DexHunter: Toward Extracting Hidden Code from Packed Android Applications

Yueqian Zhang, Xiapu Luo(✉), and Haoyang Yin

Department of Computing, The Hong Kong Polytechnic University Shenzhen Research Institute, The Hong Kong Polytechnic University, Kowloon, Hong Kong
{csyzhang,csxluo}@comp.polyu.edu.hk, yin.haoyang@connect.polyu.hk

Abstract. The rapid growth of mobile application (or simply app) economy provides lucrative and profitable targets for hackers. Among OWASP's top ten mobile risks for 2014, the lack of binary protections makes it easy to reverse, modify, and repackage Android apps. Recently, a number of packing services have been proposed to protect Android apps by hiding the original executable file (i.e., `dex` file). However, little is known about their effectiveness and efficiency. In this paper, we perform the first systematic investigation on such services by answering two questions: (1) what are the major techniques used by these services and their effects on apps? (2) can the original `dex` file in a packed app be recovered? If yes, how? We not only reveal their techniques and evaluate their effects, but also propose and develop a novel system, named *DexHunter*, to extract `dex` files protected by these services. It is worth noting that *DexHunter* supports both the Dalvik virtual machine (DVM) and the new Android Runtime (ART). The experimental results show that *DexHunter* can extract `dex` files from packed apps effectively and efficiently.

1 Introduction

Being the most popular mobile operating system [29], Android has attracted around 60 % more app downloads than iOS, and made nearly $3 billion in revenue from Google Play last year [18], not to mention many other third-party Android markets. The massive success of Android apps poses lucrative and profitable targets for attackers. For example, it was recently reported that 98 % of mobile malware targeted on Android devices [21]. In particular, attackers usually disassemble popular apps, insert malicious components, and then upload the repackaged apps to various markets for compromising victims' smartphones [13,15,16,45,51,53]. Moreover, attackers can make profits by changing the client IDs of ad components in apps created by others or adding new ad libraries to these apps [23]. These attacks are due to the lack of binary protections, which is among OWASP's top ten mobile risks for 2014 [4].

Recently, a number of packing services (or simply packers) have been proposed to protect Android apps from being reversed, modified, and repackaged [10,22]. The packers usually adopt various approaches to hide the original executable file (i.e., `dex` file) and impede the attempt of dumping the `dex` file. They

G. Pernul et al. (Eds.): ESORICS 2015, Part II, LNCS 9327, pp. 293–311, 2015.
DOI: 10.1007/978-3-319-24177-7_15

also employ code obfuscation techniques to raise the bar of understanding the internal logics. Note that attackers also use packers to harden malware so that they could evade signature-based detection and make it very difficult for security analysts to understand malware [9].

However, little is known about these packers, such as their effectiveness and efficiency. In this paper, we conduct the *first* systematic investigation on Android packers by answering two questions:

- What are the major techniques used by these packers and their effects on apps?
- Can the original `dex` file in a packed app be extracted? If yes, how?

We inspect six packing services that provide web portals to allow users to upload apps for hardening [8,11,12,30,39,50]. Our analysis in Sect. 2 reveals that these packing services usually employ one or more techniques to protect apps, including code obfuscation, dynamic code modification, dynamic loading, and anti-debugging. Moreover, we quantify their overhead, in terms of app's size and launch time, in Sect. 5.1.

Then, we examine whether the original `dex` file in a packed app can be extracted. We propose and develop a novel system, named *DexHunter*, which provides a general approach to recover the `dex` files from packed apps. *DexHunter* exploits the class loading process of Android's virtual machine, including both the Dalvik virtual machine (DVM) and the new Android Runtime (ART) [25]. It is non-trivial to design and develop *DexHunter* because of many challenging issues, such as handling dynamic code modification through a general approach, avoiding anti-debugging techniques, etc. By applying *DexHunter* to packed apps, we found that the packers under examination cannot effectively protect apps and the original `dex` files can be recovered. Note that in this paper we focus on how to extract hidden `dex` files from packed apps *without* touching on how the packers obfuscate the code [14], because obtaining the `dex` files is the prerequisite of deobfuscating the code, and we will investigate the latter in future work.

In summary, our major contributions include:

- We perform the *first* systematic examination on Android packers. We examine their techniques, assess their effectiveness in protecting apps, and evaluate their overhead introduced to apps. Our findings shed light on the research of Android apps protection.
- We propose *DexHunter*, a novel system for recovering the `dex` files from packed apps in both ART and DVM. To our best knowledge, *DexHunter* is the first system that can handle packed apps running on both Android runtimes. We implement *DexHunter* by modifying ART and DVM, and conduct careful evaluation on its effectiveness and efficiency.
- By applying *DexHunter* to real apps packed by six packers, we observe that it can automatically recover most `dex` files. The results indicate that existing packing services are *not* as secure as expected. We also share lessons learnt when dealing with these packers.

The rest of this paper is organized as follows. We examine the techniques used by existing packers in Sect. 2. Section 3 describes the goal and the basic idea of *DexHunter* and Sect. 4 details the design and implementation of *DexHunter*. Section 5 reports the evaluation result. Section 6 discusses the limitations of *DexHunter* and our future work. After introducing related work in Sect. 7, we conclude the paper in Sect. 8.

2 Analysis of Packing Services

In this section, we analyze six app packers, including, Ali [8], Baidu [11], Bangcle [12], Tencent [50], Qihoo 360 Mobile [39], and ijiami [30]. The reasons of selecting them are twofold. First, these packers allow users to upload apps through web portals and then return packed apps. Hence, attackers can easily use such services to pack malware. In contrast, other packers, such as Arxan[1] and Apperian[2], do not provide such services, thus having few samples for analysis. Although it was reported that malware used ApkProtect to evade the detection [9], we cannot access the web page of ApkProtect. Second, China is one of a few countries that have very high Android malware encounter rates [32] and these packers are the major packing services in China, which are developed by professional security companies or big IT companies. We introduce the major techniques used by these packers in Sect. 2.1 and report the evaluation result of the overhead introduced by packers on apps in Sect. 5.1.

