Android Application Protection against Static Reverse Engineering based on Multidexing

Nak Young Kim[1], Jaewoo Shim[1], Seong-je Cho[1], Minkyu Park[2*], and Sanghcul Han[2]
[1]Dankook University, Yongin, Gyeonggi 16890 Korea
{iuasdofil, 32131715, sjcho}@dankook.ac.kr
[2]Konkuk University, Chungju, Chungbuk 27478 Korea
{minkyup, schan}@kku.ac.kr

Abstract

DEX files are executable files of Android applications. Since DEX files are in the format of Java bytecodes, their Java source codes can be easily obtained using static reverse engineering tools. This results in numerous Android application thefts. There are some tools (e.g. bangcle, ijiami, liapp) that protect Android applications against static reverse engineering utilizing dynamic code loading. These tools usually encrypt `classes.dex` in an APK file. When the application is launched, the encrypted `classes.dex` file is decrypted and dynamically loaded. However, these tools fail to protect multidex APKs, which include more than one DEX files (`classes2.dex, classes3.dex, ...`) to accommodate large-sized execution codes. In this paper, we propose a technique that protects multidex Android applications against static reverse engineering. The technique can encrypt/decrypt multiple DEX files in APK files and dynamically load them. The experimental results show that the proposed technique can effiectively protect multidex APKs.

Keywords: Android, Reverse engineering, Multidex, Dynamic code loading, Packing

1 Introduction

In recent days, as the use of mobile devices (e.g. smartphones and smart watches) increases , the number of Android applications grows explosively. The number of Android applications is greater than 2 million in 2016 [7]. Protecting Android applications against software theft becomes one of important issues. Most Android applications are developed using Java programming language. Java source codes are compiled into bytecodes and unfortunately bytecodes can be easily decompiled using reverse engineering tools. Because of this weakness, the source codes of Android applications are exposed to software theft.

Many developers apply obfuscation techniques to hinder their applications from being exposed. The obfuscation techniques include identifier mangling, string obfuscation, inserting dead or irrelevant codes, and so on. Such obfuscation techniques, however, cannot protect source codes perfectly, but focus on making the source codes very complex and difficult to understand.

On the other hand, some tools (e.g. bangcle, ijiami, liapp) protect Android applications using code encryption and dynamic code loading [17, 18]. These tools encrypt DEX (Dalvik Executable) file in APK (Android application package), move the encrypted DEX file to other directory, and place a stub DEX file in the root directory of the APK. When the application is launched, the stub DEX decrypts and loads the encrypted DEX dynamically. However, there is a drawback in these tools; we find out that they fail to protect multidex applications. In multidex applications, there are more than one DEX files in an APK. Since the number of methods in a DEX file cannot exceed 65,536, large-sized applications can be

Journal of Internet Services and Information Security (JISIS), volume: 6, number: 4 (November 2016), pp. 54-64
*Corresponding author: Department of Computer Engineering, Konkuk University, 268 Chungwondaero, Chungju-si, Chungbuk-do, 27478, Korea, Tel: +82-43-840-3559

built as multidex applications. The above-mentioned tools encrypt only one DEX file or does not operate normally on multidex applications.

In this paper, we propose a multidex library based dynamic code loading technique for protecting Android applications against static reverse engineering. Multidex library is a library that Google developed to load more than one DEX files in Dalvik Virtual Machine (DVM). Based on this library, our technique encrypts/decrypts multiple DEX files, and loads them dynamically. Since the encrypted DEX files are decrypted only when the applications is launched in our technique, static reverse engineering of DEX files is very hard. The experimental results show that our technique can effiectively protect multidex APKs.

The rest of this paper is organized as follows. Section 2 explains the background and related work. Section 3 proposes our multidex-based dynamic code loading technique. Section 4 presents the experimental results and analysis. Finally, we conclude in Section 5.

2 Background and Related Work

2.1 Obfuscation Techniques

Most Android applications are developed using Java programming language. Developers can apply obfuscation techniques to their applications to make it difficult to reverse engineer them. There are various obfuscation techniques for Android applications as well as Java applications [15, 10] such as identifier mangling, string obfuscation, inserting dead or irrelevant codes, and so on.

