Intelligent Automation And Soft Computing, 2019

Vol.25,no.1,143 –153

Protecting Android applications with multiple DEX files against Static Reverse Engineering Attacks

Kyeonghwan Lim[1], Nak Young Kim[1], Younsik Jeong[1], Seong-je Cho[1], Sangchul Han[2], and Minkyu Park[2]

[1]Dankook University, Yongin-si, Gyeonggi-do 16890 Korea
{limkh120, iuasdofil, jeongyousik, sjcho}@dankook.ac.kr
[2]Konkuk University, Chungju-si, Chungbuk-do 27478 Korea
{schan, minkyup}@kku.ac.kr

ABSTRACT
The Android application package (APK) uses the DEX format as an executable file format. Since DEX files are in Java bytecode format, you can easily get Java source code using static reverse engineering tools. This feature makes it easy to steal Android applications. Tools such as ijiami, liapp, alibaba, etc. can be used to protect applications from static reverse engineering attacks. These tools typically save encrypted classes.dex in the APK file, and then decrypt and load dynamically when the application starts. However, these tools do not protect multidex Android applications. A multidex Android application is an APK that contains multiple DEX files, mostly used in a large-scale application. We propose a method to protect multidex Android applications from static reverse engineering attacks. The proposed method encrypts multiple DEX files and stores them in an APK file. When an APK is launched, encrypted DEX files are decrypted and loaded dynamically. Experiment results show that the proposed method can effectively protect multidex APKs.

KEY WORDS: Reverse engineering, Android, Multidex, Packing, Dynamic code loading

1 INTRODUCTION

AS the use of mobile devices such as smart phones and smart watches increases, the number of Android applications is also growing. As of 2016, the number of Android applications exceeded 2 million (AppBrain, 2016). Most Android applications are being developed using the Java language. Java source codes are compiled into bytecodes and executed on a virtual machine. Unfortunately, by reverse engineering these bytecodes, one can easily get Java codes and Android applications are exposed to code theft. Developing methods to prevent unauthorized use of code is an important research topic.

Many developers apply obfuscation techniques to prevent exposure of application codes. Obfuscation techniques include identifier mangling, string obfuscation, and dead or irrelevant code insertion. However, these obfuscation techniques do not completely protect the source code. Their goal is to transform the source code into a form that is very complicated and difficult to understand.

Meanwhile, some tools such as ijiami, liapp and alibaba protect Android applications using code encryption and dynamic code loading (Yang et al., 2015; Yu, 2014). These tools encrypt a DEX (Dalvik Executable) file, move the encrypted DEX file to another directory, and place a stub DEX file in the APK root directory. When the application is executed, the stub DEX decrypts the encrypted DEX and dynamically loads it. However, we found the fact that these tools have a weakness: these tools do not protect multidex applications, which have multiple DEX files in its APK. Since one DEX file cannot have more than 65,536 methods, large applications should be implemented in a multidex form. The above tools do not work properly for multidex applications and encrypts only a single DEX file.

We propose a dynamic code loading method for protecting Android applications against static reverse engineering. This proposed method is based on the

CONTACT Minkyu Park minkyup@kku.ac.kr

Multidex library. The Multidex library is a library developed by Google to load more than one DEX files in Dalvik Virtual Machine (DVM). The proposed method encrypts / decrypts and dynamically loads multiple DEX files. Since the encrypted DEX files are decrypted only when the application is executed, it is difficult to statically reverse-engineer the DEX files. Experimental results show that the proposed method can effectively protect multidex APKs.

We have recently proposed a protection scheme for multidexing-based Android applications (Kim, Shim, Cho, Park, & Han, 2016). In (Kim, Shim, Cho, Park, & Han, 2016), the stub DEX is a weak point. Since the stub DEX was implemented in Java language, it is vulnerable to reverse engineering attacks. If an attacker reverse-engineers and tampers the stub DEX, he/she can obtain the decrypted original DEX files. In this paper, we extend the previous version as follows. First, we propose stub DEX integrity verification as a native library. This library is called by a stub DEX, and checks if the stub DEX is tampered. If so, the library terminates the application immediately in order to prevent further dynamic reverse engineer attacks. Second, we support multiple architectures. Since the decryption and integrity verification are implemented as a native library, the library is implemented for 7 types of architectures including armeabi-v7a and armeabi.

The remainder of this paper is organized as follows. Section 2 discusses background knowledge and related studies. In Section 3, we propose a multidex-library based dynamic code loading scheme. Section 4 shows the experimental and analytical results. Finally, we concluded in Section 5.

2 BACKGROUND AND RELATED WORKS

2.1Anti-Reverse Engineering Techniques

REVERSE engineering is a process of analyzing a program to identify and represent its components, structure and/or behavior. Reverse engineering can be used to understand how a program works and design a new program by improving the technology of the existing program. Meanwhile, reserve engineering also can be used to hack or tamper a program for illegal benefits. Especially Android applications are exposed to such software theft, because most Android applications are written in Java programming language and their Java source codes can be easily obtained using reverse engineering.

There are many studies on protecting Android applications against reverse engineering attacks. Pro-Guard (Android, 2016e) can shrink, optimize, obfuscate, and pre-verify Java class files. The ProGuard reduces APK file size by removing unused classes, fields, methods, and attributes. It optimizes the bytecode of the methods. The obfuscation step uses short meaningless names to change the names of the remaining classes, fields, and methods. Each of these steps is optional. For instance, ProGuard can be used to obfuscate the APK file only.