2.1 Common Techniques Used by Packing Services

Obfuscation. Obfuscation aims at preventing analysts from understanding the code [14]. Android provides ProGuard to obfuscate apps through modifying the names of classes, fields, and methods [24]. Advanced techniques to obfuscate Android apps, such as reordering control flow graphs, encrypting constant strings, etc., have been recently proposed [40,52]. Developers can also manually conduct obfuscation, such as, using Java reflection to call methods and access fields, implementing major functions in native code and then invoking them through Java native interface(JNI), etc. They can further obfuscate the correlation between Java code and native code by registering JNI methods with semantically meaningless names in the `JNI_OnLoad` function.

Dynamic Code Modification. Android apps are mostly written in Java and then turned into Dalvik bytecode. Note that it is not easy for apps in Dalvik bytecode to arbitrarily modify itself in DVM in a dynamic manner. Instead, they can invoke native code through JNI to modify bytecode in DVM [37], because the native code is running in the same context as the app's DVM so that the native code can access and manipulate the memory storing the bytecodes. As

[1] https://www.arxan.com/.
[2] http://www.apperian.com/.

an example, malware can employ native code to generate malicious bytecodes dynamically and then execute them in DVM [44].

Before executing the `dex` file in the new Android runtime (i.e., ART), ART will compile the `dex` file into `oat` file in the ELF format. The native codes in so files can not only change instructions in `dex` and `oat` files, but also modify key data structures in the memory, such as `DexHeader`, `ClassDef`, `ArtMethod`, etc., in order to assure that the contents are correct only when they are used, and the contents will be wiped out after they have been used.

Dynamic Loading. Android allows apps to load codes from external sources (in `dex` or `jar` format) at runtime. Leveraging this feature, packers usually encrypt the original `dex` file, decrypt and load it before running the app.

Anti-debugging. Since Linux allows a process to attach to another process for debugging, to thwart the debugging through *gdb*, packed apps usually attach to themselves using *ptrace*[1]. The rationale is that only one process can attach to a target process at the same time. In other words, if an app (target process) attaches to itself at runtime, gdb cannot attach to it, thus further debugging operations are prohibited. Some packers will also check whether special threads, such as the JDWP (Java Debug Wire Protocol) thread, have been attached. Moreover, advanced packers can check whether the apps are running in an emulator or the underlying system has been rooted.

Table 1. A summary of the six packers' features.

Packing service	Obfuscation	Dynamic code modification	Dynamic loading	Anti-debugging	Add shared libraries	Insert classes	Support ART
Bangcle	YES	NO	YES	YES	YES	YES	YES
Tencent	YES	YES	NO	YES	YES	YES	YES
360 Mobile	YES	NO	YES	YES	YES	YES	YES
ijiami	YES	NO	YES	YES	YES	YES	YES
Ali	YES	YES	YES	YES	YES	YES	NO
Baidu	YES	YES	YES	YES	YES	YES	YES

2.2 Packers Under Investigation

We identify the major techniques used in the six packers through manual analysis. Since these packers are evolving and do not provide version number, our examination is based on the packed apps whose original versions were uploaded those packers' web portals on March-15-2015. As shown in Table 1, all of them

add extra shared libraries (i.e., 6th column) and new instructions to the original app (i.e., 7th column). Moreover, they adopt obfuscation (i.e., 2nd column) and anti-debugging techniques (i.e., 5th column). While only half of them use dynamic code modification (i.e., 3rd column), all except Tecent packer employ dynamic loading approach. As Google introduced the new runtime (i.e., ART) to replace DVM, all except Ali packer support ART.

3 DexHunter: Goal and Basic Idea

DexHunter aims at extracting `dex` files from packed apps through a unified approach. It first launches the packed app in a real smartphone, and then locates and dumps the unpacked content when the app is running. We will also correct some fields corrupted by packers if necessary. Note that *DexHunter* does not handle code obfuscation and junk instructions. Moreover, it only considers the dynamic loading conducted when an app is executed, because most packers do so to shorten launch time. We discuss how to extend *DexHunter* to deal with arbitrary dynamic loading in Sect. 6.

3.1 Basic Idea

Android apps are compiled to `dex` files, which are in turn zipped into a single `apk` file together with other resources. If DVM is utilized, when a newly installed app is started for the first time, DVM converts the `dex` file extracted from the `apk` file to the `odex` format. If ART is used, it will turn the `dex` file into the `oat` file upon the installation [20].

An intuitive approach to realize *DexHunter* is to first locate the `odex` header or the `oat` header in the memory by searching for their magic numbers, and then dump the corresponding memory by parsing the headers. However, this approach has several limitations. First, accessing the packed app's memory requires such approach to attach to the app's process, such as using `ptrace`[1]. Unfortunately, packed apps usually employ anti-debugging techniques as described in Sect. 2 to prevent itself from being attached. Second, this approach will miss the real content resulted from dynamic code modification that happens when a class is being initialized. Note that a class may be loaded without being initialized. Third, this approach may miss `dex` files due to corrupted `dex` headers caused by packed apps. Fourth, this approach may dump fake `odex` or `oat` files because packed apps can create fake headers.

To tackle these issues, we propose a novel and unified approach that exploits the class loading process of Android runtime, including both DVM and ART, to locate and dump the desired files. The rationale behind the basic idea is that Android runtime can locate and parse the `dex` file in order to execute it. While the following analysis is based on the source code of Android 4.4.3, we believe the basic idea can be applied to future versions.

