DOI: 10.19729/j.cnki.1673-5188.2018.03.008
http://kns.cnki.net/kcms/detail/34.1294.TN.20180825.1050.002.html, published online August 25, 2018

DexDefender: A DEX Protection Scheme to Withstand Memory Dump Attack Based on Android Platform

RONG Yu [1], LIU Yiyi [1], LI Hui [1], and WANG Wei [2]
(1. Beijing University of Posts and Telecommunications, Beijing 100876, China;
2. Government & Enterprise Communications Institute, ZTE Corporation, Nanjing 210012, China)

Abstract

Since Dalvik Executable (DEX) files are prone to be reversed to the Java source code using some decompiling tools, how to protect the DEX files from attackers becomes an important research issue. The traditional way to protect the DEX files from reverse engineering is to encrypt the entire DEX file, but after the complete plain code has been loaded into the memory while the application is running, the attackers can retrieve the code by using memory dump attack. This paper presents a novel DEX protection scheme to withstand memory dump attack on the Android platform with the name of DexDefender, which adopts the dynamic class-restoration method to ensure that the complete plain DEX data not appear in the memory while the application is being loaded into the memory. Experimental results show that the proposed scheme can protect the DEX files from both reverse engineering and memory dump attacks with an acceptable performance.

Keywords

Android; DEX; memory dump; reverse engineering

1 Introduction

Although the Android platform employs multi-level security mechanisms, the adoption of Java language in most of Android applications makes the applications on the platform prone to be decompiled and vulnerable to reverse engineering. An attacker can obtain the Java source code by decompiling an application's Android Package (APK) file, and then repackage them as another APK, which may cause a serious problem with the copyright protection of the software. For example, the game developer of "Dead Trigger", Madfinger, was forced to provide software for free because of software piracy [1], which has brought huge loss. More seriously, attackers can also insert malicious codes into the application that has been cracked [2], and then it will be disguised as a legitimate application to steal user's sensitive information. This not only violates the developer's copyright, but also harms the interests and personal privacy of users.

In order to prevent the applications from being decompiled and reassembled, various methods have been proposed. In 2012, Moon et al. designed a software protection system based on symmetric and asymmetric cryptography [3], in which the users buy applications from a specific application market. The purchased applications use users' public key to encrypt, the users can decrypt applications with the private key, so that only the legitimate users can run the applications. However, attackers can also obtain the applications by copying its codes from the path of mobiles: /data/App.

In the same year, Jeong et al. proposed a mechanism for anti-piracy based on component separation and dynamic loading [4], in which the applications are divided into main programs and plugins. Users install the main program, and then the main program downloads the plugins from the web before the system reminds users to pay. These plugins are protected by encryption, only paid authorized users can decrypt correctly and the decrypted plugins are stored into the phone's security area. However, malicious users can also get root privilege to copy the code of plugins.

All of the above mentioned methods provide ideas for software protection on the Android platform. However, they have their own shortcomings. One possible way to protect the applications is to always keep the key parts of the applications confidential, only decrypt the key parts in memory when it is running, and clear the memory after use, so that the decryption process and calling process will be difficult to track. In this paper, we define the Dalvik Executable (DEX) file as the key part of applications because it contains main information of the applications' source codes. The DEX file is a kind of Dalvik binary byte code file generated by the java source code and can directly run on the Dalvik virtual machine.

The concept of code obfuscation proposed by Collberg et al [5] can be used to protect DEX files by making data promiscuous or obfuscating the control flow so that the code and program become obscure and complex, which can protect the application from being reversed and can prevent the software from direct static analysis. However, the obfuscated executable code can still be deobfuscated by the general approach proposed in [6].

Another way to protect the applications is to encrypt the

This work was supported by ZTE Industry-Academia-Research Cooperation Funds.

DEX files and then hide them in the applications, which has been applied by Bangbang [7], 360 [8], and Dong et al [9]. They encrypt the DEX files of the source APK with encryption algorithms and replace them with a fake DEX file prepared previously. When the program is being executed, the fake DEX file will be run first, and then the fake DEX file can lead the original DEX file to run. Since all these methods protect the DEX files completely, the plaintext of the DEX data will appear in the memory in the run time, which makes it possible for attackers to dump the DEX file from the memory by using Interactive Disassembler (IDA), ZjDroid, Drizzle Dumper or other tools.