Identifier mangling is to rename identifiers (the names of packages, classes, methods and variables) as meaningless characters. String obfuscation is to encrypt static-stored strings and decrypt them at runtime. Inserting dead or irrelevant codes is to inject codes that are never executed. The flow of original codes is changed, but the functionality of the program is not changed.

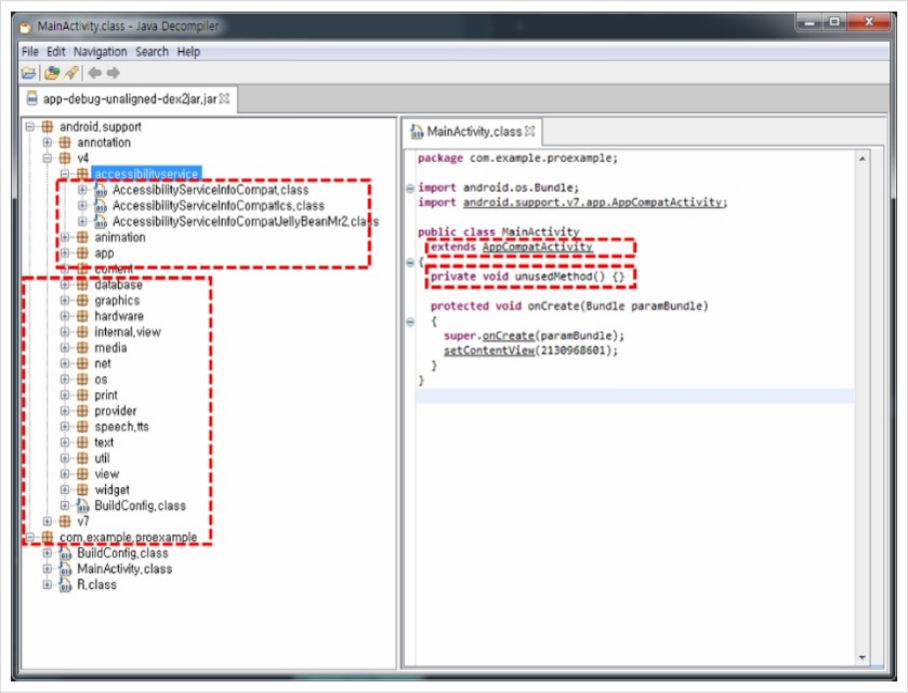

Figure 1: Code without Identifier magling

We can see `MainActivity` class inherits `AppCompatActivity` class in Figure 1. In contrast, looking at Figure 2, we see the name of `AppCompatActivity` class has been changed to `u` class. Changing name omits useful information that can be estimated from the name of identifier.

