entire original bytecode is loaded. When loading is completed, the original bytecode unpacked in the temporary path is repacked to prevent exposure of the original bytecode file of the temporary path [3]. Ijiami, like Bangcle, packs the original bytecode into a single packed code, places it in the asset path, and unpacks and loads the entire original bytecode at launch time. Unlike other packers, Ijiami makes it difficult to identify the original bytecode in memory by loading the modified code with a different magic number or header size with the original bytecode header [3].

2.4 Dynamic Loading

Dynamic loading techniques unpack and load statically packed original code into memory and change the execution flow to the original code. All class files based on Java are dynamically loaded into a virtual machine. They can be divided into either load-time or run-time dynamic loading, depending on the moment that code is loaded. Load-time dynamic loading is a mandatory procedure for loading code; it is necessary for the initial execution. Run-time dynamic loading is an explicit loading of additional code by the developer at run-time to solve problems, such as dynamic code execution or heap memory limitation, according to the execution condition. Among them, the technology used for packing is run-time dynamic loading. It is possible to determine if packing is applied based on whether dynamic loading is performed at the time of the app analysis.

There are two ways to apply dynamic loading in Android apps: using the Java application programming interface (API) (*Dalvik.system.DexClassLoader*) provided by the Android framework, and calling the native API (*openDexFileNative()*) directly from the Dalvik virtual machine. There is no difference between the two methods because the method using the Java API is eventually connected to the native API. The relationship between the Java API and native API is shown in Figure 4.

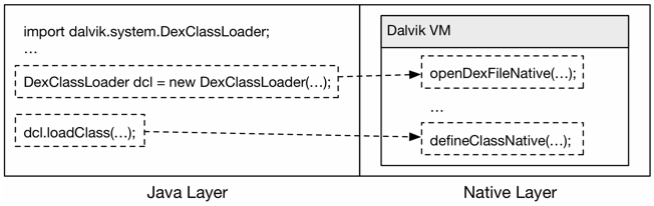

Figure 4. Relationships in the dynamic loading API

Because dynamic loading is basically requested by Java code, it is common to use the Java API. However, when using the Java API, the Dalvik bytecode is not suitable for code protection because its dynamic loading and location are easily exposed by its vulnerability to reverse engineering. On the other hand, using the native API provides a higher resistance to reverse engineering than Dalvik bytecode. Additionally,

when dynamic loading is applied using the native API, it is difficult to identify the position of the dynamic loading code and whether dynamic loading is used. This is because the part that calls the native API in Java is the same as the general JNI call. As a result, it becomes difficult to judge whether the packing is applied.

3 Proposed Scheme

As mentioned earlier, the existing packer technique packs the entire original bytecode, unpacks the entire packed code at runtime, and loads it into memory. This reduces resistance to dynamic analysis because the entire original bytecode is exposed in memory from the start of the app to the end. In the proposed approach, we protect the original bytecode in memory by reducing the exposure size and time of the original bytecode. To this end, we propose a packing technique that separates the core code from the entire code without applying packing to the entire code. In the following, we use the term wrapping instead of packing to distinguish it from existing packers that target the entire original code. The proposed scheme greatly reduces the exposure size by consisting of core bytecode with more than one method or class and by separately wrapping several core bytecodes. The core bytecode is exposed only from the time the call is made until the execution is completed. After execution completion, the core bytecode is dropped from memory to minimize the exposure time. The separated core bytecode is wrapped inside the native library (*.so*) file. Furthermore, as mentioned above, it is difficult to identify dynamic loading. Moreover, the execution speed is faster than that of the Java API because dynamic loading is performed using the native API through the JNI call. The core functions of the proposed scheme—wrapping and dropping—are described below.

3.1 Wrapping

The wrapping technique encrypts core bytecode selected for a class or method and hides it in the native module. It is decrypted by calling the stub code. The process of applying wrapping to the core bytecode selected in the original bytecode is shown in Figure 5.