To solve this problem, Fan et al. proposed a method to prevent Android App repackaging based on code splitting [10], in which the DEX files are divided into multiple fragments in accordance with the DEX file's format, making the application's executable code be fragmented in its entire life cycle in the memory. Since each DEX file fragment in this approach has a certain feature for attackers to identify, they can get the complete DEX file by dumping and combining from the memory.

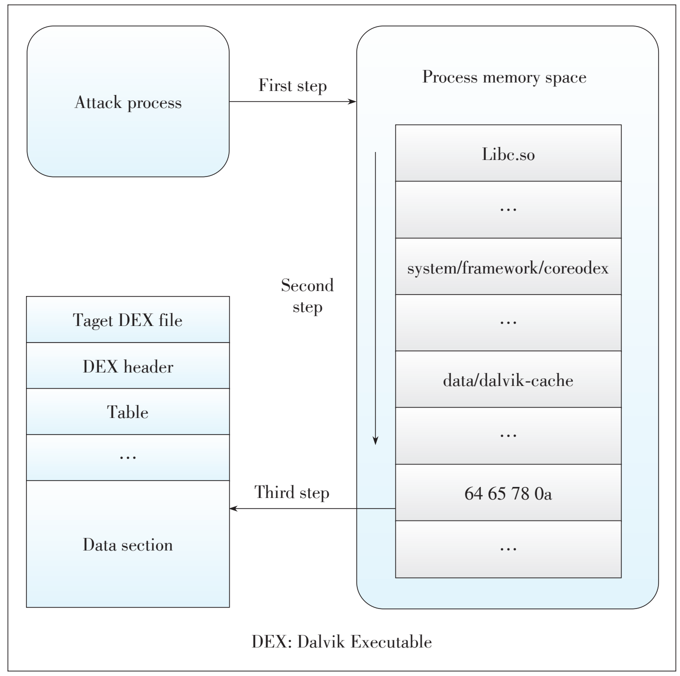
In order to prevent the direct copy of DEX files from memory, this paper presents a scheme named DexDefender to withstand memory dump attacks. It extracts the code fields of classes in the DEX files and then restores each class dynamically into the memory when the program is running. The snippets of the extracted code have no features to be identified by attackers so that it can effectively prevent attackers from cracking the applications by dumping DEX data from the memory.

The rest of this paper is organized as follows. Section 2 presents a memory dump attack approach to obtain DEX data on the Android platform. The third section describes the proposed protection scheme which uses the dynamic class-restoration method to avoid the complete plain DEX data from appearing in the memory. In Section 4 the proposed scheme is analyzed and evaluated. Finally, we conclude our work in Section 5.

2 Memory Dump Attack

The traditional way to enhance the security of DEX files is to protect the files completely. No matter how to hide the DEX file, even if the DEX file is encrypted, the whole plain DEX data must exist in the memory while an application is running. Attackers can dump the DEX data from memory through such tools as IDA, ZjDroid, and Drizzle Dumper. This kind of attack is called DEX memory dump attack.

Such an attack includes three steps as shown in **Fig. 1**. In the first step, when an application reinforced by an existing approach is running, the attacker attaches its process to the application's process. In the second step, the attacker locates the DEX in the memory. The DEX file usually has a consistent and specific format. The DEX file header records some basic information of the DEX file and has a constant length of 0x70 bytes. The first 8 bytes of the file header are named magic field



▲**Figure 1. DEX memory dump attack steps.**

that is used to identify a valid DEX file of a specific value 64 65 78 0a 30 33 35 00. An attacker can search for those 8 bytes in memory, and if the magic field is found in a virtual memory area, the attacker can locate the DEX successfully. Finally the attacker can dump the complete plain DEX data from the memory.

After the attacker obtains the application's DEX file, the application can be cracked so that the attacker can steal the program logic, insert malicious program or repackage the APK.

