IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received September 24, 2018, accepted October 5, 2018, date of publication October 12, 2018, date of current version November 9, 2018.

Digital Object Identifier 10.1109/ACCESS.2018.2875694

DexX: A Double Layer Unpacking Framework for Android

CAIJUN SUN[1], HUA ZHANG[1], SUJUAN QIN[1], NENGQIANG HE[2], JIAWEI QIN[1], AND HONGWEI PAN[3]

[1]State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications, Beijing 100876, China
[2]National Computer Network Emergency Response Technical Team and Coordination Center of China, Beijing 100029, China
[3]Department of Technology, China Minsheng Bank, Beijing 100031, China

Corresponding authors: Hua Zhang (zhanghua_288@bupt.edu.cn), Sujuan Qin (qsujuan@bupt.edu.cn), and Nengqiang He (hnq@cert.gov.cn)

This work was supported by the National Natural Science Foundation of China (NSFC) under Grant 61502044.

ABSTRACT In recent years, many packing services emerge and have been used to protect Android applications by concealing the executable files. However, it also brings some severe problems. For example, Android malwares use packers to escape detection from the anti-virus engine, which makes it harder to filter out the malicious applications. At present, existing Android unpacking exploits are designed complicated and not adaptive for new packers, which makes the unpackers always failed to keep up with the new packing techniques. In this paper, we propose a universal unpacking framework named DexX to extract dex files protected by these packing services. We apply DexX to packed Android applications, the experiment results show that our DexX can extract and recover original executable files (dex files) packed by most well-known commercial packers effectively and accurately.

INDEX TERMS Android packer, Android unpacker, dex extraction, Android malware detection.

I. INTRODUCTION

According to recent worldwide smart phone shipments reported by IDC [1], Android continues to capture 85% of the worldwide smart phone volume in Q1 2017 and worldwide downloads on Google Play exceeded 19 billion in Q4 2017. The rapid growth of Android ecosystem brings great profits for Android economy. Meanwhile Android becomes the ideal target for attackers and a number of sophisticated malicious softwares. Qihoo [2] reports that they have detected more than 7 million Android malwares in 2017 and the affected users are as many as 214 million. Thus more powerful Android malware detection technology [3], [4] is in urgent need.

One of the significant hurdles in anti-virus analysis is the packing techniques used by malwares. A number of security companies proposed their own packing services to protect Android applications from being cracked. However, the malwares can also leverage packing technology to conceal their malicious executable code so that they can evade security filters that based on static analysis [5]–[7]. Lately, a new version of an old malware called *Expensive Wall* [8] has been found on Google Play and the related apps have been downloaded more than 4 million times [9]. The new version of *Expensive Wall* is different from old ones because it is packed. Packer encrypts malicious code so that the anti-virus engines of Google Play Store failed to detect them. Android malwares are becoming more and more resilient because of packing techniques. Symantec [10] reports that the ratio of malwares that leveraging packing techniques increases from 10 % to 25% from Nov. 2015 to Aug. 2016. AVL Team [11] reports that they detected 3357948 packed Android apps and more than 50% of these packed apps are malwares in 2017. A number of anti-virus engines are based on static analysis, which means they need to get the original smali code to conduct analysis. Otherwise they will fail to work. To recover original dex files from packed malwares, some unpacking exploits are proposed (e.g., AppSpear [12], DexHunter [13] and PackerGrind [14]).

However, packers are evolving frequently and existing unpackers always fall back [7]. The reason is that their architectures are coupled with the kernel runtime tightly. Such architectures make the unpackers hard to update. Here we list the limitations of existing unpackers from three aspects. (1). All of them embedded dex extraction logics in the kernel space, which makes the unpackers complicated and not flexible to update. It always costs long time even with tiny update.

When error occurs, researchers have to rewrite program logics and recompile the kernel again. Packers are evolving frequently, which makes most unpacking approaches only work for a limited period or for certain packer versions. (2). Android systems update frequently and some old versions will not be compatible with latest applications, which means unpackers hard coded on old Android systems will be ineffective. It's time-consuming to implement unpackers on new Android systems since different Android versions vary a lot. (3). Above three unpackers stub dex extraction logics in certain kernel functions. Sophisticated packers may invoke customized functions to bypass kernel functions and make the unpackers failed to work.