In (Schulz, 2012), the authors proposed several obfuscation techniques and dynamic code loading technique. In the dynamic code loading technique there are two components: an encrypted DEX file and a decryption stub (stub DEX). The original DEX file is encrypted and stored at somewhere in the APK. When the application is launched, the decryption stub decrypts the encrypted DEX file and loads it into Dalvik Virtual Machine (DVM) to execute.

In (Ghosh, Tandan, & Lahre, 2013), the authors added few more logic on Android source code to make it more complex to understand. They exploit try-catch blocks to change the control flow at runtime. They exchanged the role of try-catch blocks. They put business logic into catch block and dummy code, meaningless loop, and senseless if-else condition into try block. Also, they put one line of code that always cause exception at runtime. Reverse engineers try to understand the try block but it has meaningless code. This makes understanding code more complex.

In (Xu, Zhang, Sun, Lin, & Mao, 2015), the authors presented a technique that can detect debug state and debug environment by detecting debugging-related processes and emulator specific features. On the basis of the results, they proposed some solutions to prevent Android applications being decompiled and cracked. Taking advantage of these methods, they mostly eliminate the feasibility of the secondary packaging for Android software.

In (Sun, Cuadros, & Beznosov, 2015), the authors evaluate rooting detection techniques. They found many techniques used by apps to detect rooted devices, but all rooting detection methods studied can be evaded. They concluded that Android OS can only provide a reliable rooting detection method. OS also implements rooting detection logic in the trusted components, such as integrity-protected kernels or external trusted execution environments.

Lim et al. (Lim et al., 2016) proposed an anti-dynamic reverse engineering scheme for Android applications on real smart phones, where its stub DEX tries to detect dynamic reverse engineering attacks. Especially they focus on call stack-based evasion attack (API hooking) detection. If the stub DEX does not detect any dynamic reverse engineering attack it loads the original DEX file dynamically.

Kim et al. (Kim, Shim, Cho, Park, & Han, 2016) proposed a technique for protecting multiple DEX Android applications. The technique is similar to Schulz's one in the sense that the original DEX files are encrypted and the stub DEX decrypts/loads them dynamically. However, Kim's technique supports multiple DEX Android applications.

In (Na, Lim, Kim & Yi 2016), the authors summarized the difference of attacks to DEX files and

OAT files. DEX is for Dalvik Virtual Machine and OAT for ART environment. The authors explored static and dynamic analysis technique for DEX and OAT files on Android 4.0 (KitKat). Note that, in our work, we tried to prevent static attacks for DEX files, but not for OAT files.

2.2 Multidex APK

Most Android application package (APK) is the file format for packages for distribution and installation of applications on Android platform. An APK file contains manifest files, architecture-dependent native codes, resources, DEX files, etc. In most APKs, there is one DEX file, classes.dex, which is loaded and executed on DVM. Android application developers usually write Java source programs and compile them all together with third-party libraries into a DEX file.

One problem in the build process is that the number of referred methods in a single DEX file is limited to 65,536 (= 216), including user-defined methods, third-party library methods and framework methods (Android, 2016b). This is because invoke-* Dalvik instruction takes a 16-bit argument which represents the reference index of the target method. Thus, if the number of methods in classes.dex exceeds 65,536, the build process fails. To remedy this problem, developers shrink codes using tools such as ProGuard and/or adopt lightweight libraries (Reznik, 2014). A tool that counts the referred methods is also used (Parparita, 2014).

To get around this limitation, Android provides multidex library. With this library an APK file can contain multiple DEX files without changing the DEX file format. The multidex library is available in build tools 21.1.1 or later (Android, 2016f). When developers build an APK, they switch on the multidex support option in Android Studio or Eclipse. In multidex APKs, the names of DEX files are classes.dex, classes2.dex, and so forth. If classes.dex in a multidex APK is decompiled using dex2jar and jd-gui (Java Decompiler, 2008), android.support.multidex package is found. We collected and decompiled popular 40 Android applications from Google Play. We found out that 22 (55 %) APKs among them are multidex APKs. The Table 1 lists 19 apps among them, which the proposed method is successfully applied to.

Table 1. 19 Multidex APK

APK name
com.snapchat.android-1.apk
com.infraware.office.link-1.apk
com.toxic.apps.chrome-1.apk
com.myfitnesspal.android-1.apk
com.twitter.android-1.apk
com.zhiliaoapp.musically-1.apk
jp.naver.line.android-1.apk
com.skype.raider-1.apk
mobi.byss.instaweather.watchface-1.apk
com.undertap.watchlivetv-1.apk
com.talkatone.android-1.apk
com.estrongs.android.pop-1.apk
net.zedge.android-1.apk
air.com.officemax.magicmirror.ElfYourSelf-1.apk
com.mcentric.mcclient.FCBWorld-1.apk
com.viber.voip-1.apk
kik.android-1.apk
com.fitbit.FitbitMobile-1.apk
com.pinterest-1.apk

2.3 Analysis of Android Packers

An Android packer is a tool that protects Android applications by encrypting, decrypting and dynamically loading classes.dex (Yang et al., 2015; Yu, 2014). In this section, we analyze three Android packer services, ijiami, liapp, and alibaba, to explore whether they can protect multidex APKs. All these packers deploy Application class for dynamic code loading. “Application class is a base class for maintaining global application state” (Android, 2016a). When an APK is packed, Application class (or its subclass) is replaced with each packer’s custom class to control the execution flow. Thereby, the name attribute of <application> element in AndroidManifest.xml is modified to their custom class’ name. When a packed application is launched, its custom class is executed first. It decrypts the encrypted DEX file, and then loads the decrypted DEX file dynamically using DexClassLoader class.