Since each class should be loaded before it can be used, Android provides three approaches [28] to load classes: (1) the implicit procedure of loading classes,

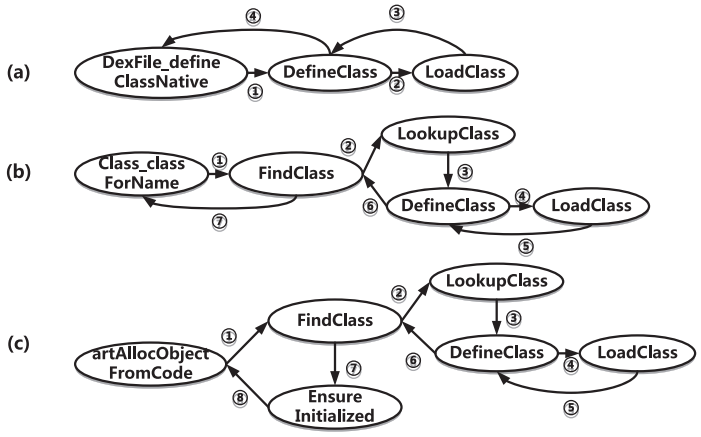


Fig. 1. The three approaches of loading classes and their invocation graphs in ART. The numbers indicate the invocation order.

such as the `new` operation, which happens if the corresponding class has never been used before; (2) the explicit invocation of `Class.forName`; (3) the explicit invocation of `ClassLoader.loadClass`. Although DVM and ART have different implementations for these class loading approaches, we observe that for a given virtual machine these three approaches share a few key common functions, which will be elaborated in Sects. 3.2 and 3.3 for ART and DVM, respectively.

Leveraging this observation, *DexHunter* inserts codes into a selected key function to locate the required files and trigger the invocation of <clinit>. Moreover, we propose novel approaches (Sect. 4.3) to pro-actively load and initialize classes. To overcome anti-debugging and anti-emulating techniques, we integrate *DexHunter* with DVM and ART, and execute packed apps in a real smartphone running modified DVM and ART as described in Sect. 4.2.

3.2 ART

In KitKat (Android 4.4), the new Android runtime, ART, was introduced to replace DVM for better performance by compiling an app's bytecode into native instructions. Adopting the ahead-of-time compilation *(AOT)* technology, ART performs the compilation when an app is being installed. More precisely, the `dex` file will be compiled into `oat` file that adopts the ELF format.

To load a class, ART reads the `dex` or `jar` file using a native method called `DexFile_openDexFileNative` in *libart.so*. If the corresponding `oat` file does not exist, ART invokes a tool named `dex2oat` to compile the `dex` or `jar` file into an `oat` file. If the `oat` file exists but has not been loaded, ART reads it and puts it into a memory cache map to avoid opening the file repeatedly. After successfully accessing the `oat` file, ART creates a structure named `OatFile` to record important information of this file. We will detail it when describing how to dump the `dex` file in Sect. 4.2.

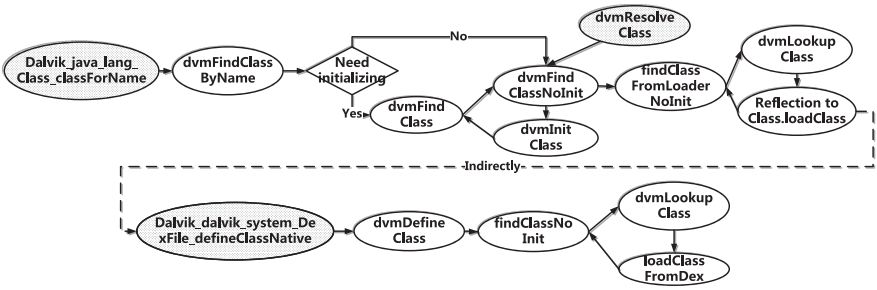


Fig. 2. The three approaches of loading classes and their invocation graphs in DVM.

Then, ART can use different methods to load the class, whose invocation graphs are shown in Fig. 1. More precisely, the explicit invocation of `ClassLoader.loadClass` will call the native method `DexFile_define ClassNative` (i.e., Fig. 1(a)). The invocation of `Class.forName` will call the native method `Class_classForName` (i.e., Fig. 1(a)). The `new` operation will eventually call the native method `artAllocObjectFromCode` (i.e., Fig. 1(c)). By comparing the two sub-figures in Fig. 1, we can locate the common functions called by these three approaches. More precisely, we select `DefineClass` as the key function for inserting *DexHunter*'s code, because it creates the `Class` object and is responsible for loading and linking classes.

3.3 DVM

Figure 2 illustrates the three approaches of loading classes and their invocation graphs in DVM. The invocation of `Class.forName` will call `Dalvik_java_lang_ Class_classForName`. Calling `ClassLoader.loadClass` will eventually invoke `Dalvik_dalvik_system_DexFile_defineClassNative`. The implicit class loading will result in the invocation of `dvmResolveClass`. Moreover, `dvmInitClass` is responsible for initializing a class. Before invoking it, the initialization status is checked through `dvmIsClassInitialized`. The *Reflection to Class.loadClass* in Fig. 2 means that there is a reflection invoking procedure that invokes the related class loader's `loadClass` method at Java level. By analyzing Fig. 2, we select `Dalvik_dalvik_system_DexFile _defineClassNative` as the key function for injecting *DexHunter*'s code, because it creates the `Class` object and loads the class from the dex file directly.

4 DexHunter: Design and Implementation

4.1 Architecture

Figure 3 depicts the major procedure of *DexHunter*. Given a packed app, *DexHunter* first determines whether it is packed by known packing services (i.e., those in Table 1) through the signatures to be described in Sect. 4.4. Moreover,

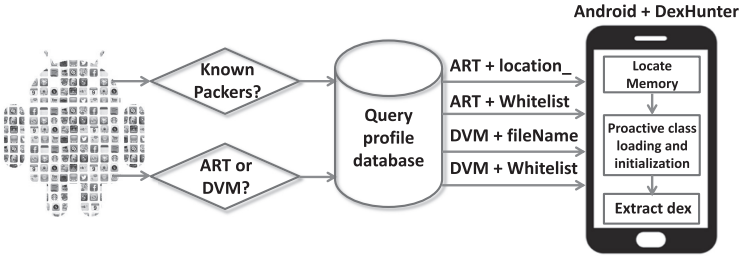


Fig. 3. Using DexHunter in smartphone to recover `dex` files from packed apps.

we will check which runtime can run this app. If the app supports both DVM and ART, we will use the ART version *DexHunter* to recover the `dex` file. If the app is packed by known packers, we will obtain the corresponding parameters from the profile database, including `location_` for ART and `fileName` for DVM, which will be detailed in Sect. 4.2. Otherwise, *DexHunter* will dump the target memory but exclude system libraries listed in a while list.