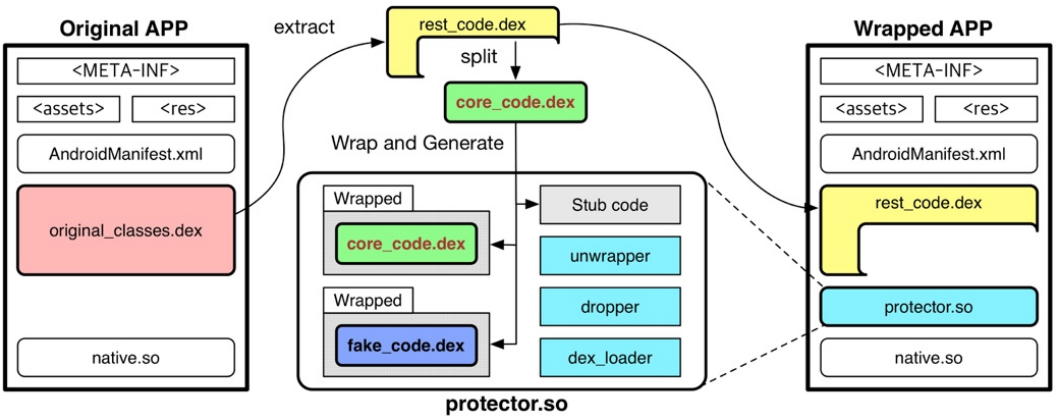

Figure 5. Wrapping process of the proposed scheme

First, we create the stub code that can call the selected bytecode from the original bytecode and the fake bytecode required for dropping. Now, the stub code and fake bytecode are generated with a dependency according to the core bytecode signature. Then, we wrap the core bytecode and fake bytecode with encryption and add it to the protection module along with the stub code. The protection module consists of an *unwrapper*, *dropper*, *dex_loader*, and the added code. The module is created as a native one and is included in the app. Finally, the remaining bytecode, except for the core bytecode, is regenerated to call the core bytecode, and it is included in the app.

When the core bytecode is called after the app is executed, it is loaded into memory through the unwrapping process, as shown in Figure 6. First, the native protection module is loaded using the JNI call in the remaining bytecode. The call is connected to the stub code. The stub code first calls the *unwrapper* to unwrap the wrapped core bytecode. Second, *dex_loader* is called to load the unwrapped core bytecode into memory. Finally, the core bytecode, which is loaded in memory, is called from the stub code.

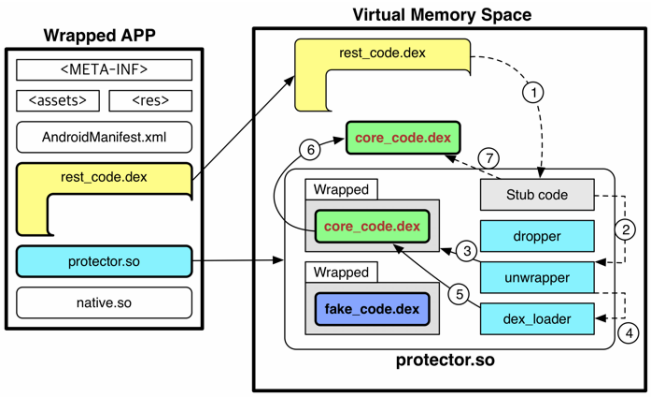

Figure 6. Unwrapping process of the proposed scheme

This process minimizes the exposed code size in memory. In addition, the following advantages are obtained as the core bytecode is encrypted in the native protection module and invoked through the JNI call. In general, developers use the framework's *DexClassLoader* class to dynamically load bytecode in Android apps. When using this API, it is easy to identify the dynamic loading and calling code of the wrapped core bytecode. If the core bytecode with protection is easy to identify, the analysis time is decreased, which is advantageous to the attacker. Thus, we add *dex_loader*, a native dynamic loading module that uses *dvm_Dalvik_system_DexFile*, the native API of the Dalvik VM, to make it difficult to identify the core bytecode. The remaining bytecode calls the wrapped core bytecode through a JNI call and the core bytecode is dynamically loaded into the memory via *dex_loader* after unwrapping by *unwrapper*. Because JNI calls are commonly used to improve the reusability and performance of native modules in Android apps, it is difficult to identify the proposed technique, which can delay the attacker's analysis time.

3.2 Dropping

The dropping technique ameliorates the disadvantages of existing packers, specifically the exposing of the original bytecode until the application is terminated. After the loaded core bytecode is completed through the wrapping process, it is individually unloaded from memory through the dropping process to decrease the exposure time.