To develop an effective and universal unpacking tool, we need to address three **challenges**:

C1. How to update the unpacker as fast as the packing techniques evolve?

C2. How to design an architecture evolving as quickly as the Android systems updating?

C3. How to implement the unpacker independent of certain kernel functions which can be bypassed by packers?

Existing Android unpackers cannot fully address above challenges. We design and implement an effective and universal unpacking framework named DexX for ART runtime [15] to solve above challenges. To address challenge **C1**, DexX adopts a double layer architecture that is compatible with different Android systems and able to crack most advanced packing techniques. The architecture contains two layers, Framework Layer and Application Layer. Framework Layer components work at kernel runtime. Their major function is to inject the dex extraction components to application process. Application Layer components work at user space. All the dex extraction components work at Application Layer, which makes implementing DexX like developing a common Android application. This advantage helps DexX solve challenge **C1**. To address challenge **C2** and **C3**, DexX uses Java language to load and initialize classes defined in dex file. The advantages of using Java are two folds. (1). Java is cross-platform language and compatible with almost all Android systems. The components written with Java is more reusable than C/C++. This advantage helps DexX solve challenge **C2**. (2). Since some packers customize class loading functions at native space, loading and initializing classes with Java code do not have to consider how native function works. This advantage helps DexX solve challenge **C3**.

In summary, our major contributions include:

1. We propose the first double layer unpacking framework named DexX with tiny modifications at Android kernel. The extractor loading component works in Framework Layer without any dex extraction logics. The dex extraction components are implemented at Application Layer. When updating program logics of DexX, it's not necessary to modify the kernel, even not to restart the system.

2. We implement DexX for most latest Android versions with ART runtime, including Lollipop-5.1.0_r1, Marshmallow-6.0.0_r1, Nougat-7.1.1_r1 and Oreo-8.1.0_r1. Most of the packed applications we tested can be decrypted and the recovered dex files can be parsed by decompilers successfully. The success of experiments also indicates that the architecture of DexX is compatible with most latest Android versions.

3. We introduce the first systemic evaluation model named Item Recover Ratio (IRR) for unpacking exploits. Through extensive experiments in section IV, the results show that DexX can recover most dex items in packed samples with high accuracy.

The rest of the paper is organized as follows. Section II introduces related background knowledge. Section III presents the technical implementation of our framework DexX, which provides a universal approach to recover the dex files from packed APKs. Section IV evaluates DexX and the experiment results show DexX outperforms other advanced unpackers. Section V discusses related work and Section VI concludes the paper.

II. PRELIMINARIES

A. ANDROID PACKER

A simple runtime packer consists of a process running at the beginning when app starts [17]. Then the process unpacks the protected dex file to memory. After the unpacking process has terminated, the packer launches the original application and switches the execution to it. Through manual analysis, we find packers usually protect dex code from six aspects.

1) ANTI DEBUG

Packers check whether being debugged. Process injection leveraging *ptrace* [18] is frequently used when conducting dynamic debug. Some packers launch multiple threads and let one thread attach to another using ptrace since the process can only be attached by one process [14]. Some packers also check the integrity of APK files in case of debug code injection. Those methods that modifying the packed application and inserting debug code will fail.

2) ENCRYPT DEX FILE

Packers encrypt the original dex code and release dex code at runtime [17]. Those statically reverse-engineering approaches will not work when dex is encrypted. Some packers also split the dex file and stored in non-continuous memory to prevent direct memory dump. Even if the unpacker can locate the dex address, it cannot dump the whole dex file directly.

3) NATIVE METHOD

Packers re-implement Java methods to native ones [14]. The instructions of re-implemented methods are not recognized by Android runtime. They will be invoked by JNI calls and executed by dedicated runtime. For different packers, the patterns vary a lot. Unpackers have to know the patterns of convention when recovering these customized instructions.

4) CUSTOMIZE CLASS LOADING

Packers load classes with customized class loader and load classes dynamically [14], [17]. They hide the class data and recover them in customized manners. Accessing class data directly by offsets defined in dex structure will fail. Unpackers stub into kernel class loading functions will be bypassed since the code may never be executed.

5) DAMAGE DEX STRUCTURE

Packers damage dex structure, such as magic number, item offset, item size, etc [13]. Simple runtime unpacker using these information to collect the content of dex file may fail to dump the dex file.