Depending on the selected runtime, the packed app will be installed and executed in a smartphone with modified libart.so or libdvm.so for ART or DVM, respectively. If DVM is used, *DexHunter* will first dump the optimized `dex` file from the smartphone and then combine it and its dependent files to reconstruct the `dex` file. If ART is adopted, *DexHunter* will generate the `dex` file directly.

4.2 Locating and Dumping Dex Files

ART. Note that each `oat` file contains the information of the original `dex` file in its `oatdata` section [43]. Therefore, after ART opens and reads an `oat` file, it will create an `OatFile` structure to record important information of the file and a `DexFile` object containing information related to the original `dex` file. In particular, there are three important values in the `DexFile` object, through which we can locate the `dex` file, including:

- `begin_`, which depicts the start address of the memory region containing the original dex file;
- `size_`, which represents the length;
- `location_`, which indicates the `oat` file's location.

We add codes in the `DefineClass` function to check the value of `location_` when a class is being loaded. Section 4.5 describes how to decide the packed app's `location_` and the system libraries' `location_`. Therefore, by specifying the value of `location_`, we can recognize all classes in the original `dex` file and then create a thread to accomplish the dumping operation. In this thread, the `DexFile` object, which is also a parameter of the `DefineClass` function, is passed in and then the thread can get the memory region to which the `DexFile` object refers. By invoking the methods `DexFile::Begin()` and `DexFile::Size()`, we can obtain the start address and the length of the memory region containing the original dex file. As a result, we can recover the original `dex` file.

DVM. After loading a `dex` or `jar` file, DVM will create a structure named `DexOrJar`, which records the information of the file. One member named `fileName` refers to the location of the file. Moreover, a `DvmDex` object, which represents an open `odex` file, is associated with the corresponding `DexOrJar` object. The `DvmDex` object has a member named `memMap` that maintains the corresponding memory region of the opened dex file. Its `addr` member stores the start address while the `length` member denotes the length of the memory region.

To dump the desired `dex` file, we add codes to the selected function `Dalvik_dalvik _system_DexFile_defineClassNative` and specify the value of `fileName`. Once the `dex` file we expect is located through `fileName`, the memory region of the targeted `odex` file can also be figured out through the related `DvmDex` object. More precisely, the member `memMap` in the `DvmDex` object records the specified memory region. The member `addr` of `memMap` indicates the start address while the member `length` stores the length. As a result, we can obtain the `odex` file.

The `odex` file format was designed to let DVM work more efficiently and it is usually much smaller than the original `dex` file, because it only includes critical information. For instance, in an `odex` file, references to framework APIs are replaced by indexes of a pre-loaded *vtable* and therefore methods can be quickly invoked. Therefore, `odex` files rely on dependence files, which are device-specific and can be found in the directory `/system/framework`.

`Odex` files cannot be converted into `dex` format directly because they rely on dependencies. Since dependencies are device-specific, they must be copied from the same device that runs the packed app. Finally, *DexHunter* uses smali/backsmali to recover the `dex` file from the `odex` file and its dependencies [2].

4.3 Proactive Class Loading and Initialization

For each newly loaded class, its class initializer (i.e. <clinit>) may not be invoked yet. Since this method is invoked before any other method in the same class, packers can add codes in <clinit> to perform dynamic code modification.

To deal with this potential issue, we propose a novel approach that turns ART's lazy initialization into proactive class loading and initialization. Note that ART calls <clinit> only after the `Class` object is used for the first time, such as invoking static method member, etc. Our approach loads all classes in the same `dex` file and initializes them as shown in Algorithm 1. More precisely, in ART, before the dumper thread is created, *DexHunter* traverses all classes in the same `dex` file in `DefineClass` function, and then invokes the `FindClass` function along with every class's descriptor for loading them. Note that invoking `FindClass` can avoid loading the same class repeatedly in the same class loader. After that, each class is initialized by invoking `EnsureInitialized`. All these operations are done in the same loop.

The algorithm for DVM is similar except that `FindClass` is changed to `dvmDefineClass` and `EnsureInitialized` is replaced with `dvmIsClass Initialized` and `dvmInitClass`.

Algorithm 1. Traversing and Initializing Classes

input : A "DexFile" pointer *dex_file* and the number of classes in this dex file n
output: All initialized "Class" objects belonging to the dex file

```
for i ← 0 to n − 1 do
    ClassDef ← GetClassDef(dex_file,i);
    Descriptor ← GetClassDescriptor(ClassDef);
    ClassObject ← FindClass(Descriptor);
    ClassObject ← EnsureInitialized(ClassObject);
end
```

4.4 Identifying Packers

Known Packers. *DexHunter* identifies known packers using (1) changes in files, (2) inserted classes, and (3) `location_` for ART and `fileName` for DVM. We observe that all packers add new files, especially native codes (i.e., so files), as shown in Table 2. Moreover, they modify the original `AndroidManifest.xml` and `classes.dex`. After inspecting packed apps, we find that all packers insert their own classes into the app, as shown in Table 3. We will describe how to extract `location_` or `fileName` in Sect. 4.5. Since it is easy to recognize and differentiate these inserted files and classes, we use them as features to recognize known packers. In future work, we will investigate advanced features, such as Software bertillonage [17], if packers try to hide current features.