As shown in Figure 7, dropping can minimize the exposure time by removing the used core bytecode from memory. Now, the fake bytecode generated in the wrapping process is loaded, and the stub code is linked to the fake bytecode. This fake bytecode has a signature like that of the core bytecode; however, it contains completely different logic, causing confusion in the dynamic analysis of the attacker.

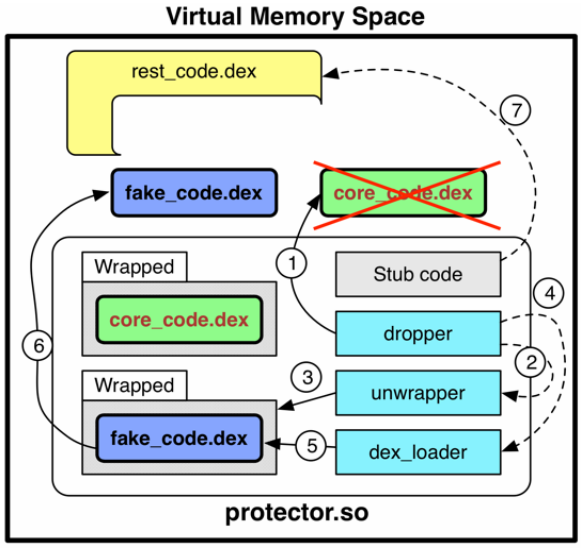

Figure 7. Dropping process of the proposed scheme

In the case of the signature, the Java language consists largely of classes and methods. Each class and method can be identified and converted by the type of each type or parameter. Now, it is converted by using the defined identification symbols. This is called "signature". The reason why this signature should be like fake bytecode and core bytecode is as follows. The first is to prevent the attacker from noticing that the proposed scheme has been applied, and the second is to make it easier to apply the proposed scheme.

4 Feature Comparison

In this section, we compare the advantages and disadvantages of the proposed scheme with those of the existing packing schemes. We compare the commonly used products—DexProtector and Ijiami—in terms of packer type identification, unpacking time, in-memory code identification, and in-memory code exposure. Table 1 lists and describes the features.

Table 1. Feature comparison

	DexProtector	Ijiami	Proposed Scheme
Packer type identification	Easy	Easy	Difficult
Unpacking time	Launch-time	Launch-time	Core code calling time
Original code identification	Easy	Difficult	Difficult
In-memory code exposure	Entire dex	Entire dex	Only working code (temporary exposure)

4.1 Packer Type Identification

The Android's packer changes the application entry point to a specific class to perform anti-analysis and unpacking, which restores the execution flow as the starting point of the original application. These classes commonly inherit and implement the *android.app.Application* class because the latter is the first execution on the Android system. In addition, it marks the earliest point at which the developers can intervene. Specifically, *Application* (DexProtector) and *Super Application* (Ijiami) classes inherit the *android.app.Application* and act as unpackers. Therefore, in these packers, the entry point is changed to a specific class and the files generated for performing this task are added to the *'/assets'* directory. These characteristics provide an opportunity for attackers to determine what methods are applied to protected applications. Accordingly, a specific attack method can be devised after only superficial inspection. However, the proposed scheme not only maintains the existing execution flow of the original application, but it also makes it difficult to recognize the application type because the core code call is the same as the native

method call of a general application.

Like the reverse engineering of other programs, reverse engineering an Android app usually involves static analysis and dynamic analysis. When an attacker reverse-engineer Android apps using existing commercial protection techniques, it is easy to identify them using the characteristics of each packer. This allows the attacker to perform a faster analysis by applying known analysis methods for each protection technique. However, if it is not possible to identify the application of the protection scheme or the type of technique as in the proposed scheme, it is not possible to use a known analysis method and the initial static analysis process is delayed because the start point cannot be set. Considering the attacker's analysis process, identifying the packer type is a good indicator of the performance of the packer.

4.2 Unpacking Time

A difference between the existing packing scheme and the proposed scheme is the protected code management unit where the management unit indicates the size of the exposure code. This unit determines the point at which unpacking occurs. Existing packing schemes unpack the entire code at once; thus, unpacking is inevitably performed at the start of the application. On the other hand, when the proposed scheme is applied, unwrapping is not performed at a certain time because it is managed by a class or method unit. Therefore, it is difficult for an attacker to designate an analysis point and recognize the method through static analysis.