6) HIDE METHOD INSTRUCTION

Packers extract and hide the method instructions and release them only when class has been initialized. If the hidden instructions are not recovered, the extracted dex file is incomplete and related method bodies are empty.

B. CLASS LOADING

Advanced packers load classes dynamically. Hence when recovering dex files, it is recommended to load and initialize classes to make sure the class data in memory is integrated. Java provides two approaches to load classes, Explicit Loading and Implicit Loading [13]. Java runtime often loads classes through a combination of explicit and implicit manners [19]. For example, a class loader could load a class explicitly at first, then load all referenced classes implicitly.

1) IMPLICIT CLASS LOADING

This occurs when the class is loaded because of inheritance, reference or initialization. For instance, when we are using *new* keyword to create class instance, the runtime will load and initialize the class implicitly.

2) EXPLICIT CLASS LOADING

This occurs when class is loaded by one of the following methods, *ClassLoader.loadClass()* and *Class.forName()*. If the class is already loaded, class loader returns its reference from cache. Otherwise class loader goes through the delegation model [20] to load the class.

In fact, Java framework only provides interfaces, above class loading operations are defined as native functions and implemented at Java runtime of Android kernel. Take Android Lollipop 5.1.0_r1 [21] ART runtime for example, the *new* keyword is compiled to opcode *new-instance* which will be executed by function *AllocObjectFromCode()* eventually. The invocation of *java.lang.ClassLoader.loadClass()* will call the native method *DexFile_defineClassNative()*. The invocation of *java.lang.Class.forName()* will call the native method *Class_classForName()*. The invocation graphs are shown in Fig. 1.

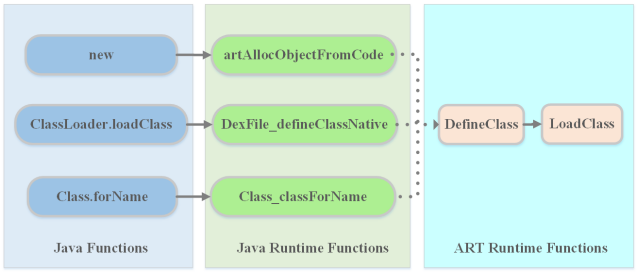

FIGURE 1. Function Call Graph Of Class Loading.

DexHunter selects kernel function *DefineClass()* as key function which is invoked indirectly by both explicit and implicit class loading manner [13]. However, it may fail to extract dex files if packers bypass this function and load classes with customized loading methods. DexX recommends to load classes using explicit manner with Java class loader. DexX select *ClassLoader.loadClass()* as key function to load classes. Reasons are four folds: (1). DexX simulates the application to load classes by using *Java Reflection* [22]. This method is flexible and does not need to know how kernel works. (2). DexX chooses *ClassLoader.loadClass()* instead of *java.lang.Class.forName()* because only the former can use specific class loader. (3). Some packers implement their own class loading functions to bypass the kernel one. This method may hinder unpackers whose logics are embedded in kernel functions, such as AppSpear, DexHunter and PackerGrind. (4). Since Java language is at up level and forward compatible, DexX is more compatible and scalable with different Android systems.

III. DexX: DESIGN AND IMPLEMENTATION

A. BASIC IDEA

The core intuition of DexX is to transparently inject into the target application process and execute DexX extractor. The DexX is part of the target process and can act in its context once injected. After injection, DexX locates the dex file in memory and recovers all the items listed in dex file.

B. ARCHITECTURE

DexX framework employs a double layer architecture and contains two parts, Framework Layer and Application Layer. As shown in Fig. 2, Framework Layer components work at kernel runtime and extended startup framework by loading an additional shared library after application context having been bound. Application Layer components work at user space and are used to extract the dex file. Once the application started, Framework Layer components and Application Layer components become part of the running application.

1) FRAMEWORK LAYER

The Framework Layer includes following three components.