Unknown Packers. For unknown packers, we observe that they usually adopt dynamic code modification with the following common steps. First, they load packed `dex` files dynamically into memory, which will be converted to `oat` files by ART. Then, they employ memory manipulation functions (e.g., *"memcpy"*) to modify the code. Before that, they may call *"mprotect"* to alter the accessing attributes of corresponding memory regions, for example, changing a memory fragment from read-only (`r--`) to readable and writable (`rw-`). We can hook aforementioned functions to capture this behavior patten. If such behavior pattern is observed, *DexHunter* regards the app as a packed app.

4.5 Extracting the Values of `location_` and `fileName`

`location_` and `fileName` provide hints to dump the desired `dex` files in ART and DVM, respectively. To examine their values set by different packers and those used by system libraries, we modify ART and DVM to collect these values.

In ART, we add a function named `GetUid` to obtain the current process's user id by invoking system calls directly instead of using `getuid` in bionic library due to the limit of the configuration for compiling Android. Moreover, we modify `DefineClass` function to record all `location_` values if the current process's user id is equal to that of the target app. Therefore, when `DefineClass` is used to generate the `Class` object for the opened `oat` file, we can obtain the names

Table 2. New files introduced by the packers. "xxx" denotes the app's original package name.

Packers	New files
360	assets/libprotectClass.so, assets/libprotectClass_x86.so, assets/libqupc.so
ALi	lib/armeabi/libmobisec.so, lib/armeabi/libmobisecx.so,lib/armeabi/libmobisecy.so,
	lib/armeabi/libmobisecz.so
Baidu	assets/baiduprotect.jar,assets/libbaiduprotect_x86.so,lib/armeabi/ libbaiduprotect.so,
	lib/x86/libbaiduprotect.so
Bangcle	assets/bangcleplugin/container.apk,assets/bangcleplugin/dgc,assets/meta-data/manifest.mf
	assets/meta-data/rsa.pub,assets/meta-data/rsa.sig,assets/bangcle_classes.jar
	assets/libsecexe.so,assets/libsecexe.x86.so,assets/libsecmain.so
	assets/libsecmain.x86.so,assets/libsecpreload.so,assets/libsecpreload.x86.so
	assets/xxx,assets/xxx.art,assets/xxx.L
	assets/xxx.x86,assets/xxx.x86.L
ijiami	assets/ijm_lib/armeabi/libexec.so,assets/ijm_lib/armeabi/libexecmain.so,
	assets/ijm_lib/x86/libexec.so
	assets/ijm_lib/x86/libexecmain.so,assets/ijiami.dat
	META_INF/af.bin, META_INF/sdata.bin,META_INF/signed.bin
Tencent	assets/lib/armeabi/libmain.so,assets/lib/armeabi/libshell.so

Table 3. Inserted classes. The classes in parentheses will only appear if the original `dex` file has an `Application` class. Otherwise, they will not be inserted.

Packers	Inserted classes
360	com.qihoo.util.StubApplication, com.qihoo.util.DefenceReport
ALi	com.ali.mobisecenhance.StubApplication
Baidu	com.baidu.protect.A, com.baidu.protect.StubApplication, com.baidu.protect.StubProvider
Bangcle	com.bangcle.protect.Acall,com.bangcle.protect.MyClassLoader, com.bangcle.protect.Util
	neo.proxy.DistributeReceiver
	(com.bangcle.protect.FirstApplication), (com.bangcle.protect.ApplicationWrapper)
ijiami	com.shell.NativeApplication
	(com.shell.SuperApplication)
Tencent	com.tencent.StubShell.ProxyShell, com.tencent.StubShell.ShellHelper

Table 4. The values of `location_` or `fileName` in apps packed by six packers.

Packers	String
Bangcle	/data/data/package_name/.cache/classes.jar
Baidu	/data/data/package_name/.1/classes.jar
Tencent	/data/app/installed_apk_name
360	internal.dex (/data/local/tmp/fake@apk.dex)
ijiami	/data/data/package_name/cache/.0000
ALi	/data/app-lib/installed_apk_name/libmobisecy.so (i.e., the path of libmobisecy.so, which is located in the app's native library directory)

of all `dex` files related to the classes being loaded. We first filter out all known system libraries and then decide which names should be kept according to the features of different packers. For instance, some packers load the original `dex` file dynamically and the `oat` file bound to the name of installed apk is only a stub. Hence, such names should be removed.

In DVM, we follow the similar steps to collect the values of `file Name`. More precisely, we modify the function `Dalvik_dalvik_system_ DexFile_defineClassNative` to locate the `DexOrJar` object and get the value of `fileName` in this object.

Table 4 lists the `location_` or `fileName` from six packers we examine. For apps packed by 360 packer, the value is "/data/local/tmp/fake@apk.dex" when the apps are executed for the first time. Then, the value is changed to "internal.dex".

5 Evaluation

We downloaded 40 open source apps from F-Droid [6] and uploaded them to the web portals of the six packers. Then, we execute the packed apps and *DexHunter* on a Nexus 4 smartphone running Android 4.4.3 with Qualcomm Snapdragon S4 Pro 1.5 GHz CPU and 2G RAM. Table 5 shows that not all apps can be successfully packed by those packers and some packed apps cannot be run.

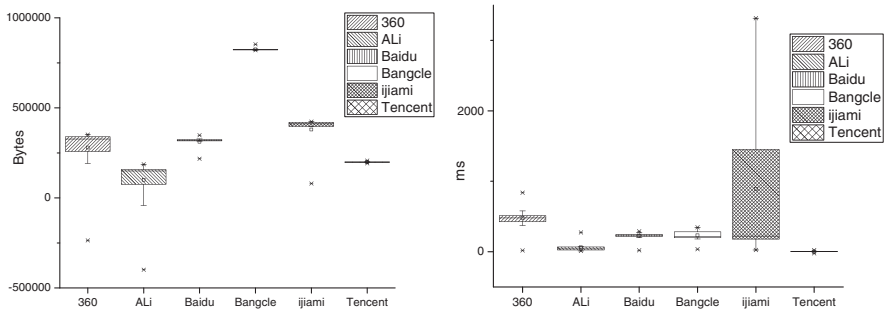
5.1 Overhead Introduced by Packers

We evaluate the overhead introduced by different packers in terms of increased file size and prolonged launch time. By subtracting the original file size from the size of packed app, we obtain the increased file size. Figure 4 illustrates that most packed apps are larger than the original apps and Bangcle introduces more than 600 KB data. A few packed apps are smaller than the original ones. The reason is some packers will compress the original dex file.