4.3 Original Code Identification

All Android applications are forked and executed from the Zygote process. Moreover, they all have similar virtual memory spaces. In addition, the executable files and libraries of the target application are loaded in a similar area of memory. Thus, it is possible to search the memory space of a specific area and find the original bytecode by using the magic number and header information of the unpacked DEX file. In the case of Ijiami, it is difficult to find the original byte code simply by a magic number because the header information, including the DEX magic number, is mapped in memory, which is modified to protect against such analysis. However, after the application continues running, the entire bytecode is retained in memory. Thus, it is possible to obtain all bytecode by only seeking the location. On the other hand, the proposed scheme restricts the exposure time and size of the core bytecode by using wrapping and dropping, and it exposes fake bytecode when the core bytecode is not executed.

4.4 In-Memory Code Exposure

Ijiami packs the entire bytecode and unpacks it at the start of the application. DexProtector performs packing by class; however, like all packers, it unpacks all classes at the start of the application and loads them into the memory. For this reason, existing packers have a vulnerability in exposing the original bytecode to memory from the time that the application starts. However, in the proposed scheme, it is only exposed during the execution of the core bytecode, and it is replaced by fake bytecode when the execution is complete. Therefore, for an attacker to acquire all bytecode, it is necessary to capture the respective execution times of each core bytecode. Figure 8 compares the times of code exposure in memory.

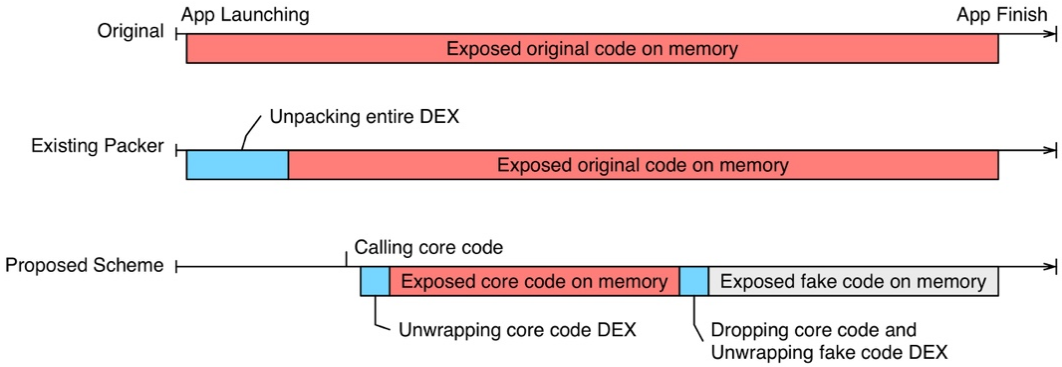

Figure 8. Comparison of core code exposure duration

5 Experiments

The core concept of the proposed scheme is to minimize the exposure of code in memory. Therefore, in this section, the code exposure of the proposed scheme is confirmed through static and dynamic analyses, and its performance is compared to those of the existing packers.

The experiment was performed on Android 4.4. Android Runtime (ART) has been newly introduced since version 4.4 and only ART has been supported since version 5.0 [4]. Considering these aspects, the proposed scheme uses APIs that are compatible up to version 7.1.

5.1 Experimental Setup

In the experiment, the proposed scheme was applied to a sample application. Then, static and dynamic analysis were performed to check whether the core bytecode was exposed. As shown in Figure 9, the sample application was created by adding a module that validates the serial key value to the Notepad application in the Google Android API sample. In this experiment, the *checkSerial()* function, which checks the validity, was classified as core code and the proposed scheme was applied to this part. Therefore, the main purpose of this experiment was to analyze whether the code of the *checkSerial()* function was exposed.

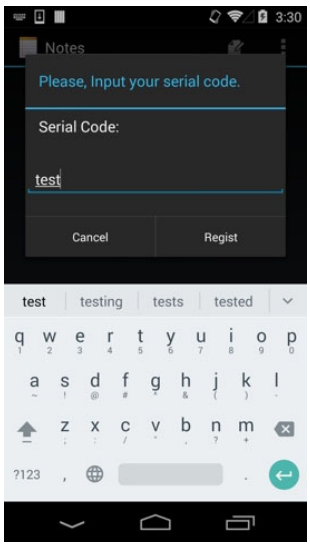

Figure 9. Sample app used for experiments

5.2 Code Exposure

We obtained the results for whether the code leaked when static and dynamic analyses were performed on the application with the proposed scheme. Static analysis decomposed the bytecode to determine whether core strings were exposed; dynamic analysis verified whether core code was exposed through a memory dump.