Algorithm 1 Locate Dex File in Memory

Input: Injected object *obj*
Output: Dex location in memory *dex_location*

```
1: void *lib = dlopen(“libart.so”, 0);
2: current_thread = (art::Thread *(*)()) dlsym(lib, “_ZN3art6Thread14CurrentFromGdbEv”);
3: decode_object = (art::mirror::Object *(*)(void *, jobject)) dlsym(lib, “_ZNK3art6Thread13DecodeJObjectEP8_jobject”);
4: art::Thread *self = current_thread();
5: art::mirror::Class *clz = (art::mirror::Class *) decode_object(self, obj);
6: art::mirror::DexCache *dc = (art::mirror::DexCache *) clz→dex_cache_;
7: art::DexFile *dex_file = reinterpret_cast< art::DexFile *>( dc→dex_file_);
8: return dex_location;
```

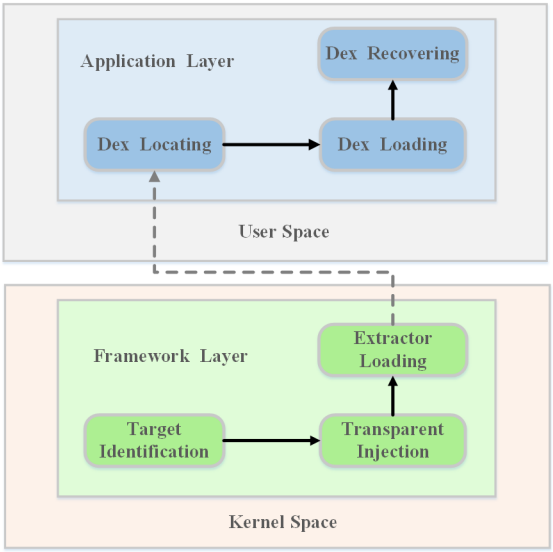

FIGURE 2. Architecture of DexX.

a: TRANSPARENT INJECTION

Advanced packers have anti-debug features built into them to detect debug techniques. They will suspend or exit the program once being debugged. For the purpose of bypassing the debug detection of packers, DexX rewrites the Java framework and integrates process injection program. DexX recommends function *onCreate* of component *android.app.Activity* and *android.app.Service* as injection point. Because both *android.app.Activity* and *android.app.Service* are basic components that widely used when building Android applications. Function *onCreate* is the entrance of these two components and will be called when components are starting.

b: TARGET IDENTIFICATION

Since every application will be monitored when starting, DexX has to filter out the application to be extracted. To identify the target application, DexHunter uses feature strings which is manually analyzed. However, the feature strings change all the time with packers evolving. DexX suggests a more universal technique to identify the target application. DexX uses package name to filter out the target application since the package name is unique for every application running on an Android phone. Once DexX injects into the running application, DexX shares the application context. The package name can be visited by invoking function *getApplicationContext().getPackageName()*.

c: EXTRACTOR LOADING

Once the injected application is identified as extraction target, DexX executes the extractor. DexX extractor contains two parts, Java code and Native code (implemented in C++). We use Java reflection to load the Java part. The Native part is a shared library which will be loaded by function *System.load()*. After both the Java part and Native part loaded, the extractor launches and works at the Application Layer.

2) APPLICATION LAYER

The Application Layer includes following three components.

a: DEX LOCATING

Before extracting the dex file, DexX needs to determine where the dex file is located in memory. There are quite a few methods to locate the dex file, DexX recommends the following solution implemented in Algorithm 1. The structure *art::mirror:Class* contains dex file which it belongs to. DexX can access the DexFile object by visiting constants *class_object→dex_cache_→dex_file_*. The next problem is how to convert Java class to native class type so that DexX can visit its class data. Android NDK tools do not provide such interfaces for application to convert Java class to native type. DexX adopts method which uses functions implemented in shared library *libart.so*. Two critical functions *art::Thread::DecodeJObject()* and *art::Thread::CurrentFromGdb()* will be called. It's recommended to use function *dlopen()* to obtain the symbol address of above two functions. *art::Thread::DecodeJObject()* is used to convert Java object to native object type. Since this function is a member function, DexX invokes it with the thread object it belongs to. The thread object can be found by calling function *art::Thread::CurrentFromGdb()*. Access to private native libraries has been restricted on Android 7.0 or higher. To get around this restriction, it is recommended to list *libart.so* in system/core/rootdir/etc/public.libraries.android.txt before compiling DexX.

b: CLASS LOADING

By Principle, the dex file is a group of classes. DexX can recover the whole dex file through gathering all the classes. Specifically, once injected into the process, DexX is able to access all the dex data as the application does. Most packers modify the offset of object, so visiting the object directly with offset in DexFile structure will lead to errors. The best way is to load and initialize the target object at first, then visit its memory address. The details are described as follows: DexX gets all the class descriptors from dex file in initialization phase implemented by Algorithm 2. Then DexX loads all the classes by using class loader of current application context which is an instance of class *dalvik.system.PathClassLoader*. This class loader contains all the path information of the loaded APK. The approach of loading classes is shown in Algorithm 3.