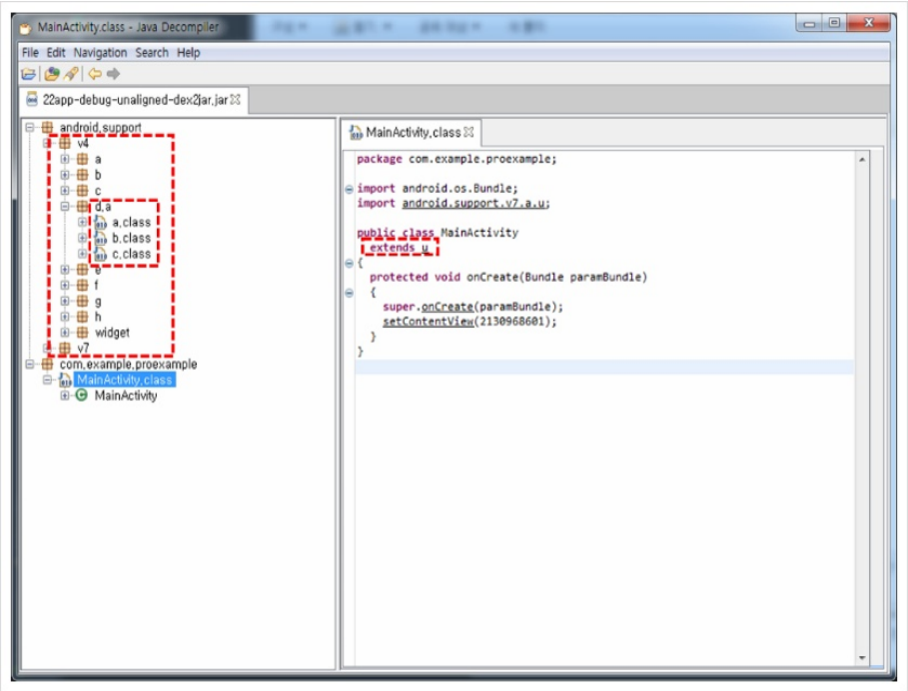

Figure 2: Code with Identifier magling

However, obfuscation techniques merely increase the time complexity for reverse engineering of applications. Seasoned hackers can disassemble APK into smali without decompiling it into Java. Obfuscation can not protect your app from smali attacks. If you think obfuscation is not working enough, you can write Android code in C ++ and compile your application using NDK. This method is not complete because it can also be disassembled. However, the code becomes more difficult to analyze or debug [11].

2.2 Multidex APK

Most Android application packages (APKs) contain *only one* DEX file that is executed on Dalvik VM . A DEX file is developed in Java programming language by developers. The Java source files and third-party libraries are compiled all together into a DEX file. A compiled DEX file defines various fields such as `string_ids`, `type_ids`, `field_ids`, `method_ids` and `class_defs` to indicate Dalvik bytecodes. The maximum number of referred method in a single DEX file is 65,536 ($= 2^{16}$), including framework methods, third-party library methods and user-defined methods [8]. If the number of methods in an application is greater than 65,536, APK file cannot be built. To overcome this limitation, developers have to eliminate unused methods using tools like Proguard, and/or employ lightweight libraries [14]. To make lightweight libraries, developers usually shrink frequently used third-party libraries. Developers also use a tool that counts the number of methods [13].

Google developed the multidex library, which enables APK files to contain *multiple* DEX files without changing the format of DEX file. The multidex library is available in build tools 21.1.1 or later [6]. Developers can build multidex APKs by setting options for multidex support in Android Studio or

Eclipse. In multidex APKs, DEX files are named as `classes.dex`, `classes2.dex`, and so on. If we decompile `classes.dex` using dex2jar and jd-gui [1], we can find package `android.support.multidex` that can load two or more DEX files. In order to find how many Android applications are built as multidex APKs, we downloaded popular 50 applications from Google Play. The number of multidex APKs among them is 14 (28%). Table 1 shows the multidex apps and their number of DEX files.

Table 1: Multidex apps and the number of dex files of each app

APK name	The number of dex files
com.beatpacking.beat-1.apk	2
com.chbreeze.jikbang4a-1.apk	2
com.findapp.yogiyo-1.apk	2
com.fitbit.FitbitMobile-1.apk	2
com.forshared-1.apk	4
com.nhn.android.band-1.apk	2
com.nhn.android.blog-1.apk	3
com.nhn.android.music-1.apk	2
com.nhn.android.navercafe-1.apk	2
com.skype.raider-1.apk	2
com.twitter.android-1.apk	3
jp.naver.line.android-1.apk	4
net.daum.android.cafe-1.apk	2
sixclk.newpiki-1.apk	2

2.3 Analysis of Existing Packers

We analyze existing Android packers [17, 18] such as bangcle, ijiami, and liapp. We also explore whether the packers can protect the secondary DEX file of multidex applications or not.

All packers above use `Application` class to load code dynamically [2]. `Application` class is a "base class for maintaining global application state". Packers replace this `Application` class with their custom class to control the execution flow as they want. While bangcle uses the same name for this custom class. ijiami and liapp modify the `name` attribute of `<application>` element in `AndroidManifest.xml` to their custom class. When apps start running, these custom classes are executed first and decrypt the encrypted DEX file. The custom classes, then, load the original DEX file dynamically using `DexClassLoader` API

We checked whether each packer protects DEX files from static reverse engineering attacks or not. We first examine *fitbit* app to see if we can statically reverse-engineer its `classes.dex` file. Figure 3 shows the structure of the original source codes and the decompiled source codes of the APKs packed by bangcle, ijiami, and liapp, respectively[18]. AS to bancle, you can see the structures are similar to the original source structure. On the other hand, as to ijiami and liapp, the structures are different. Bangcle packer does not change the `name` attribute of `<application>` element and also encapsulates Broadcast Receiver and Content Provider components [18]. We can see `classes.dex` file is protected from static reverse engineering attack in Figure 3.