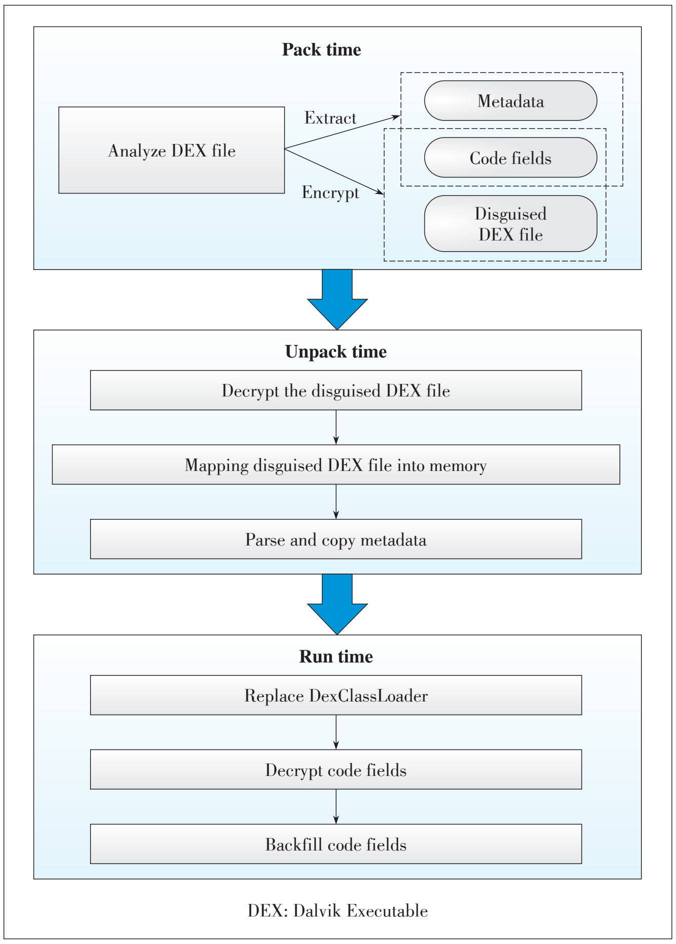
3 The Proposed Solution

3.1 Overview

Because the traditional reinforcement technologies cannot resist the memory dump threat as described in Section 2, this paper presents a DEX protection scheme to withstand memory dump attacks. The purpose of this scheme is to ensure that the complete plain DEX data not appear in the memory when an application is being loaded into the memory. This can better protect the DEX file from being completely dumped from the memory and reduce the possibility of crack applications.

Fig. 2 shows the overall framework of the proposed scheme, which is divided into three phases: pack time, unpack time and run time.

In the pack time, the DEX parser will first analyze the DEX file of APK, extract the code fields of DEX file and encrypt it. Then the code fields' metadata (the offset and length of code fields) is saved, the code fields of the original DEX file (disguised DEX file) is cleared and replaced with the fake DEX



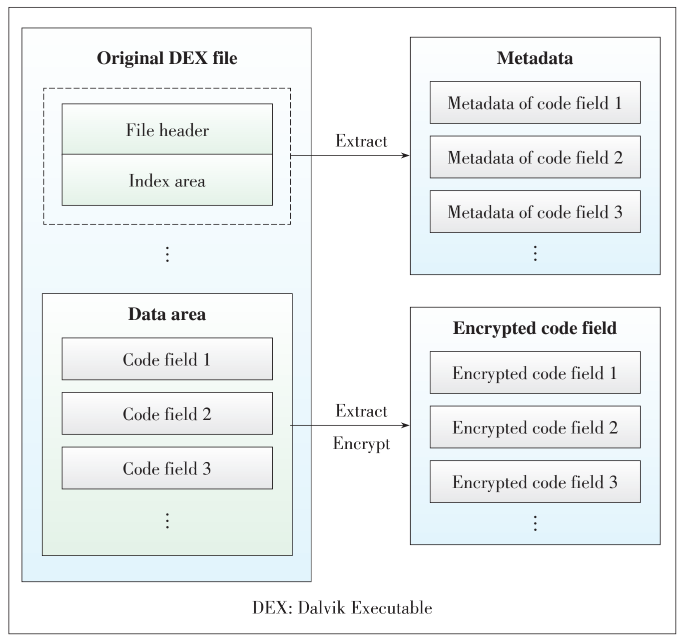
▲**Figure 2. Framework of DexDefender.**

file. Thirdly, the disguised DEX file, metadata, and the encrypted code fields are used as input during the unpack time. The unpack time process is mainly responsible for loading the disguised DEX file from the fake DEX file. Because the disguised DEX file's code fields have been cleared, the complete plain DEX will not appear both in the file system and the memory. In order to keep the original APK running normally, a code field will be decrypted according to the class name that belongs to and backfilled to restore the disguised DEX during the running time. The process of analyzing the original DEX file and restoring the class dynamically in memory is described in Sections 3.2 and 3.3 of this paper.