Static Analysis. Figure 10 and Figure 11 show the source code level by decompiling *classes.dex* of the protected application using the original scheme and the proposed scheme, respectively, without applying the proposed scheme. The tools used in decompilation were dex2jar (v.2.0) [19] and JD-GUI (v.1.4) [22].

```
package com.example.android.notepad;
public class Loader
{
public static final String isRegistString =
"isRegist";

  private static boolean checkSerial(String
paramString)
  {
    boolean bool = false;
    if(paramString.equals("msec")) {
      bool = true;
    }
    return bool;
  }

  public static Boolean registSerial(String
paramString)
  {
    return checkSerial(paramString);
  }
}
```

Figure 10. Decompiled source code before DexWrapper

```
package com.example.android.notepad;
public class Loader
{
public static final String isRegistString =
"isRegist";

static {
  System.loadLibrary("obfuscated");
}
  private static native boolean
checkSerial(String paramString);

  public static Boolean registSerial(String
paramString)
  {
    return checkSerial(paramString);
  }
}
```

Figure 11. Decompiled source code after DexWrapper

In the case of the original application, the contents of the *checkSerial()* function could be analyzed with a simple decompilation. However, in the case of the protected application, the *checkSerial()* function could check only the native method declaration for the JNI call. Furthermore, adding the *System.loadlibrary()* function for native library loading and dynamically loading the core bytecode was identical to the normal JNI call. Thus, it was difficult to identify the proposed scheme. Therefore, it was confirmed that the static code analysis could not identify the core code of the application to which the proposed scheme was applied.

Dynamic Analysis. To check whether the core bytecode of the application with the proposed scheme was exposed to memory, we dumped the memory after core code execution, as shown in Figure 12.

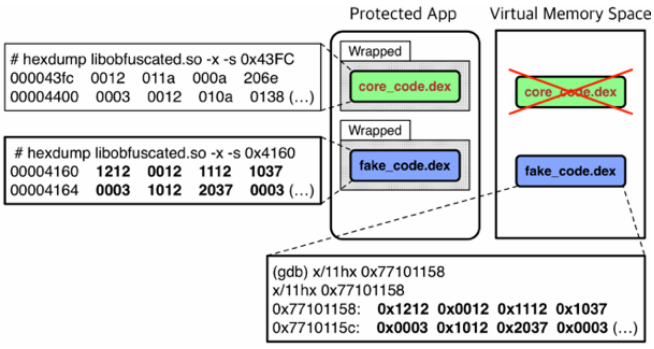

Figure 12. Result of memory dump

After executing the core code, the core bytecode was dropped from memory and could not be checked; only the fake bytecode was dumped. In other words, in the proposed scheme, both the core code and fake code were wrapped, and both were loaded into memory at the time of execution. At this point, the core code was immediately dropped from memory after execution; only the fake code remained in memory. Therefore, if the attacker attempted to perform dynamic analysis through a memory dump, only the fake code would be obtained. Thus, the experiment showed that the core code was more securely protected.

5.3 Performance

Table 2 shows the results of the comparison of the launch time, core code execution time, and file size overhead by applying DexProtector, Ijiami, and the proposed scheme.

Table 2. Performance comparison

	Launching time (ms)	Core code execution time (ms)		File size (Byte)
		CPU time	Actual processing time	
Original	338	265.144	276.609	65,106
DexProtector	742	304.678	354.461	910,275
Ijiami	506	276.306	296.027	979,809
Proposed scheme	255	278.738	394.767	87,175