We then explore whether `classes2.dex` file is protected from static reverse engineering attack or not. Figure 4 shows the source structure of the original `classes2.dex` and its decompiled codes. AS to bancle and ijiami, they maintain the same structure and some source codes are the same as the original

app. As to liapp, it has a different structure and some classes are missing, but the sources that belong to both apps are totally same.

Existing packers protect `classes.dex` files from static reverse engineering attack of multidex applications, but do not protect `classes2.dex` files. The hackers can reverse engineer the `classes2.dex` files and see the original source codes easily.

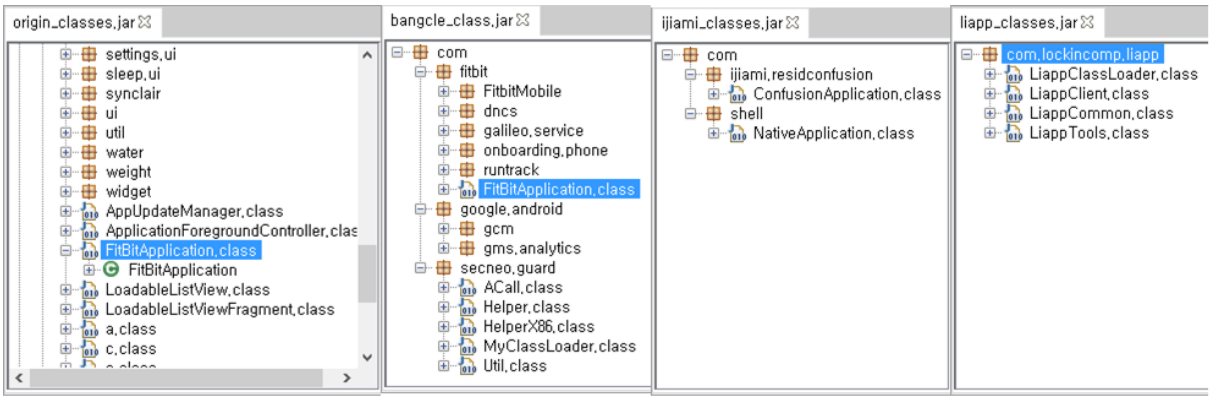

Figure 3: Decompiled `classes.dex` in Packed APK

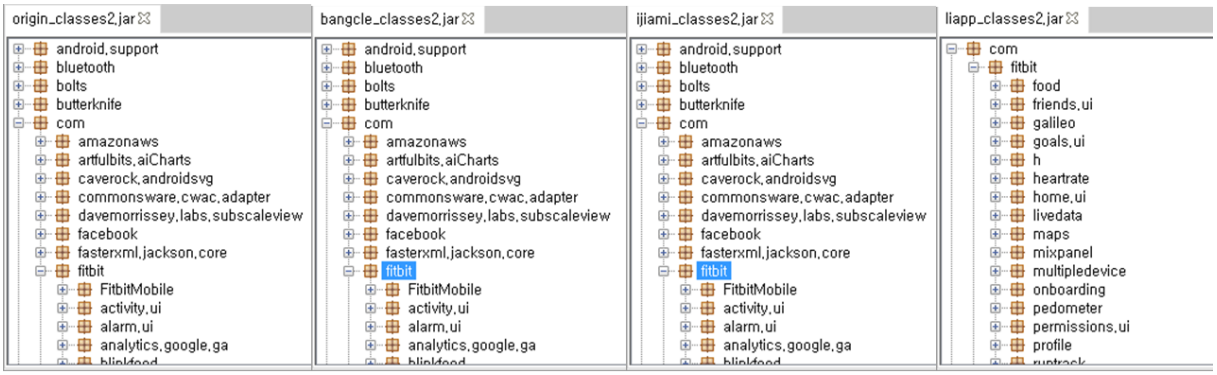

Figure 4: Decompiled `classes2.dex` in Packed APK

3 Proposed Method

As the size of recent Android applications grows, more and more Android applications adopt multidex technique. However, the existing Android packers cannot prevent static reverse engineering of multidex Android applications as explained in Section 2.3. We overview Google's multidex library in Section 3.1, and present a multidex library based dynamic code loading technique that can prevent reverse engineering of Android applications in Section 3.2.

3.1 Overview of Multidex Library

Dalvik loads `classes.dex` using the constructor of class `DexFile`. `DexFile` constructor is coded so that it loads only `classes.dex` from an APK file [3]. Google did not modify Dalvik to load multiple DEX files from an APK file because Android users may not update OS or venders may not support the

modification. Thus, Google developed multidex library [5] that can load more than one DEX files from an APK file without modifying Dalvik. The multidex library is embedded in `classes.dex` and is executed to load the remaining DEX files when application is executed. The `name` attribute of <application> element needs to be set `MultiDexApplication` or the parent class of the custom class need to be `MultiDexApplication`.