To measure the prolonged launch time, we randomly select 17 apps and run each original app and the packed one 30 times in the smartphone. We collect

Table 5. Creating packed apps.

Packers	Number of apps	Number of packed apps	Numbers of packed apps that can run
360	40	39	37
ALi	40	39	37
Baidu	40	37	36
Bangcle	40	40	40
ijiami	40	40	40
Tencent	40	40	38



(a) Increased file size. (b) Prolonged launch time.

Fig. 4. Overhead introduced by packers in terms of increased file size and launch time.

the samples of launch time (i.e. from its start to the end of its main activity's initialization) measured by executing "am start -n -W MainActivity", and then compute the inflated launch time. Figure 4(b) demonstrates that all packers introduce obvious additional delays. The minimal delay brought by Tencent packer may be due to the fact that it does not load external `dex` files.

5.2 DexHunter's Effectiveness

We apply *DexHunter* to all packed apps that can run in the smartphone. In fact, *DexHunter* can bypass all anti-debugging methods used by these packers. Since it becomes part of the process created by Zygote, all anti-debugging methods mentioned in Sect. 2.1 will not stop *DexHunter*.

For apps packed by 360 packer and ijiami packer, *DexHunter* can recover the `dex` files in both ART and DVM. Moreover, the extracted `dex` files can be parsed by de-compilers (e.g., smali/baksmali, IDA, etc.).

For apps packed by Bangle, *DexHunter* can successfully extract the `dex` files in both ART and DVM. The `dex` files dumped from DVM can be parsed by de-compilers. However, the `dex` files recovered from ART have some instructions that cannot be parsed by baksmali. The reason is that the `dex` files are extracted

from the `oat` files prepared by Bangcle packer that has used some new Dalvik opcodes [5]. The developer of baksmali said that this issue will be fixed soon.

For apps packed by Tencent packer, we found that the dex files dumped by *DexHunter* are incomplete in both ART and DVM, because the method objects in the heap, which represent hidden methods, are modified dynamically but the `dex` file in memory is not changed. However, since the valid data is still in the `dex` file's data section, we can manually correct the attributes and the related pointers of the hidden methods in the `dex` file.

For apps packed by Baidu packer, we observe that the `dex` file's header will be wiped if any class's initializer is executed. Hence, we perform the dumping operations without pro-actively initializing the classes. Moreover, we found that the dumped `dex` files are incomplete. More precisely, in `dex` files, for each class, there is a `class_data_item` object to describe the members of the class. However, some `class_data_item` objects of the dumped dex file are wiped by Baidu packer. In order to capture the positions of the `class_data_item` objects, we modified the runtime to record the addresses of `class_data_item` objects. When the application runs, the wiped `class_data_item` objects in so files will be released to the heap and the pointers, which are in the `dex` file, to the `class_data_item` objects will also be corrected. After filling in the correct data in the `class_data_item` objects, we can obtain complete dex files.

Since Ali packer only supports DVM, *DexHunter* recovers the dex files in DVM. In a `dex` file, each `code_item` object describes a method and maintains a pointer to it. But some pointers to `code_item` objects are invalid in the dumped dex files. We modified DVM to obtain the addresses of `code_item` objects and the corresponding instructions. Combining the process's memory layout, we found that the lost `code_item` objects and instructions are located in a memory region allocated by the packed app. To repair the `dex` files, we could also dump this memory region and record the addresses of the lost `code_item` objects.

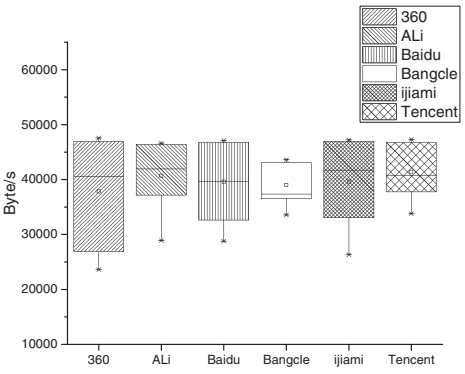


Fig. 5. Dumping speed of *DexHunter*.

5.3 DexHunter's Efficiency

We also evaluate *DexHunter*'s efficiency on the same Nexus 4 device. We randomly select 15 apps that can be packed by all six packers. For each sample, *DexHunter* performs the dumping operation for 30 times. Note that the time complexity of the dumping procedure is $O(n)$(n represents size of the target memory region in bytes). Figure 5 shows *DexHunter*'s dumping speed which is around 40 KB/s and does not change much among different packers.

6 Discussion

Although *DexHunter* can recover the `dex` files from apps packed by existing packers, it has the following limitations and we will tackle them in future work. First, some packers will wreck some fields in the dumped `dex` files as mentioned in Sect. 5.2. Currently, we repair them through semi-automatic or manual approach. In future work, we will enhance *DexHunter* to automate this process.

Second, if an app dynamically loads components from other places after waiting for a long period or certain conditions, *DexHunter* cannot dump this `dex` file, because *DexHunter* does not know when the component will be loaded. We will extend DexHunter to handle it by hooking all methods for dynamic class loading in future work. Alternatively, we can first conduct static analysis [38] to determine how to trigger the app's dynamic class loading and then perform it.