We investigate whether each packer can protect multiple DEX files from static reverse engineering. First we pack com.twitter.android-1.apk using the three packers. We unzip the original APK and the packed APK, respectively, to obtain classes.dex. Then we decompile each classes.dex using dex2jar (Pan, B) and gd-gui (Java decompiler) to obtain Java source codes, and examine their code trees. As shown in Figure 1, the code trees are different from each other. We can find the name of the custom class for each packer. Figure 1 implies that the three packers can protect classes.dex from static reverse engineering.

Now we examine the code trees of the decompiled classes2.dex. As shown in Figure 2, the code trees of the packed APKs are identical to the original APK. Furthermore, some decompiled codes are the same as the original ones. Figure 2 implies that the three Android packers do not protect classes2.dex at all from static reverse engineering.

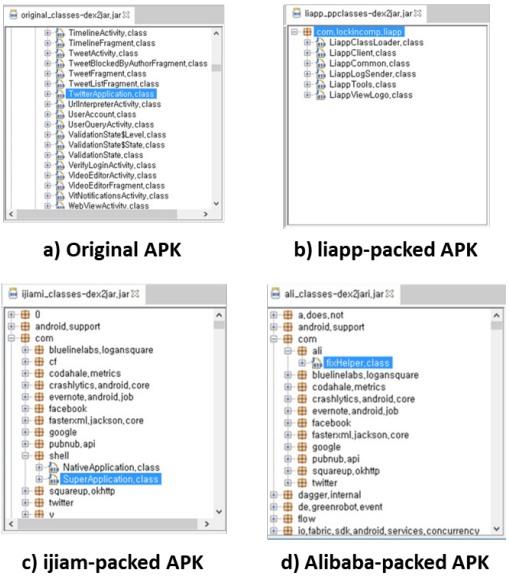


Figure 1. Code tree of decompiled `classes.dex`.

Figure 2. Code tree of decompiled classes2.dex.

There are commercial packers such as DexProtector and DexGuard. Since these tools do not provide a free or trial version, we could not examine whether they can protect multidex APKs. According to DexProtector's documentations, DexProtector supports multidex APKs. However, they do not describe how DexProtector protects multiple DEX files. Meanwhile, DexGurard's documentations do not mention multidex support.

2.4 Overview of Multidex Library

Dalvik loads classes.dex using the constructor of the DexFile class. The DexFile constructor loads only classes.dex in APK file (Android, 2016c). Google does not modify the Dalvik to provide the functionality to load multiple DEX files from an APK file. This is because the Android users may not update the OS or the vendors may not support these modifications. Instead, Google offers a Multidex library (Android, 2016g) that enable Android to load multiple DEX files from your APK. The Multidex library is built into classes.dex and executed to load the rest of the DEX files when the application runs. The name attribute of the <application> element in AndroidManifest.xml must be set to MultiDexApplication class or a user-defined subclass of MultiDexApplication.

Figure 3 shows the execution flow of the Multidex library. The MultiDex.install method calls the MultiDexExtractgor.load method which extracts DEX files (classes*.dex) from the APK's root directory, compresses and saves them as classes*.dex.zip in the /data/data/package/code_cache/secondary-dexes directory, and returns an array containing the full paths of classes*.dex.zip. Then MultiDex.install passes the array to the installSecondaryDexes method which invokes a build-platform-specific install method. The install method invokes in turn the makeDexElements method of the DexPathList class using API reflection (Android, 2016d). The makeDexElement method allows additional DEX files such as classes2.dex, classes3.dex, etc. to be loaded. Finally, more than one DEX files are loaded and the application runs normally.

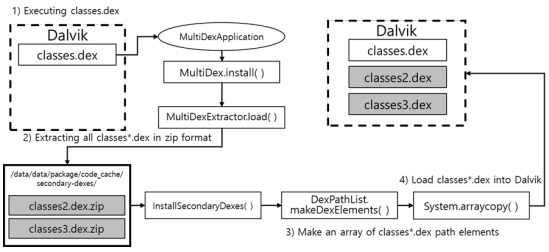


Figure 3. Execution Flow of Multidex APK.

3 PROPOSED METHOD

OUR anti-static reverse engineering method for multidex Android applications is embedded in applications. Figure 4 shows the overview of the proposed method. Our method consists of several techniques. In Java layer, DEX file substitution, dynamic code loading for multidex APK and entry point change are implemented. In native C layer, DEX

file decryption and stub DEX integrity verification are implemented. When a packed APK is launched in a user device, these techniques are performed in the following order: (1) stub DEX integrity verification, (2) DEX file decryption, (3) dynamic code loading for multidex APK, (4) entry point change, (5) the original DEX execution.