Launching time. The launching time was measured by using Logcat to show the time when the main activity was displayed after the execution of the application. The times are shown in Figure 13. For DexProtector and Ijiami, the initial unpacking process required considerable time. However, in the case of the proposed scheme, the measurement time was shorter than that of the original. This is because the number of classes loaded at the beginning of the application was reduced as much as it was for the core bytecode separated by the proposed scheme.

```
07-05 14:04:35.394: D/NotesList(19348):
current timestamp:1467695075407
07-05 14:04:35.524: I/Adreno-EGL(19348):
<qeglDrvAPI_eglInitialize:320>:
EGL 1.4 QUALCOMM Build:
I0404c4692afb8623f95c43aeb6d5e13ed4b30ddb
Date: 11/06/13
07-05 14:04:35.554: D/OpenGLRenderer(19348):
Enabling debug mode 0
07-05 14:04:35.634: D/dalvikvm(19348):
GC_FOR_ALLOC freed 188K, 2% free
17018K/17240K, paused 17ms, total 17ms
07-05 14:04:35.694: I/ActivityManager(845):
Displayed com.example.android.notepad/.
NotesList: +383ms
07-05 14:05:00.094: D/audio_hw_primary(187):
select_devices: out_snd_device(2: speaker)
in_snd_device(0: )
07-05 14:05:00.154: D/dalvikvm(845):
GC_FOR_ALLOC freed 914K, 24% free
31838K/41428K, paused 57ms, total 57ms
```

Figure 13. Time required to display the main activity on Logcat

Core code execution time. In this experiment, the execution time of the *checkSerial()* function was measured by Eclipse's method profiling function. Thus, comparing the core code execution time, the existing packers and proposed scheme showed no difference in the CPU time based on the original. However, in the case of the actual processing time, the delay time of approximately 100 to 120 ms was measured for the proposed scheme. This is because the context switch process caused by the JNI call occurred twice (i.e., core bytecode and fake bytecode loading). This overhead is not a problem for practical use, compared to the problematic overhead of DexProtector.

Additional file size. Owing to the technical nature of the packing technique, additional files were added to increase the application size. This increase in file size also serves as overhead in applying real protection schemes. In this experiment, we measured the additional file size and found that the existing packers added an additional file that was significantly larger than the original file. In the case of the proposed scheme, we confirmed that the size increased partially because it was only applied to partial files.

6 Conclusion

Among existing mobile application protection schemes, the commercial packing technique can effectively protect against static analysis. However, the existing packing technique alone cannot provide sufficient dynamic analysis resistance. This is because the existing packing technique unpacks the entire original bytecode at the application startup and retains it in memory until the application terminated. To

overcome this drawback, the proposed scheme separates and wraps the core bytecode from the original bytecode into a method or class unit. The exposure size and time of this protected code is effectively reduced because it is removed from memory via dropping after the operation is completed at run-time. As a result, we conclude that the proposed scheme has a strong resistance to dynamic analysis compared to the existing packing methods and it shows excellent execution performance.

In the future, we plan to apply more categories of apps to compare performance with existing packers. Through these various experiments, we expect to see the practicality of the proposed scheme and broaden the application range.

Acknowledgments

This research was supported by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIT) (2017-0-00168, Automatic Deep Malware Analysis Technology for Cyber Threat Intelligence).

This is an extended version of a paper presented at MobiSec'16 in Taiwan, 2016 [6].

References

[1] H. Cho, J. Lim, H. Kim, J. H. Yi, Anti-debugging Scheme for Protecting Mobile Apps on Android Platform, *The Journal of Supercomputing*, Vol. 72, No. 1, pp. 232-246, January, 2016.

[2] J.-H. Jung, J. Y. Kim, H.-C. Lee, J. H. Yi, Repackaging Attack on Android Banking Applications and Its Countermeasures, *Wireless Personal Communications*, Vol. 73, No. 4, pp. 1421-1437, December, 2013.

[3] W. Yang, Y. Zhang, J. Li, J. Shu, B. Li, W. Hu, D. Gu, Appspear: Bytecode Decrypting and Dex Reassembling for Packed Android Malware, *Research in Attacks, Intrusions, and Defenses*, Kyoto, Japan, 2015, pp. 359-381.

[4] G. Na, J. Lim, K. Kim, J. H. Yi, Comparative Analysis of Mobile App Reverse Engineering Methods on Dalvik and ART, *Journal of Internet Services and Information Security*, Vol. 6, No. 3, pp. 27-39, August, 2016.

[5] J. Park, H. Kim, Y. Jeong, S.-j. Cho, S. Han, M. Park, Effects of Code Obfuscation on Android App Similarity Analysis, *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications*, Vol. 6, No. 4, pp. 86-98, December, 2015.