Algorithm 2 Associate Class and Its Descriptor

Input: DexFile object *dex_file*, class size *class_size*
Output: class id and descriptor mapping *map_list*

```
std::unordered_map < int, std::string> map_list;
for i ← 0 to class_size by 1 do
    DexFile::ClassDef* def = dex_file→GetClassDef(i);
    const char* des = dex_file→GetClassDescriptor(def);
    map_list.insert(std::make_pair(i, des));
end for
return map_list;
```

Algorithm 3 Load Class Object With ClassLoader

Input: class id and descriptor mapping *map_list*
Output: Loaded Java classes *jclasses*

```
HashMap< String,Class> jclasses = new HashMap();
for i ← 0 to class_size by 1 do
    String des = map_list.get(i);
    Class a = Class.forName("android.app.ActivityThread");
    Method m = a.getDeclaredMethod("currentApplication");
    Application app = (Application)m.invoke(null);
    Context context = app.getApplicationContext();
    ClassLoader cl = context.getClassLoader();
    Class cls = cl.loadClass(des);
    jclasses.put(des, jclass);
end for
return jclasses;
```

c: DEX RECOVERING

Since some packers will damage the dex file structure, we need to face two main challenges when extracting the dex file. (1). The first challenge is that some packers split the dex file into pieces and load them into non-continuous memory. When DexX traverses the dex structures, there are chances DexX only gets non-continuous memory pieces, not complete dex file. DexX recommends putting the dex file pieces together after all the items collected. (2). Another challenge is that some packers will modify the magic number, address offset, data size, dex file size, etc. Therefore, it is necessary to fix these items before dumping.

To solve above challenges, DexX conducts following three steps. (1). First, DexX estimates the dex file size by summing up all the items' size. (2). Then, it allocates proper memory to hold the fixed dex file. To hold the collected dex file, DexX will allocate a continuous memory block. (3). Finally, DexX starts to copy the content to newly created memory one by one. Meanwhile, DexX will fix the tampered items and recalculate the new address offset, data size and file size.

According the dex structure [34], the following items will be copied at first: *header*, *string_ids*, *type_ids*, *proto_ids*, *field_ids*, *method_ids*, *class_defs*. Then DexX adjusts following items in header structure for copied dex file: *string_ids_off*, *type_ids_off*, *proto_ids_off*, *field_ids_off*, *method_ids_off*, *class_defs_off*, *data_off_*. These values can be obtained by corresponding items' address minus the dex file begin address in memory. The remained items *data_size_* and *map_off_* will be fixed later.

Since DexX loads all the classes in dex file, it reads the class data by visiting the memory address of each class. DexX uses these data to recover *class_defs*. After all the items are copied, DexX assigns *data_size_*, *map_off_* the new value. Finally we obtain the whole fixed dex file. Modern packers have anti-debug features and will hook functions used to read the memory. Hence, dumping the recovered dex file by calling *fwrite* will failed. Thus DexX recommends using system calls to solve this problem.

IV. EVALUATION

We randomly select 20 applications from F-Droid [23] project, all of them are packed by six commercial packing vendors (Ali [24], Baidu [25], Bangcle [26], Ijiami [27], Qihoo [28], Tencent [29]) on Jan. 2018. In other words, they are all packed by latest packing engines. There are three tools Appspear, DexHunter and PackerGrind that claim to be universal unpacking tools. However, only DexHunter is the open source project and the source code is available on github. Hence, we only compare DexX with DexHunter using above samples.