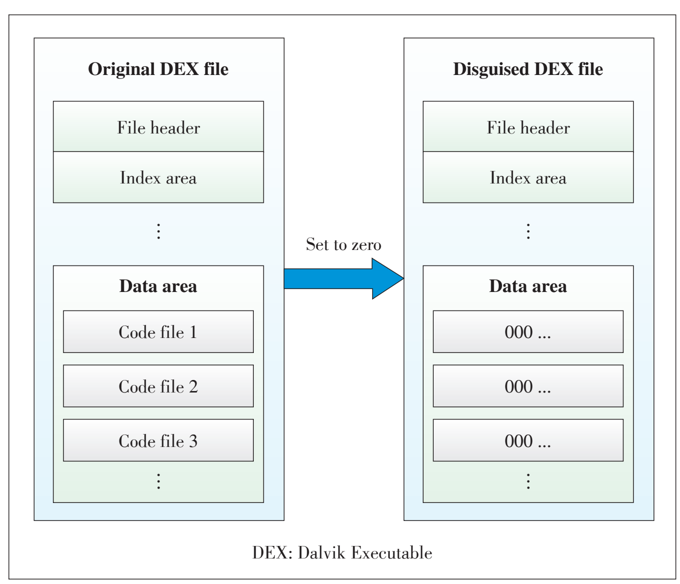
3.2 Analysis of Original DEX File

DEX file structure mainly contains three parts: the DEX file header, index area, and data area. Code fields that contain primary information of the applications are in the data area. The offset and length of the code fields in the DEX file are called code fields' metadata.

The process of extracting code fields and metadata is shown in **Fig. 3**. Because the code fields are not stored in the data area continuously, it is necessary to extract the metadata of each code field, i.e. extract the offset and length of the code field in the original DEX file according to the file header and index area. The metadata will be used to first restore code fields in order to ensure that APK run successfully, and then extract and encrypt each code field. The code fields will be decrypted in the memory to restore the DEX while the application is being loaded into the memory. Finally, the values of code fields of the original DEX need to be changed to zero as shown in **Fig. 4**. With the metadata and encrypted code fields stored separately, it is hard for an attacker to restore the whole DEX file if it only obtained one of them. In addition, since the code fields do not have fixed identifiable features, it is difficult for attack-



▲**Figure 3. Process of extracting code fields and metadata.**



▲**Figure 4. Process of setting code fields as zero.**

ers to locate all real code fields in the memory.

Because the values of code fields of disguised DEX file are zero, even if the attacker can find and dump the disguised plain DEX data from the memory, he cannot get any information about the application.

3.3 Dynamic Class Restoration

When the application is being loaded into the memory, Android system will create a default class DexClassLoader for the application to load class, in which the values of code fields in the disguised DEX are zero so that the DexClassLoader cannot find the real class. Therefore, we need to use our customized DexClassLoader to replace the default DexClassLoader. First, the system's default DexClassLoader is inherited. Then, the findClass method is rewritten. In the findClass method, the dynamic class restoration process is implemented. When the program needs to load a class of the original DEX, the customized DexClassLoader will first index the name of the class, find the corresponding encrypted code fields based on the metadata extracted before, and then decrypt and backfill it to the correct position of the DEX in the memory. The process of class restoration is shown in **Fig. 5**.

By this way, the original APK can run correctly and the memory will be cleared after running the APK.

4 Analysis of DexDefender

DexDefender has been implemented on Android 4.4.4, Android 5.1.1 and Android 6.0. Android 4 uses the Dalvik mode, in which DEX is optimized to Optimized Dalvik Executable (ODEX). Android 5 or Android 6 uses Android Runtime (ART) mode, in which DEX is optimized to Optimized Android Runtime Machine Code (OAT). The specific implementation of DexDefender in Dalvik and ART modes is slightly different, but the structures of DEX are the same in ODEX and OAT. Therefore, the customized DexClassLoader can be used to load the DEX in both Dalvik and ART modes, which makes the number of codes required to be modified in different modes minimal.