[6] Y. Park, T. Park, S. T. Kim, J. H. Yi, Mobile Code Packing Scheme based on Multi-partitioned Bytecode Wrapping, *Research Briefs on Information & Communication Technology Evolution*, Vol. 2, Article No. 12, September, 2016.

[7] W. Enck, D. Octeau, P. McDaniel, S. Chaudhuri, A Study of Android Application Security, *USENIX Security Symposium*, San Francisco, CA, 2011, p. 21.

[8] S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A.-R. Sadeghi, S. Brunthaler, M. Franz, Readactor: Practical Code Randomization Resilient to Memory Disclosure, *IEEE Symposium on Security and Privacy*, San Jose, CA, 2015, pp. 763-780.

[9] A. Henderson, A. Prakash, L. K. Yan, X. Hu, X. Wang, R. Zhou, H. Yin, Make It Work, Make It Right, Make It Fast: Building a Platform-neutral Whole-system Dynamic Binary Analysis Platform, *The ACM International Symposium on Software Testing and Analysis*, San Jose, CA, 2014, pp. 248-258.

[10] K. Lu, C. Song, B. Lee, S. P. Chung, T. Kim, W. Lee, Aslr-guard: Stopping Address Space Leakage for Code Reuse Attacks, *The 22nd ACM SIGSAC Conference on Computer and Communications Security*, New York, NY, 2015, pp. 280-291.

[11] X. Ugarte-Pedrero, D. Balzarotti, I. Santos, P. G. Bringas, Sok: Deep Packer Inspection: A Longitudinal Study of the Complexity of Run-time Packers, *IEEE Symposium on Security and Privacy*, San Jose, CA, 2015, pp. 659-673.

[12] Y. Zhang, X. Luo, H. Yin, Dexhunter: Toward Extracting Hidden Code from Packed Android Applications, *European Symposium on Research in Computer Security*, Vienna, Austria, 2015, pp. 293-311.

[13] Y. Park, *We Can Still Crack you! General Unpacking Method for Android Packer (No Root)*, in: Black Hat Asia 2015, Marina Bay, 2015.

[14] T. Strazzere, J. Sawyer, *Android Hacker Protection Level 0*, DEF CON 22, August, 2014.

[15] C. Collberg, C. Thomborson, D. Low, *A Taxonomy of Obfuscating Transformations*, Technical Report, July, 1997.

[16] Android Build Guide, https://developer.android.com/studio/build/index.html.

[17] Apktool, https://ibotpeaches.github.io/Apktool/.

[18] Bangcle, http://www.bangcle.com.

[19] dex2jar, https://bitbucket.org/pxb1988/dex2jar.

[20] Dexprotector, http://www.dexprotector.com.

[21] Ijiami, http://www.ijiami.cn.

[22] Jd-gui, http://jd.benow.ca/.

[23] Upx, http://upx.sourceforge.net.

[24] IDC Research, Smartphone OS Market Share, q2, http://www.idc.com/prodserv/smartphone-os-market-share.jsp.

Biographies

Yongjin Park received the B.S. and M.S. degrees in Computer Science and Engineering from Soongsil University in 2014 and 2016, respectively. He is now working for National Security Research in Korea. His research interests include mobile security, Android platform, and IoT .

Taeyong Park received the B.S. and M.S. degrees in Computer Science and Engineering from Soongsil University in 2015 and 2017, respectively. His research interests include mobile security, code obfuscation and system programming.

Jeong Hyun Yi is an Associate Professor in the School of Software and the Director of Cyber Security Research Center at Soongsil University, Seoul, Korea. He received the B.S. and M.S. degrees in computer science from Soongsil University, Seoul, Korea, in 1993 and 1995, respectively, and the Ph.D. degree in information and computer science from the University of California, Irvine, in 2005. He was a Principal Researcher at Samsung Advanced Institute of Technology, Korea, from 2005 to 2008, and a member of research staff at Electronics and Telecommunications Research Institute (ETRI), Korea, from 1995 to 2001. Between 2000 and 2001, he was a guest researcher at National Institute of Standards and Technology (NIST), Maryland, U.S. His research interests include mobile security and privacy, IoT security, and applied cryptography.