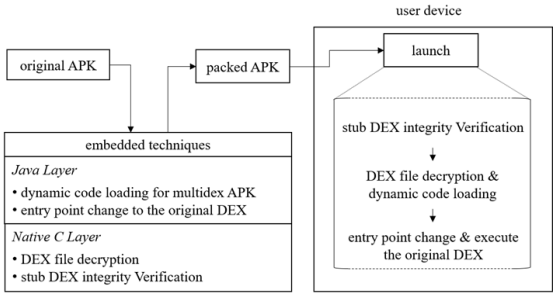


Figure 4. Overview of Proposed Method.

Figure 5 shows the structure of an APK before/after packing using our method. In Figure 5(b), gray boxes represent modified files. The original DEX files are encrypted during packing and stored as encrypted_classes.dex, encrypted_classes2.dex, and so on. Instead of the original classes.dex, a stub DEX file, named as classes.dex, is added to the packed APK. The stub DEX implements the Java layer; it calls native codes, loads the decrypted DEX and changes the entry point to the DEX. The name attribute of <application> element in AndroidManifest.xml is changed to the package name of the stub DEX so that the stub DEX can be executed first. The native C layer is implemented as libStub.so file, which is loaded on memory at the start of execution. By invoking functions in libStub.so, the stub DEX verifies the integrity of itself and decrypts encrypted_classes*.dex file.

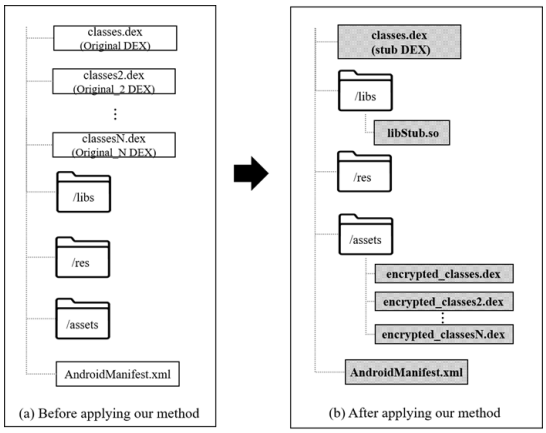


Figure 5. Structure of APK files before/after Packing.

3.1 Verifying Integrity of Stub DEX

In our proposed method, since stub DEX must be executed first in the application, we need to set the value of the name attribute of the <application> element in the AndroidManifest.xml file to the package name of stub DEX.

To begin with, the stub DEX must verify its own integrity. Because stub DEX is implemented in Java language, it is vulnerable to reverse engineering attack compared with C implementation of Android native library layer. Unless the integrity of stub DEX is verified, the following problems may occur. Attackers can prevent normal execution of applications by modifying the decryption codes of the stub DEX. The stub DEX calls an Android native function to decrypt the encrypted original executable code. If the codes that call the functions of this native layer are deleted, the original app code will not be decrypted and the app will not run normally.

To make matters worse, if you modulates codes that decrypt the original DEX file and load it into memory, attackers may get the original code. He can also tamper with this part to load and run malware. Therefore, it is necessary to verify the integrity of the stub DEX itself. Upon passing the verification process, multidex decryption and dynamic loading are performed.

Integrity verification is performed as follows. When a server applies the proposed scheme, it obtains the SHA1 hash value for a stub DEX file, encodes the obtained SHA1 hash value into Base64 string, and stores it in libstub.so. Then, when the developer signs the application, the server stores the Base64 string of SHA1 of the stub DEX in the signature-related file MANIFEST.MF and distributes afterwards. The MANIFEST.MF file exists in the META-INF directory and stores information on all files in the APK package except the files in META-INF directory. Information include the file path and the Base64 encoded string of SHA1 hash value of each file. When the application is executed, libstub.so is loaded and Base64 string of stub DEX stored in libstub.so is compared with the Base64 string stored in MANIFEST.MF.

3.2 Multidex Library based Dynamic Code Loading

This section proposes a dynamic code loading technique based on multidex library. As explained in Section 2.4, 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 (Advanced Encryption Standard) 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 5 shows the structure of APK files that are repackaged using our method.

We implement the stub DEX 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, we encounter a pre-verification error because Dalvik tries to verify and load android.support.multidex twice; in the stub DEX and the original classes.dex. We can find it out using logcat.

Our library extracts and decrypts the encrypted DEX files (named as encrypted_classes.dex, encrypted_classes2.dex, ...) from directory assets instead of simply extracting classes*.dex. The process is as follows.

1. A user launches an application.
2. Stub DEX (classes.dex) is loaded and executed on Dalvik.
3. It verifies the integrity of itself by invoking libStub.so through JNI.
4. It extracts encrypted_classes*.dex (AES-encrypted DEX files) in assets into /data/data/package/code_cache/secondary-dexes directory.
5. It decrypts the encrypted_classes*.dex files by invoking libStub.so through JNI.
6. It compresses the decrypted files in the ZIP format.
7. It loads the compressed files on Dalvik.
8. It deletes the compressed files

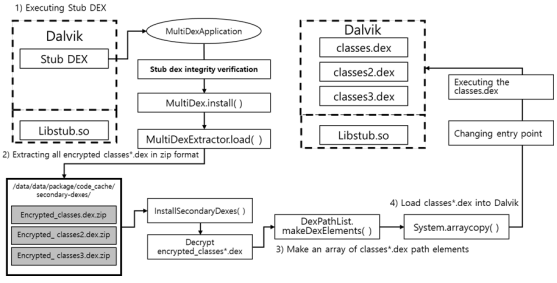


Figure 6. Execution Flow of Multidex Library based Dynamic Code Loading.