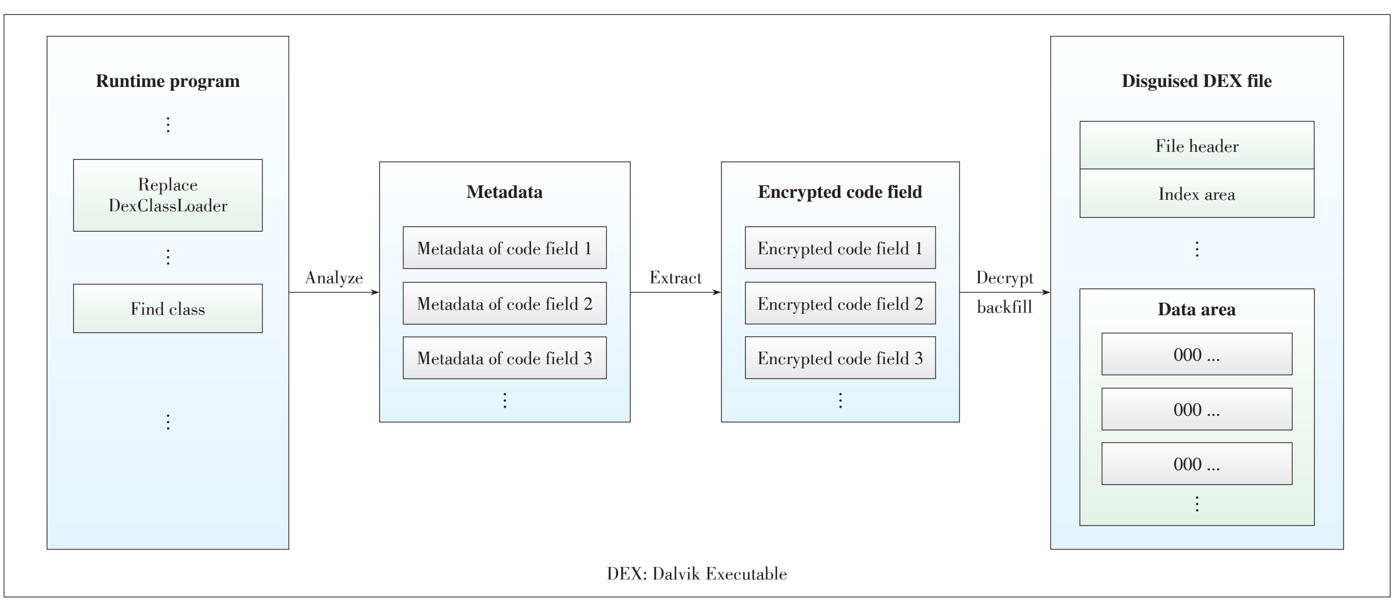
DexDefender adopts the symmetric encryption algorithm of Cipher Block Chaining (CBC) mode. This section will analyze and evaluate the effectiveness and performance of the proposed scheme through the experiment.

4.1 Analysis of Effectiveness

The purpose of designing the approach to withstand the memory dump attack is to avoid loading the whole DEX file into the memory at once. In the proposed scheme, the code fields which contain the most important information are not stored in the DEX file. When the program needs to run and load a class, corresponding code fields will be located through the previously saved metadata and be decrypted to restore the DEX. By this way, only disguised DEX and DEX fragments (code fields) are in the memory and this make it difficult to obtain DEX files at once.

Even if the attacker can locate disguised DEX and dump it from the memory according to the characteristics of the DEX file, the values of DEX file's code fields are zero and attackers cannot get any information about the class of the application. As described in Section 1 of this paper, the attacker could not crack the application even if all the attack steps are completed.

If an attacker wants to retrieve the complete DEX file, it must analyze the characteristics of each field code, and then find and dump all the code fields from the memory. In addition, the attacker would also require a lot of time to restore the DEX file and this process is prone to making mistakes, which



▲Figure 5. Process of dynamic class restoration.

greatly increases the cost of the attack.

To prove that the proposed scheme can prevent the complete plain DEX data from being dumped from the memory, we implemented the attack scenario as described in section 2, using IDA pro for dynamic debug attacks. We installed and ran the reinforced APK, attached to the program's process with IDA pro, found the position of DEX in the memory is 0x74f99028, as shown in **Fig. 6**.

The location of the code field corresponded to the class appstore. Appstore_codec.CharEncoding was 0x7500CD3C. The values of code fields in the corresponding location were changed to zero, as shown in **Fig. 7**. The length of this code fields was 8 bytes.