7 Related Work

Hardening Android apps has attracted great attention from the industry [9,26]. Although there are a few simultaneous work from the industry, there lacks of a systematic study on it yet. In a recent article and presentation [9,34], Apvrille and Nigam reported the results of manually unpacking apps packed by a few packers, such as APKProtect and Bangcle. Strazzere and Sawyer reported their tool, named android-unpacker, to defeat four packers including APKProtect, Bangcle, 360 Mobile, and LIAPP [48,49]. Since it will attach to the last thread of an app, we observed that it failed in several scenarios, such as, the thread has already been attached by a ptrace, the thread is killed, etc. Note that *DexHunter* will not be affected by this issue because it is integrated into the runtime. We developed *DexDumper* for extracting the `dex` files of apps running on Android 2.3 or older versions [45]. Note that *DexDumper* lacks of the functions provided by *DexHunter*, including handling apps running on Android with version newer than 2.3, dealing with anti-debugging, processing odex files, etc.

ZjDroid was released by Baidu Inc. [7] for unpacking packed apps. It relies on Xposed [3] and locates the `dex` files by hooking `BaseDexClassLoader` to obtain `DexOrJar`. There are several significant differences between ZjDroid and *DexHunter*. First, *DexHunter* supports both ART and DVM while ZjDroid only works in DVM. Second, ZjDroid cannot pro-actively load and initialize classes

and therefore it may miss the real content resulted from dynamic code modification that happens when a `Class` object is being initialized. *DexHunter* can overcome this issue because it conducts pro-active class loading and initialization. Third, since ZjDroid waits for user commands to dump the `dex` files, it may be evaded by packers that destroy some key data which is used only once. *DexHunter* can handle this issue because it extracts the `dex` files *before* the first class in the `dex` file is used. Fourth, since ZjDroid relies on Xposed and obtains the information at Java level, it can be easily detected and interrupted by advanced packed. In contrast, *DexHunter* will not be affected.

Park described one general unpacking method for packed apps [35]. It is quite different from *DexHunter* because it needs to insert codes to packed apps (i.e., repackage the packed app). This approach can be easily detected by packed apps. Moreover, compared to *DexHunter*, its functionality is quite limited.

Since packing is widely used by malware to evade the signature-based detection, many studies have investigated how to unpack such malware [19]. However, all of them focus on packers for Windows/Linux native codes [41]. It is worth noting that unpacking techniques for x86 binaries cannot be applied to Android because of two reasons. First, Android and x86 have different execution model. Second, techniques for x86 unpacking only need to examine x86 instructions in memory while dumping `odex` files need to investigate both the memory of Android runtime (e.g., DVM) and that of the underlying Linux because packers usually use native codes running on Linux to modify the byte codes in DVM.

We review some representative work of automatically dumping packed native executables because an app's native codes can be packed through traditional approaches. PolyUnpack is the first general approach to automatically identify and dump packed codes [42]. It first statically analyzes an executable and then uses debugging APIs to check each instruction. If an instruction sequence does not exist in the disassembly of the executable, PolyUnpack identifies the packed codes and then extracts them. Renovo runs a packed executable in QEMU and monitors each instruction [31]. If new codes are written to memory and then executed, Renovo regards it as one layer of unpacking conducted by the packed program. Instead of tracking each instruction, OmniUnpack [33] and Eureka [46] adopt coarse-grained execution monitoring to improve the performance. The former uses OllyBone [47] to track executed pages and the latter monitors selected system calls. Justin employs a set of heuristics to improve the detection of the end of unpacking and adopts several countermeasures to defeat some evasion techniques used by malware [27]. Although dynamic approaches could effectively extract packed code, they suffer from some common limitations, such as, higher overhead compared to static analysis, limited time of executing packed program, etc. Perdisci et al. developed a classification system for determining whether an executable is packed or not before sending it to unpacking systems, thus significantly saving the processing time [36].

8 Conclusion

We conduct the first systemic investigation on existing Android packers by examining their major techniques, evaluating their effects on apps, and assessing their effectiveness. We propose and develop *DexHunter*, a novel system for recovering `dex` files from packed apps in both ART and DVM. To our best knowledge, it is the first unpacking system that supports both ART and DVM. The experimental results based on real packed apps demonstrate the effectiveness and efficiency of *DexHunter*. This research reveals important issues in existing Android packers and sheds light on the future research of Android apps protection.

Acknowledgments. We thank the anonymous reviewers for their quality reviews. We thank Yuru Shao and Xian Zhan for their contributions to the preliminary study of this research. This work is supported in part by the Hong Kong GRF (No. PolyU 5389/13E), the National Natural Science Foundation of China (No. 61202396), the PolyU Research Grant (G-UA3X), and the Open Fund of Key Lab of Digital Signal and Image Processing of Guangdong Province (2013GDDSIPL-04).

References

1. ptrace. http://linux.die.net/man/2/ptrace
2. Smali. https://code.google.com/p/smali/
3. Xposed. http://forum.xda-developers.com/xposed/xposed-installer-versions-changelog-t2714053
4. Owasp mobile top 10 risks (2014). http://bit.ly/1FAIJiv
5. Dalvik opcode changes in art (2015). https://github.com/anestisb/oatdump_plus#dalvik-opcode-changes-in-art
6. F-droid (2015). https://f-droid.org/
7. Zjdroid (2015). http://safe.baidu.com/opensec_detail_2.html
8. Alibaba Inc.: http://jaq.alibaba.com/
9. Apvrille, A., Nigam, R.: Obfuscation in android malware, and how to fight back. In: Virus Bulletin, July 2014
10. Arxan Tech., Inc.: Securing mobile apps in the wild with app hardening and runtime protection (2014). http://bit.ly/1aliJil
11. Baidu Inc.: http://apkprotect.baidu.com/
12. Bangcle Inc.: http://www.bangcle.com/
13. Chen, K., Liu, P., Zhang, Y.: Achieving accuracy and scalability simultaneously in detecting application clones on android markets. In: Proceedings of the ACM ICSE (2014)
14. Collberg, C., Nagra, J.: Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection. Addison-Wesley, Upper Saddle River (2009)
15. Crussell, J., Gibler, C., Chen, H.: Attack of the clones: detecting cloned applications on android markets. In: Foresti, S., Yung, M., Martinelli, F. (eds.) ESORICS 2012. LNCS, vol. 7459, pp. 37–54. Springer, Heidelberg (2012)
16. Crussell, J., Gibler, C., Chen, H.: Scalable semantics-based detection of similar android applications. In: Proceedings of the ESORICS (2013)