Figure 5 shows the flow of the execution of multidex library. Method `Multidex.install` invokes `MultiDexExtractgor.load`. This method stores `classes*.dex.zip` in the root directory of APK into directory `/data/data/package/code_cache/secondary-dexes`. After `classes*.dex.zip` are extracted from APK file and stored into the directory, `MultiDexExtractor.load` return an array that contains the full path of `classes*.dex.zip`. `MultiDex` class invokes method `makeDexElement` in class `DexPathList` using API reflection [4]. Method `makeDexElement` enables Dalvik to load additional DEX files, such as `classes2.dex, classes3.dex`, and so on. It returns a list of the additional DEX files to load. The list is added to `dexElements` of `DexPathList` class using `System.arraycopy` API. Eventually more than one DEX files are loaded in Dalvik and the application executes normally.

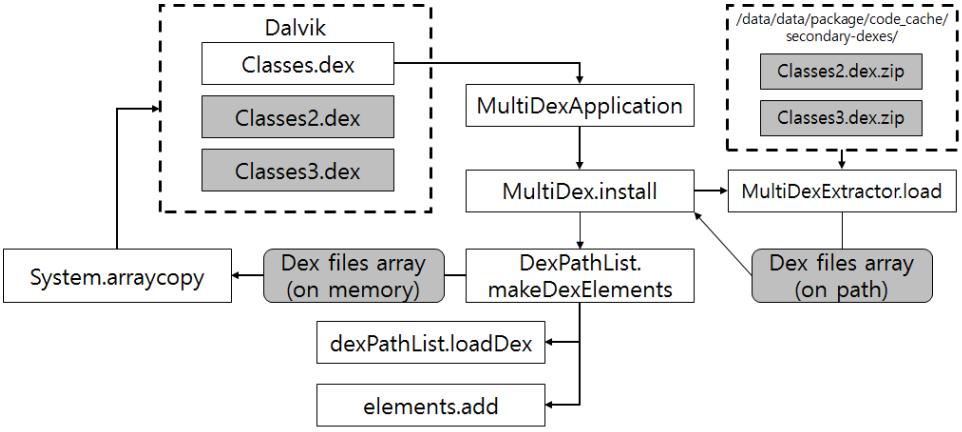

Figure 5: Execution Flow of Multidex Library

3.2 Dynamic Code Loading based on Multidex Library

This section proposes a dynamic code loading technique based on multidex library. As explained in Section 3.1, multidex library is executed first when an application starts. In our technique, all DEX files in the root directory in an APK file are encrypted using AES and moved to `asset` directory. When the application is executed, a stub DEX file (named as `classes.dex`) located in the root directory is executed first. It decrypts the encrypted DEX files and dynamically loads them. The AES decryption is implemented as JNI codes and the codes are stored as `libStub.so` in directory `lib`. Figure 6 shows the structure of APK files that are repackaged using our technique.