It can be seen from the corresponding location in the original DEX file as shown in **Fig. 8** that the values of code fields can be successfully changed to zero. The code fields will be restored at the corresponding location in the memory when the program needs to get the class.

In summary, the proposed scheme can ensure that from loading to running time of the application, the complete plain DEX data are not appear in the memory, which makes the cracking more difficult and provides defense against memory dump attacks.

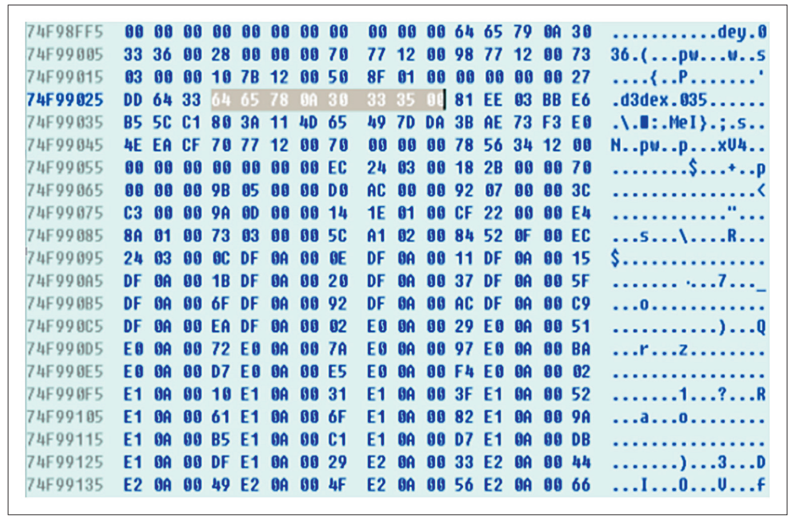
4.2 Analysis of Performance

20 popular applications were selected and tested on an Intel core i5 computer. Both space consumption and time consumption were measured using LG Nexus5. Experimental results show that the increase of the size of applications is less than 1 M. In the Dalvik mode, the increase of the initial startup time of applications is no more than 5 s as shown in **Table 1**. In the ART mode, the increase of the initial startup time of applications does not exceed 5 s, and the restart time does not exceed 2 s, as shown in **Table 2**.

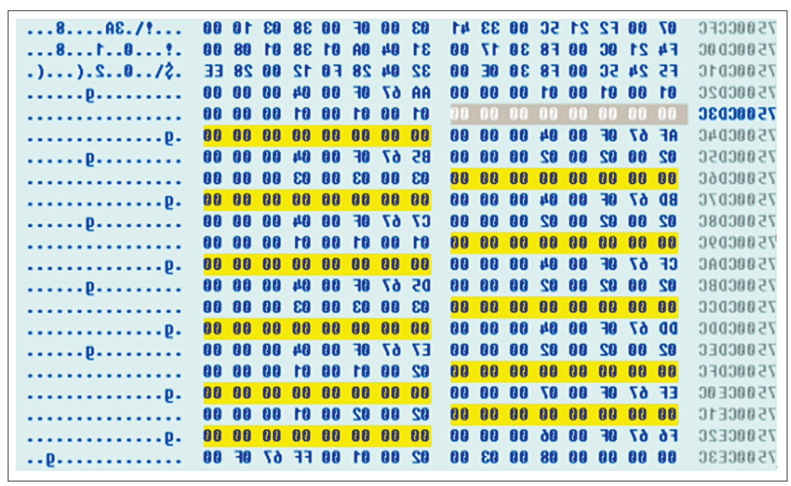
From the experimental results, the space overhead and time overhead of the scheme are within the acceptable range.