3.3 Changing Entry Point & Executing the original DEX

In order that the application may execute normally, the stub DEX needs to configure the execution environments as if the original DEX were executing (Lim, Jeong, Lim, et al., 2016). The last task of the stub DEX is to change the entry point to a proper method of the loaded DEX. The process is as follows. (1) The original DEX files are already loaded on Dalvik. (2) Fill the fields of loadedapk object with the information of the original application. The loadedapk object maintains the information of the current application such as ClassLoader, ActivityThread and ApplicationInfo. The original DEX can execute normally by using this object. (3) Invoke makeApplication() which in turn invokes onCreate() of the initial activity of the application.

3.4 Multiple Architecture Support

Google provides a number of Application Binary Interfaces (ABIs) that allow Android handsets to be manufactured using a variety of CPUs (Wikipedia, 2016). ABI is a low-level binary interface used between the application and the operating system or application and its library, or finally the components of the application.

The ABI is distinguished from the API (Application Programming Interface). The API is used at the source code level, and the ABI is used at the binary code level. These ABIs can be cross-compiled using NDK, and currently there are seven kinds of ABIs supported. A detailed description of each ABI can be found on the Google developer site (Google, 2016).

Since the proposed method adds a native library that is responsible for integrity verification of the stub DEX file itself and decryption of the original DEX files, it must support various architectures as mentioned above. Therefore, the proposed method inserts the corresponding native library for each architecture supported. Among them the most commonly used ABIs are armeabi-v7a and armeabi.

4 EXPERIMENTAL RESULTS

IN this section, we verify that the proposed **method** can protect multidex applications against reverse engineering and measure performance overhead of the method by carrying out experiments. Our experiments are conducted on Google Nexus 7 using the multidex applications listed in Table 1. The specifications of Google Nexus 7 are shown in Table 2.

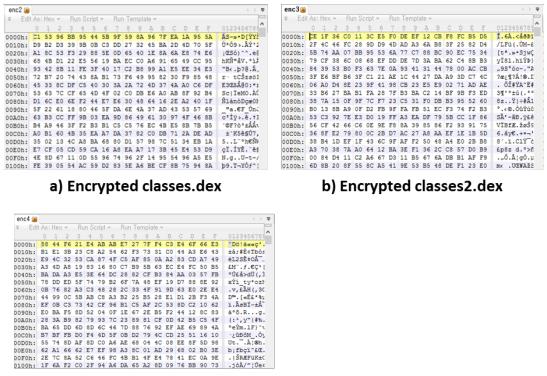
Table 2. Specification of Google Nexus 7

Operating System	Android 4.3(Jelly Bean)
Kernel Version	3.4.0-g6537a16
CPU	1.51 GHz quad-core Krait 300
Memory	2GB DDR3L RAM
Storage	16 GB

4.1 Static Reverse Engineering Attack Test

For the experiment, we pack a sample original application using our proposed method and create its packed version, that is, the packed application. We then reverse engineer the packed version using baksmali and dex2jar tools (Tumbleson & Wisniewski, 2010; Freke, 2009; Pan, 2014). We can analyze classes.dex of the packed version. It contains only the classes of the stub DEX file introduced by our proposed method and does not contain the classes of in the original application. Next, we examine the

encrypted_classes*.dex files (shortly named as Enc*) in the packed version. Their contents are shown in Figure 7. It is impossible to identify any original DEX file from the encrypted_classes*.dex files because none of them has DEX file header "dex\n035\0" (Freke, 2009). Therefore, we verify that our proposed method can protect Android applications against static reverse engineering attacks.



a) Encrypted classes.dex b) Encrypted classes2.dex

c) Encrypted classes3.dex

Figure 7. Contents of *Enc files of the packed application, *com.twitter.android-1.apk*.**

4.2 Dynamic Code Loading Overhead

AT the beginning of an application execution, the procedure for the dynamic code loading technique is described in Section 3.2. In case of the packed application with our proposed method, the dynamic code loading is performed after verifying the integrity of the stub DEX. During the dynamic code loading, our method decrypts the encrypted_classes*.dex files and compresses the decrypted files one by one to the ZIP format. Then, the compressed files are dynamically loaded on Dalvik using DexPathList shown in Figure 3. Therefore, we measure the performance overhead of our method as follows.

First, we calculate the processing time for dynamically loading code of an original application and its packed version with our proposed method, respectively. By repeating the calculation ten times, we measure the average processing time. Figure 8 shows the average processing times side by side which are spent for dynamically loading the codes of the original applications and their packet versions. The system.currentTimeMillis () method, which returns the current time, is used to measure the total time taken for dynamic loading, integrity verification, decryption time, compression time, DexOpt, and changing entry point time. To measure the processing time of each element, the time difference was calculated by placing the method at the start and end. For dynamic code loading, the packed applications with our method takes more time than their original ones ranging from minimally 1.46 times to maximally 5.9 times.