We implement the stub DEX file by modifying multidex library. We name the modified library as `com.dynamic.loading`. If we use the original name `android.support.multidex` in the stub DEX file, we encounter a pre-verification error in Dalvik which we can find out using `logcat`. The library extracts and decrypts the encrypted DEX files (named as `Enc1, Enc2, ...`) from directory `assets` instead of extracting `classes*.dex`. The process is as follows.

1. User launches an application.

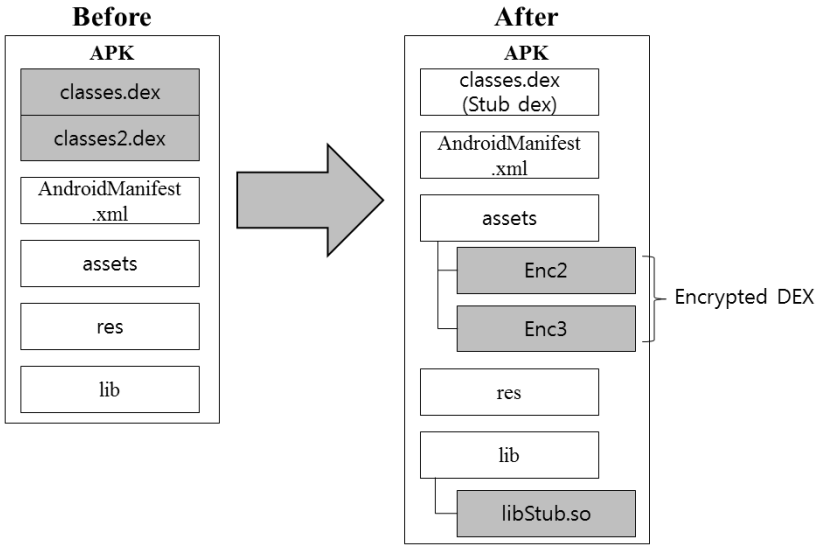

Figure 6: Structure of APK files with Encrypted DEX files

2. Dalvik loads and executes `classes.dex` (stub DEX file) in the application's APK file.

3. The modified multidex library is executed.

4. It extracts `Enc*` (AES-encrypted DEX files) in `assets` into `/data/data/package/code_cached /secondary-dexes`. The extracted files are compressed in ZIP format.

5. It decompresses the extracted files.

6. It decrypts the decompressed files by invoking a library in the native layer through JNI.

7. It dynamically loads the decrypted files in Dalvik.

4 Experimental Results

We packed multidex apps by applying our technique, then reverse engineered them using baksmali and dex2jar. We compared the decompiled sources and the original ones to find whether we can extract the original source codes from packed apps or not. The experiments were done on Android version 4.3 (JellyBean).

4.1 Static Reverse Engineering Attack Test

We first pack the original app using the proposed method and reverse engineer source codes from it using baksmali and dex2jar tools[16, 9, 12]. We can see `classes.dex` of packed app does not contain source classes in the original one, but only the classes of the stub DEX file. Next, we examine the encrypted original `classes*.dex` files (named as `enc*`). Their contents are shown in Figure 7. We cannot identify a DEX file because it does not have DEX file header "`dex\n035\0`"[9].

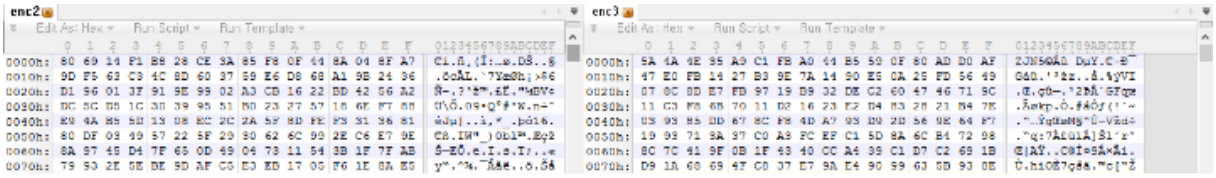

Figure 7: Encrypted Original DEX files

4.2 Dynamic Code Loading Overhead

We apply our technique to the applications listed in Table 1. We measure the time consumed in dynamically loading codes of the packed applications. We also instrument the time measurement codes into the original apps using baksmali [9] and simply repackage them. The results are shown in Figure 8.