5 Conclusions

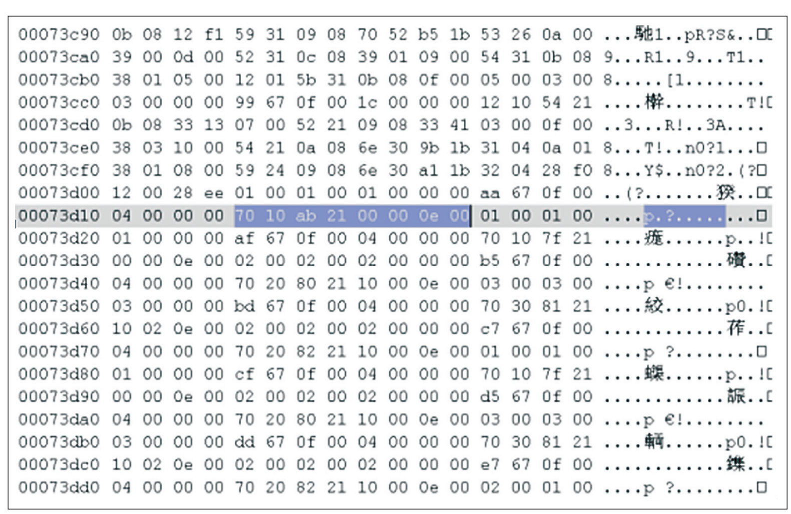
The traditional methods to protect the DEX files cannot withstand the memory dump attack because the whole plain DEX data can be copied after the application is loaded into the memory. In order to protect the DEX files from memory dump attack, DexDefender, a novel DEX protection scheme is proposed. It extracts the code fields in the DEX file and dynamically restores the code fields of each class while the application is loaded. In this way, no complete plaintext of DEX files exist in the memory during the



▲Figure 6. The DEX in memory by using IDA pro.



▲Figure 7. The code fields corresponding to class appstore.appstore_codec.CharEncoding in the memory by using IDA pro.



▲Figure 8. The code fields corresponding to original apk'class appstore.appstore_codec.CharEncoding.

▼Table 1. Time consumption in the Dalvik mode

APK	APK version	Mean of the initial startup time before reinforcement (ms)	Mean of the initial startup time after reinforcement (ms)	Mean of the restart time before reinforcement (ms)	Mean of the restart time after reinforcement (ms)	Initial startup time increment (ms)	Restart time increment (ms)
DicProvider	1	399	717	378	276	318	−102
file_rc4	1	377	815	162	352	438	190
calculator	1	299	680	180	320	381	140
appstore	1	568	942	480	496	374	16
autorun	null	223	1213	219	246	990	27
iietransfer	2.1.0901.2146	785	2037	835	813	1252	−22
baifashop	1.0.0	2920	4184	1827	2875	1264	1048
MicroMessage	1	346	761	340	524	415	184
KuaiGeng	2.1.1	1119	4853	550	2534	3934	1984
Ofo	1.8.9	2358	4784	2524	3170	2426	646
Flipboard	3.5.3.0	2563	5819	2487	4239	3256	1752
Course plaid	9.0.4	1140	4694	1425	2254	3554	829
Gaokao Bang	4.1.1	1198	3969	603	2251	2771	1648
Dubbing hall	1.6.02.01	620	3378	595	1216	2758	621
Translator	5.8.1	2208	6216	2435	3056	4008	621
Tuhua	7.9.A.2.0	948	4113	847	1935	3165	1088
Lily	6.9.0	2368	5990	2385	3899	3622	1514
Yaolan	2.2.2	2580	6172	2297	4290	3592	1993
Xiao D Location	1.0.1	844	3693	645	1610	2849	965
Chuangbie Bookstore	4.1.1	756	3018	1567	2684	2262	1117

APK: Android Package

▼Table 2. Time consumption in the ART mode

APK	APK version	Mean of the initial startup time before reinforcement (ms)	Mean of the initial startup time after reinforcement (ms)	Mean of the restart time before reinforcement (ms)	Mean of the restart time after reinforcement (ms)	Initial startup time increment (ms)	Restart time increment (ms)
DicProvider	1	388	820	398	704	432	306
file_rc4	1	364	740	390	731	376	341
calculator	1	273	701	277	712	428	435
appstore	1	437	862	838	813	425	-25
autorun	null	338	744	218	711	406	493
iietransfer	2.1.0901.2146	972	1533	1055	1491	561	436
baifashop	1.0.0	1647	3756	1679	3356	2109	1677
MicroMessage	1	376	702	343	722	326	379
KuaiGeng	2.1.1	1882	2525	706	1885	643	1179
Ofo	1.8.9	4369	5923	2850	3734	1554	884
Flipboard	3.5.3.0	1926	4604	1595	3586	2678	1991
Course plaid	9.0.4	1925	2733	1099	2184	808	1085
Gaokao Bang	4.1.1	871	2662	368	1349	1791	981
Dubbing hall	1.6.02.01	527	2526	559	2268	1999	1709
Translator	5.8.1	1786	4236	1108	2703	2450	1595
Tuhua	7.9.A.2.0	1051	2728	842	2221	1677	1379
Lily	6.9.0	2340	5909	1306	2529	3569	1223
Yaolan	2.2.2	1429	4073	1284	3169	2644	1885
Xiao D Location	1.0.1	708	2138	663	2590	1430	1927
Chuangbie Bookstore	4.1.1	493	2510	1070	2947	2017	1877

APK: Android Package

whole lifecycle of the application, which increases the difficulty of dumping DEX directly from the memory and cracking the applications. The experimental results show that the proposed scheme can resist memory dump attack with an acceptable performance in both the Dalvik and ART modes on the Android platform.