The processing time for dynamically loading the codes of the packed applications can be broken into five major components: integrity verification time, decryption time, compression time, the execution time of the DexOpt program and changing entry point time. We analyze the major components of the processing time in detail and show the results in Figure 8. And Table 3 lists the number and size of the encrypted_classes*.dex files contained in each packed application. DexOpt is a system-internal program that is used to produce optimized DEX files. DexOpt loads, verifies and optimizes DEX files. It performs these tasks in favor of Dalvik VM to avoid allocating some resources to Dalvik. Instead, those resources are allocated to DexOpt and it frees the resources on completion of the tasks. The app with the proposed scheme will run stub DEX first. The original classes*.dex files are then executed through a changing entry point technique. The Android system will perform bytecode optimization to run the original DEX file. The time it takes to optimize is DexOpt time. DexOpt is usually invoked once when the original applications as well as their packed versions are executed at the first time. The run time of the DexOpt is short in case of the execution of the original application because the corresponding symbol information can be obtained directly, while the run time of the DexOpt is very long in case of the execution of the packed version because the corresponding symbol information cannot be obtained directly. From the message of the logcat tool, our guess is that APK repackaging of our method have caused the mapping information of the symbols to be broken. As the result of the repackaging, the symbol information and method overriding should be recovered during the unpacking and dynamic loading the codes of the packed application.

In Figure 8, we can see that the time for verifying stub DEX integrity is almost steady. However, the time for decryption and compression is increased in proportion to the size and number of encrypted_classes*.dex files. During the execution of the packed applications, the DexOpt program is invoked. The invocation to the DexOpt is an additional overhead incurred by the proposed method. Among the sample applications used in our experiments, the com.infraware.office.link-1.apk includes four encrypted_classes*.dex files and requires the largest time, 19.66 seconds, for integrity verification, decryption, compression and the invocation of the DexOpt. On the one hand, the mobi.byss.instaweather.watchface-1.apk includes two encrypted_classes*.dex files and requires the smallest time, 6.80 seconds, for integrity verification, decryption, compression and the invocation of the DexOpt.

Table 3. The number of the encrypted DEX files and their size

APK name (its extension is .apk)	# of Encrypted DEX files	Encrypted Dex file size(MB)
com.snapchat.android-1	4	23.00
com.infraware.office.link-1	4	29.60
com.toxic.apps.chrome-1	3	15.03
com.myfitnesspal.android-1	3	20.70
com.twitter.android-1	3	18.10
com.zhiliaoapp.musically-1	3	17.70
jp.naver.line.android-1	3	22.80
com.skype.raider-1	2	11.80
mobi.byss.instaweather.watchface-1	2	9.04
com.undertap.watchlivetv-1	2	11.60
com.talkatone.android-1	2	10.90
com.estrongs.android.pop-1	2	15.20
net.zedge.android-1	2	11.80
air.com.officemax.magicmirror.ElfYourSelf-1	2	11.50
com.mcentric.mcclient.FCBWorld-1	2	16.00
com.viber.voip-1	2	14.10
kik.android-1	2	14.60
com.fitbit.FitbitMobile-1	2	15.20
com.pinterest-1	2	12.60

4.3 Discussion

LIAPP, ijiami, alibaba, etc. are representative tools for preventing static reverse engineering attacks. These tools usually encrypt and save classes.dex in the Android app in a similar way to the proposed technique. The encrypted classes.dex file is decrypted and loaded when the app launches. However, these tools only support apps with a single DEX file, and we showed that you cannot defend all classes.dex files if more than one classes.dex file exists in Section 2.3.

Table 4. Comparison of app protection tools with respect to functionality

	Liapp packer	ijiami packer	Alibaba packer	The proposed method
What it can prevent	static reverse engineering attacks	static reverse engineering attacks	static reverse engineering attacks	static reverse engineering attacks and tempering
Support multidex	X	X	X	O

We collected 22 multidex apps from the Google Play. The proposed method is successfully applied to 19 of them. Table 3 shows these 19 applications with their number of DEX and size. We cannot apply the proposed method to three apps because of verifying self-integrity and repacking. If an app verifies its integrity itself, our modification to the app cannot pass this integrity verification. Such cases are com.supo.security-1.apk and com.qihoo.security-1.apk. In case of com.tencent.mm-1.apk, we cannot repack the app because the version of apktool is incompatible with the app.

We assume that AES keys are managed properly and safely, and that AES is secure. Since our technique encrypts/decrypts the DEX files using AES, its application protection capability against static reverse engineering depends on the security of AES, which is beyond the scope of this paper. Our technique can protect applications against all static reverse engineering attacks as long as the assumption holds.