The loading time of packed apps is longer than the original ones by 8∼19 seconds. The loading of packed apps consists of three steps: unzipping, encrypting, and zipping. We measure the time overhead of each step of a packed app `com.fineapp.yogiyo-1.apk`, as shown in Table 2. Unzipping and zipping takes longer by twice and five times than encrypting, respectively. It is because zipping and unzipping codes run in Java layer on Android.

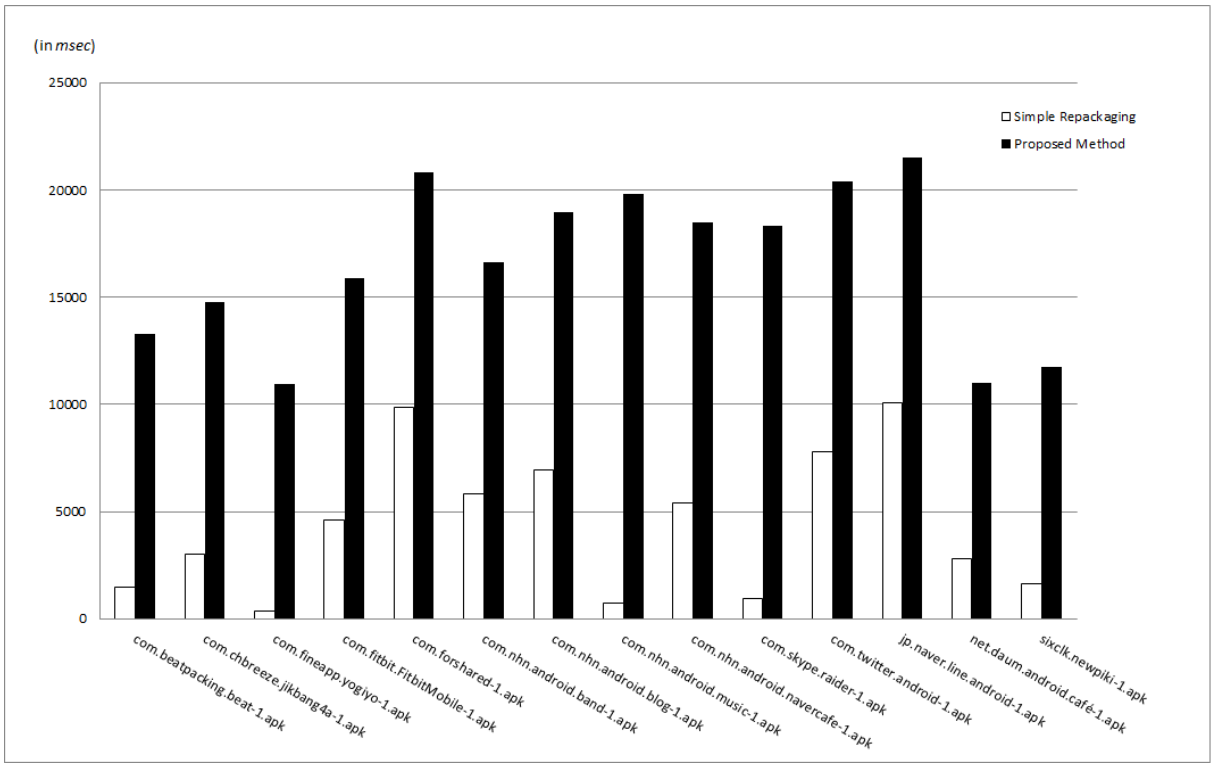

Figure 8: Time Overhead (in millisecond)

5 Conclusion and Future Work

In this paper, we propose a technique that protects Android application against static reverse engineering. Our technique encrypts DEX files in APK using AES encryption and adds a stub DEX in place of original

Table 2: Measured time for each step of com.fineapp.yogiyo-1.apk

Unzipping	Encrypting	Zipping
679ms	297ms	1632ms

`classes.dex` file. When the application is launched, the stub DEX decrypts and loads the encrypted DEX files dynamically. We implement the stub DEX by modifying Google's multidex library. We verify the effectiveness of the proposed technique using well-known reverse engineering tool such as dex2jar and baksmali. Different from other packers, our technique successfully protects multiple DEX files.