References

[1] E. Ravenscraft. (2012, Jul. 31). Just how bad is app piracy on android anyway? Hint: we're asking the wrong question. [Online]. Available: http://www.androidpolice.com/2012/07/31/editorial-just-how-bad-is-app-piracy-on-android-anyways-hint-were-asking-the-wrong-question

[2] M. T Yuan, "China Mobile Payment Security Report," *Business Culture*, pp. 54–56, May 2014.

[3] Y. C. Moon, J. H. Noh, A. R. Kim, et al., "Design of copy protection system for android platform," in *International Conference on Information Technology, System and Management*, Chongqing, China, 2012.

[4] Y. S. Jeong, J. C. Moon, D. Kim, et al., "An anti-piracy mechanism based on class separation and dynamic loading for android application," in *ACM Research in Applied Computation Symposium*, San Antonio, USA, 2012, pp. 328–332. doi: 10.1145/2401603.2401674.

[5] C. Collberg, C. Thomborso, and D. Low. (1997). A taxonomy of obfuscating transformations [Online]. Available: http://www.cs.auckland.ac.nz/staff-cgi-bin/mjd/csTRcgi.pl?serial

[6] B. Yadegari, B. Johannesmeyer, B. Whitely, and S. Debray, "A generic approach to automatic deobfuscation of executable code," in *IEEE Symposium on Security and Privacy*, San Jose, USA, pp. 674–691, 2015. doi: 10.1109/SP.2015.47.

[7] W. Zhou, Y. Zhou, M. Grace, X. Jiang, and S. Zou, "Fast, scalable detection of 'piggybacked' mobile application," in *ACM Conference on Data and Application Security and Privacy*, San Antonio, USA, pp. 185–196, 2013. doi: 10.1145/2435349.2435377.

[8] W. Zhou, X. Zhang, and X. Jiang, "AppInk: watermarking android apps for repackaging deterrence," in *Proc. 8th ACM SIGSAC Symposium on Information, Computer and Communications Security (ASLA CCS' 13)*, Hangzhou, China, pp. 1–12, 2013. doi: 10.1145/2484313.2484315.

[9] Z. J. Dong, W. Wang, H. Li, et al., "SeSoa: security enhancement system with online authentication for android APK," *ZTE Communications*, vol. 14, no. S0, pp. 44–50, Jun. 2016. doi: 10.3969/j.issn.1673-5188.2016.S0.005.

[10] R. X. Fan, D. Y. Fang, Z. Y. Tang, et al., "A method of preventing android app repackaging based on code splitting," *Journal of Chinese Mini-Micro Computer Systems*, vol. 37, no. 9, pp. 1969–1974, Sept. 2016.

Manuscript received: 2017-12-14

Biographies

RONG Yu (463397867@qq.com) graduated from Xidian University, China in 2015 and now she is studying for her master's degree at the Beijing University of Posts and Telecommunications (BUPT), China. Her research interests are software security and information security.

LIU Yiyi (793645428@qq.com) graduated from University of Electronic Science and Technology of China (UESTC) in 2016 and now she is studying for her master's degree at the Beijing University of Posts and Telecommunications (BUPT). Her research interests are software security and information security.

LI Hui (lihuill@bupt.edu.cn) got her Ph.D. in cryptography from BUPT, China in 2005. From July 2005, she has been working at BUPT as lecturer and associate professor. Her research interests are cryptography and its applications, information security, and wireless communication security.

WANG Wei (wang.wei8@zte.com.cn) received her B.S. degree from Nanjing University of Aeronautics and Astronautics, China. She is an engineer and project manager in the field of mobile Internet at Government & Enterprise Communications Institute of ZTE Corporation. Her research interests include new mobile Internet services and applications, PaaS, terminal application development, and other technologies. She has authored five academic papers.