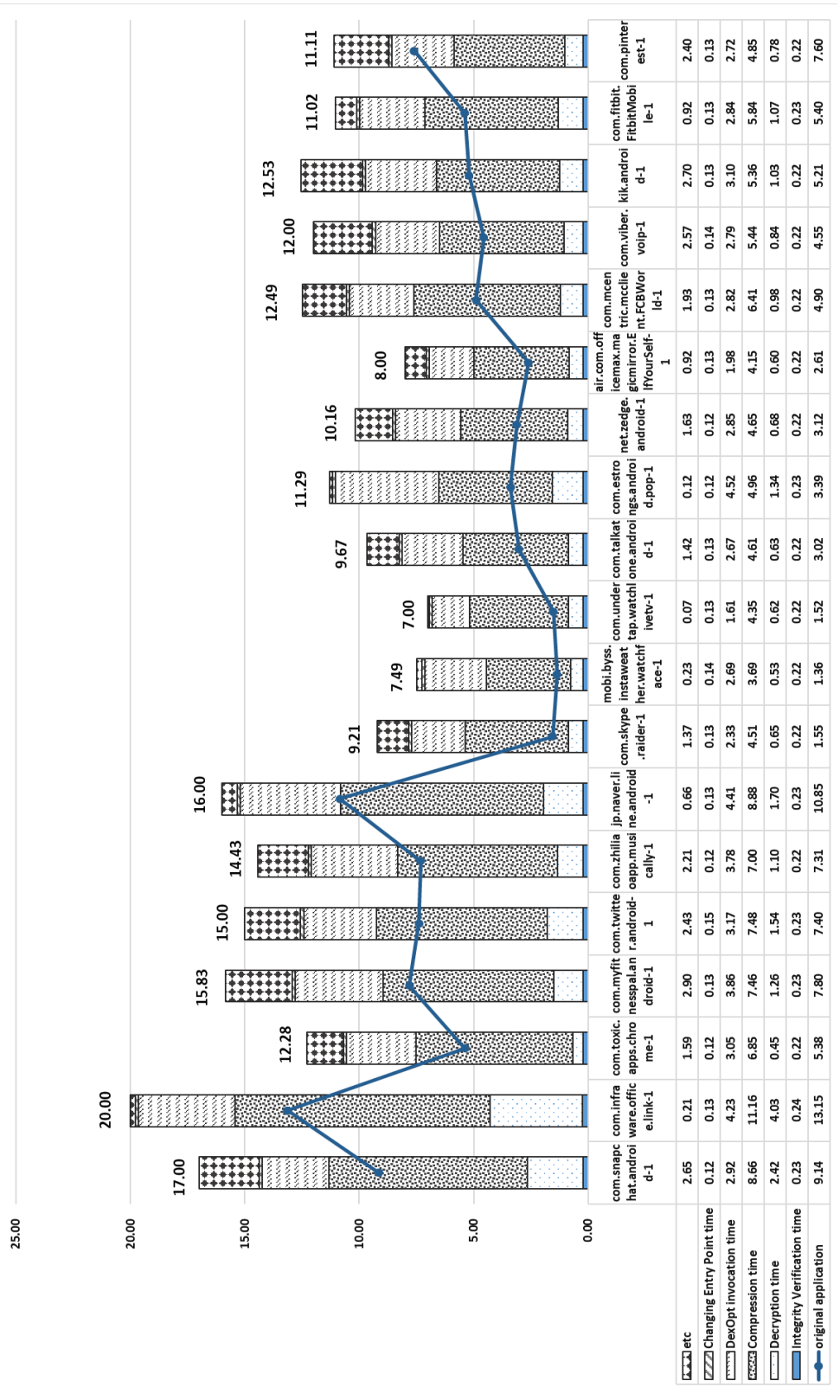
5 CONCLUSION AND FUTURE WORK

WE propose a method to protect multidex Android applications against static reverse engineering. The proposed method encrypts the DEX files of APK using the AES encryption and adds a stub DEX instead of the original classes*.dex file. When the application starts, Verify the integrity of the Stub Dex and stub DEX decrypts and dynamically loads the encrypted DEX file. We modified Google's multidex library to implement stub DEX. Experiments also show that applications implemented using the proposed technique are difficult to reverse-engineer using well-known tools such as dex2jar and baksmali. Compared to other existing packers, the proposed method is more effective for multi DEX files.

A disadvantage of the proposed method is time overhead. According to the experimental result, when the applications packed by the proposed method are executed, it takes more time from minimally 1.46 times to maximally 5.9 times than the execution time of their original Android applications. The additional overhead is mainly caused by the slow execution of the DexOpt program during decryption and dynamic loading the codes of the packed applications.

5.1 Acknowledgment

THIS work was supported by the MSIP, Korea, under the ITRC support program (IITP-2016-R0992-16-1012) supervised by the IITP. 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)



	com.snapchat.android-1	com.infraware.office.link-1	com.toxicapps.chrome-1	com.myfitnesspal.android-1	com.twitter.android-1	com.zhilioapp.musically-1	jp.naver.line.android-1	com.skype.raider-1	mobi.byss.instaweather.watchface-1	com.undertap.watchlivetv-1	com.talkatone.android-1	com.estrongs.android.pop-1	net.zedge.android-1	air.com.officemax.magicmirror.EifYourSelf-1	com.mcentric.mccllent.FCBWorld-1	com.viber.voip-1	kik.android-1	com.fitbit.FitbitMobile-1	com.pinterest-1
etc	2.65	0.21	1.59	2.90	2.43	2.21	0.66	1.37	0.23	0.07	1.42	0.12	1.63	0.92	1.93	2.57	2.70	0.92	2.40
Changing Entry Point time	0.12	0.13	0.12	0.13	0.15	0.12	0.13	0.13	0.14	0.13	0.13	0.12	0.12	0.13	0.13	0.14	0.13	0.13	0.13
DexOpt invocation time	2.92	4.23	3.05	3.86	3.17	3.78	4.41	2.33	2.69	1.61	2.67	4.52	2.85	1.98	2.82	2.79	3.10	2.84	2.72
Compression time	8.66	11.16	6.85	7.46	7.48	7.00	8.88	4.51	3.69	4.35	4.61	4.96	4.65	4.15	6.41	5.44	5.36	5.84	4.85
Decryption time	2.42	4.03	0.45	1.26	1.54	1.10	1.70	0.65	0.53	0.62	0.63	1.34	0.68	0.60	0.98	0.84	1.03	1.07	0.78
Integrity Verification time	0.23	0.24	0.22	0.23	0.23	0.22	0.23	0.22	0.22	0.22	0.22	0.23	0.22	0.22	0.22	0.22	0.22	0.23	0.22
original application	9.14	13.15	5.38	7.80	7.40	7.31	10.85	1.55	1.36	1.52	3.02	3.39	3.12	2.61	4.90	4.55	5.21	5.40	7.60

Figure 8. The major components of the processing time for dynamic code loading in case of packed application with the proposed method.