17. Davies, J., German, D., Godfrey, M., Hindle, A.: Software bertillonage - determining the provenance of software development artifacts. Empirical Softw. Eng. **18**(6), 1195–1237 (2013)
18. Dredge, S.: Android beats IOS for app downloads, but revenues are still a different story (2015). http://bit.ly/1A2conk
19. Egele, M., Scholte, T., Kirda, E., Kruegel, C.: A survey on automated dynamic malware-analysis techniques and tools. ACM Comput. Surv. **44**(2), 1–42 (2012)
20. Frumusanu, A.: A closer look at android runtime (ART) in android L
21. Fung, B.: The time a major financial institution was hacked in under 15 minutes (2015). http://wapo.st/1zcKNj0
22. Gartner Inc.: Debunking six myths of app wrapping (2015). http://gtnr.it/1aGJizc
23. Gibler, C., Stevens, R., Crussell, J., Chen, H., Zang, H., Choi, H.: Adrob: examining the landscape and impact of android application plagiarism. In: Proceedings of the ACM MobiSys (2013)
24. Google: Proguard. http://goo.gl/CLBIkD
25. Google Inc.: ART and Dalvik
26. Grassi, M.: Reverse engineering, pentesting, and hardening of android apps
27. Guo, F., Ferrie, P., Chiueh, T.: A study of the packer problem and its solutions. In: Lippmann, R., Kirda, E., Trachtenberg, A. (eds.) RAID 2008. LNCS, vol. 5230, pp. 98–115. Springer, Heidelberg (2008)
28. Halloway, S.: Component Development for the Java Platform. Addison-Wesley, Boston (2002)
29. IDC.: Android and IOS squeeze the competition (2015). http://bit.ly/17wYoFF
30. Ijiami Inc.: http://www.ijiami.cn/
31. Kang, M., Poosankam, P., Yin, H.: Renovo: a hidden code extractor for packed executables. In: Proceedings of WORM (2007)
32. Lookout Inc.: Mobile threats, made to measure (2014). http://goo.gl/EhJzdt
33. Martignoni, L., Christodorescu, M., Jha, S.: Omniunpack: fast, generic, and safe unpacking of malware. In: Proceedings of the ACSAC (2007)
34. Nigam, R.: Android packers: separating from the pack, June 2014. http://goo.gl/YiULcy
35. Park, Y.: We can still crack you! general unpacking method for android packer (no root). In: Proceedings of the Blackhat Asia (2015)
36. Perdisci, R., Lanzi, A., Lee, W.: Classification of packed executables for accurate computer virus detection. Pattern Recogn. Lett. **29**(14), 1941–1946 (2008)
37. Qian, C., Luo, X., Shao, Y., Chan, A.: On tracking information flows through JNI in android applications. In: Proceedings of the IEEE/IFIP DSN (2014)
38. Qian, C., Luo, X., Yu, L., Gu, G.: Vulhunter: towards discovering vulnerabilities in android applications. IEEE Micro **35**(1), 44–53 (2015)
39. Qihoo360 Inc.: http://dev.360.cn/protect/welcome
40. Rastogi, V., Chen, Y., Jiang, X.: Droidchameleon: evaluating android anti-malware against transformation attacks. In: Proceedings of the ACM ASIACCS (2013)
41. Roundy, K., Miller, B.: Binary-code obfuscations in prevalent packer tools. ACM Comput. Surv. **46**(1), 1–32 (2013)
42. Royal, P., Halpin, M., Dagon, D., Edmonds, R., Lee, W.: Polyunpack: automating the hidden-code extraction of unpack-executing malware. In: Proceedings of the ACSAC (2006)
43. Sabanal, P.: State of the art: exploring the new android kitkat runtime
44. Schulz, P.: Android security analysis challenge: tampering dalvik bytecode during runtime (2013). http://goo.gl/eIszsj

45. Shao, Y., Luo, X., Qian, C., Zhu, P., Zhang, L.: Towards a scalable resource-driven approach for detecting repackaged android applications. In: Proceedings of the ACSAC (2014)
46. Sharif, M., Yegneswaran, V., Saidi, H., Porras, P.A., Lee, W.: Eureka: a framework for enabling static malware analysis. In: Jajodia, S., Lopez, J. (eds.) ESORICS 2008. LNCS, vol. 5283, pp. 481–500. Springer, Heidelberg (2008)
47. Stewart, J.: Ollybone: semi-automatic unpacking on ia-32 (2006). http://goo.gl/LbQYiN
48. Strazzere, T.: android-unpacker (2014). https://github.com/strazzere/android-unpacker
49. Strazzere, T., Sawyer, J.: Android hacker protection level 0 (2014). http://goo.gl/BSKEop
50. Tencent Inc.: http://www.qcloud.com/product/product.php?item=appup
51. Zhang, F., Huang, H., Zhu, S., Wu, D., Liu, P.: Viewdroid: towards obfuscation-resilient mobile application repackaging detection. In: Proceedings of the ACM WiSec (2014)
52. Zheng, M., Lee, P.P.C., Lui, J.C.S.: ADAM: an automatic and extensible platform to stress test android anti-virus systems. In: Flegel, U., Markatos, E., Robertson, W. (eds.) DIMVA 2012. LNCS, vol. 7591, pp. 82–101. Springer, Heidelberg (2013)
53. Zhou, W., Zhou, Y., Jiang, X., Ning, P.: Detecting repackaged smartphone applications in third-party android marketplaces. In: Proceedings of the ACM CODASPY (2012)