The weakness of our technique is the time overhead. The experiment results show that our technique incurs additional $8 \sim 19$ seconds in launching an application. We found out that this overhead is mainly due to ZIP compression/decompression in the Java layer. As future work, we plan to reduce the overhead by performing the task in the native layer.

Acknowledgments

This research was supported by the MSIP (Ministry of Science, ICT and Future Planning), Korea, under the ITRC (Information Technology Research Center) support program (IITP-2016-H8501161012000 1002) supervised by the IITP (Institute for Information & communications Technology Promotion). This research was also supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT and Future Planning (No. 2015R1A2A1A15053738)

References

[1] Java decompiler, 2008. `http://jd.benow.ca/` [Online; Accessed on November 10, 2016].

[2] Android Developers. Application. `http://developer.android.com/reference/android/app/Application.html` [Online; Accessed on November 10, 2016].

[3] Android Developers. Dexfile. `http://developer.android.com/reference/dalvik/system/DexFile.html` [Online; Accessed on November 10, 2016].

[4] Android Developers. Package java.lang.reflect. `http://developer.android.com/reference/java/lang/reflect/package-summary.html` [Online; Accessed on November 10, 2016].

[5] Android Developers. Support library features. `http://developer.android.com/tools/support-library/features.html` [Online; Accessed on November 10, 2016].

[6] Android Studio. SDK build tools release notes. `http://developer.android.com/tools/revisions/build-tools.html` [Online; Accessed on November 10, 2016].

[7] AppBrain. Number of android applications, 2016. `http://www.appbrain.com/stats/number-of-android-apps` [Online; Accessed on November 10, 2016].

[8] doyee. Building apps with over 64k methods, 2016. `http://developer.android.com/tools/building/multidex.html` [Online; Accessed on November 10, 2016].

[9] J. Freke. baksmali, 2009. `https://github.com/JesusFreke/smali` [Online; Accessed on November 10, 2016].

[10] S. Ghosh, S. R. Tandan, and K. Lahre. Shielding android application against reverse engineering. *International Journal of Engineering Research and Technology*, 2(6):2635–2643, June 2013.

[11] G. Nolan. Android obfuscation tools comparison, 2016. `http://riis.com/blog/android-obfuscation/` [Online; Accessed on November 10, 2016].

[12] B. Pan. dex2jar, 2014. https://sourceforge.net/projects/dex2jar/ [Online; Accessed on November 10, 2016].

[13] M. Parparita. dex-method-counts, 2014. https://github.com/mihaip/dex-method-counts [Online; Accessed on November 10, 2016].

[14] T. Reznik. Android and the dex 64k methods limit - contentful, 2014. https://www.contentful.com/blog/2014/10/30/android-and-the-dex-64k-methods-limit/ [Online; Accessed on November 10, 2016].

[15] P. Schulz. Code protection in Android. Technical Report 110, Rheinische Friedrich-Wilhelms-Universitat Bonn, 2012.

[16] C. Tumbleson and R. Wisniewski. Apktool, 2010. http://ibotpeaches.github.io/Apktool/ [Online; Accessed on November 10, 2016].

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

[18] R. Yu. Android packers: facing the challenges, building solutions. In *Proc. of the 24th Virus Bulletin International Conference (VB'14), Seattle, Washington, USA*, pages 266–275. IEEE, September 2014.

Author Biography

Nak Young Kim received the B.E in Dept. of Software from Dankook Univeristy in 2015. He is currently a master student in Dept. of Computer Science and Engineering at Dankook University, Korea. His research interests include computer system security, mobile security, and software protection.

Jaewoo Shim is an undergraduate student in the Dept. of Software at Dankook University, Korea. His research interests include computer system security, reverse engineering, and software vulnerability analysis.

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

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

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