6 REFERENCES

Android. (2016a). Application. *http://developer.android.com/reference/android/app/Application.html.*

Android. (2016b). Configure apps with over 64k methods. *http://developer.android.com/studio/build/multidex.html.*

Android. (2016c). Dexfile. *http://developer.android.com/reference/dalvik/system/ DexFile.html.*

Android. (2016d). Package java.lang.reflect. *http://developer.android.com/reference/java/lang/reflect/package-summary.html.*

Android. (2016e). ProGuard. *http://developer.android.com/tools/help/proguard.html.*

Android. (2016f). SDK build tools release notes. *http://developer.android.com/tools/revisions/build-tools.html.*

Android. (2016g). Support library features. *http://developer.android.com/tools/support-library/features.html.*

AppBrain. (2016). Number of android applications. *http://www.appbrain.com/stats/number-of-android-apps.*

J. Freke, (2009). baksmali. *https://github.com/JesusFreke/smali.*

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

Google Inc. (2016). Abi, *https://developer.android.com/ndk/guides/abis.html?hl=en*

Java decompiler. (2008). *http://jd.benow.ca/.*

N. Y. Kim, J. Shim, S. Cho, M. Park, and S. Han, (2016). Android application protection against static reverse engineering based on multidexing, *The 2016 International Symposium on Mobile Internet Security (MobiSec 2016), July 2016, Taichung, Taiwan, Journal of Internet Services and Information Security (JISIS),* 6(4), 54–64.

K. Lim, Y. Jeong, S. Cho, M. Park, and S. Han, (2016). An android application protec¬tion scheme against dynamic reverse engineering attacks, *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications (JoWUA)*, 7(3), 40–52.

K. Lim, Y. Jeong, J. Lim, S. Cho, H. Shim, and S. Cho, (2016). Encrypting executable codes and changing entry point of them for preventing android apps against reverse engineering, *Proc. of the 26th joint conference on communications and information (jcci'16), sokcho, korea.* 26, 0244–0245.

B. Pan, (2014). dex2jar, *https://sourceforge.net/projects/dex2jar/.*

M. Parparita, (2014). dex-method-counts, https://github.com/mihaip/dex-method-counts.

T. Reznik, (2014). Android and the dex 64k methods limit-contentful, *https://www.contentful.com/blog/2014/10/30/android-and-the-dex-64k-methods-limit/.*

P. Schulz, (2012). Code protection in Android, *Tech. Rep. No. 110. Rheinische Friedrich-Wilhelms-Universitat Bonn.*

S. T. Sun, A. Cuadros and K. Beznosov, (2015). Android rooting: Methods, detection, and evasion, *Proceedings of the 5th annual acm ccs workshop on security and privacy in smartphones and mobile devices,* 3–14.

C. Tumbleson, and R. Wisniewski, (2010). Apktool, *http://ibotpeaches.github.io/Apktool/.*

Wikipedia. (2016). Abi, *https://en.wikipedia.org/wiki/Application binary interface.*

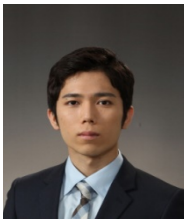
J. Xu, L, Zhang, Y. Sun, D. Lin, and Y. Mao, (2015). Toward a secure android software protection system, *Proc. of 2015 IEEE international conference on Computer and information technology; ubiquitous computing and communications; dependable, autonomic and secure computing; pervasive intelligence and computing*, 2068–2074.

W. Yang, Y. Zhang, J. Li, J. Shu, B. Li, W. Hu, and D. Gu, (2015). Appspear: Bytecode decrypting and dex reassembling for packed android malware, *Proc. of the 18th international symposium on research in attacks, intrusions and defenses (raid'15), kyoto, japan,* 9404, 359–381.

R. Yu, (2014). Android packers: facing the challenges, building solutions, *Proc. of the 24th virus bulletin international conference (vb'14), Seattle, Washington, USA*, 266–275

G. Na, J. Lim, K. Kim, and J. Yi, (2016). Comparative Analysis of Mobile App Reverse Engineering Methods on Dalvik and ART, *2016 Journal of Internet Services and Information Security (JISIS),* 6(3), 54–64.

7 NOTES ON CONTRIBUTORS



K. Lim received the B.E. degree in Dept. of Software Science from Dankook University, Korea, in 2015 and the M.E. degree in computer science and engineering from Dankook University, Korea, in 2016. He is currently a Ph.D. student in Computer Science and Engineering at Dankook University, Korea. His research interests include computer system security, mobile security.

N. Y. Kim received the B.E. degree in Dept. of Software from Dankook University in 2015 and the M.E. degree in computer science and engineering from Dankook University, Korea, in 2016. He is currently a researcher in Inetcop laboratory, Korea. His research interests include computer system security, mobile security, and software protection.

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

S. 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.

S. 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.

M. 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.