Android system compilation is done on IBM server, 24 physical processors, processor is Intel(R) Xeon(R) CPU E5-2620 v2@2.10GHz with 6 cores. Server RAM is 160G and operation system is Ubuntu 14.10 LST. Development of DexX is done on Thinkpad s540 notepad, 4 physical processors. The processor is Intel(R) Core(TM) i7-4500U CPU@1.80GHz with 2 cores. Notepad RAM is 8G and operation system is Ubuntu 14.04 LST. All the unpacking tests are conducted on two Nexus phones, Nexus 5 and Nexus 5x. Hardwares of Nexus 5 are Qualcomm Snapdragon S4 Pro

TABLE 1. Packer analysis.

	Anti Debug	Encrypt Dex	Native Method	Customize Class Loading	Damage Dex Structure	Hide Method Instructions
Ali	✓	✓	✓	✓	✓	✗
Baidu	✗	✓	✓	✓	✓	✓
Bangle	✓	✓	✗	✓	✗	✗
Ijiami	✓	✓	✗	✓	✗	✗
Qihoo	✓	✓	✓	✓	✗	✗
Tencent	✓	✓	✗	✓	✗	✓

1.5GHz CPU and 2G RAM. Hardwares of Nexus 5x are Qualcomm Snapdragon 808 1.8GHz and 2G RAM. Android operating systems are KitKat 4.4.3_r1, Lollipop 5.1.0_r1, Marshmallow 6.0.0_r1, Nougat 7.1.1_r1 and Oreo 8.1.0_r1.

A. DATASET

As shown in Table 1, packer samples adopt most advanced protection techniques. We identify the major techniques used in the six packers through manual analysis.

All packers except Baidu **conduct anti-debug check**. Except Baidu packer, all packers are tamper-proof [35]. When the APK is modified or signed with a different key, the application will quit immediately. Ali, Baidu, Bangcle and Ijiami packers check ptrace debug and debug tool cannot attach to the packer process.

All the packers **encrypt dex file**. Part or whole original dex files are encrypted. The encrypted dex content will be released and decrypted at runtime.

Ali, Baidu and Qihoo packers **re-implement some methods to native ones**. For Baidu and Qihoo packers, all the *onCreate* methods in *android.app.Activity*'s sub classes will be converted to native ones. The difference is that Baidu packer does not change the method access flag. Baidu packer implements native method by calling proxy function *A.V()* which is a native method. Qihoo packer changes the access flag and turn the method to native type directly. Ali packer is simalar with Qihoo packer, the difference is that Ali packer chooses methods randomly and turns about 56% methods to native.

All packers **load classes with customized functions**. The class data will be released or recovered only when used. In other words, extracting the dex file by dumping the memory will fail.

Ali and Baidu packers **damage dex structure**. Ali packer even splits dex and load these pieces into non-continuous memory block. These unpackers that dump memory directly will fail. Baidu packer wipes the dex header information. The extracted dex file is not integrated and even can not be parsed if the damaged items are not recovered.

Baidu and Tencent packers **hide method instructions**. Baidu packer removes instructions in *onCreate* methods of classes which extend super class *android.app.Activity* and invokes them by calling *A.V()*. *A.V()* is a native function used to execute hidden instructions. Tencent packer randomly hides method instructions and recover them when class is loaded and initialized.

B. EASY FOR DEVELOPMENT

To our knowledge, DexX is the first Android unpacking framework that can develop so efficient. Other unpackers (e.g., AppSpear, DexHunter and PackerGrind) hard coded the dex extraction logics in system kernel, which is difficult to test and update. They have to recompile the system and reinstall the kernel image even add one extra program statement, let alone debug the unpacker. The cost of updating unpacker is to recompile the whole Android Open Source Project (AOSP) [32] and reinstall the image. We implement DexX on Lollipop-5.1.0_r1, Marshmallow-6.0.0_r1, Nougat-7.1.1_r1 and Oreo-8.1.0_r1 except KitKat-4.4.3 because ART runtime of KitKat-4.4.3 is the product of transition period and has severe compatibility issues. The reason we only select ART runtime to implement DexX is that DVM [30] is out of date. More and more applications require minimum Android SDK version greater than 21. However, the largest version number of DVM is 19 and DVM is already abandoned on Lollipop in 2014. DexHunter is integrated on KitKat-4.4.3_r1. Table 2 and Table 3 show how much time will take to compile DexHunter and DexX with different CPU cores.

TABLE 2. Time consumption of unpacker compilation with 4 CPU cores.

	1st Compile	2nd Compile	3rd Compile
DexHunter	2h, 3m, 15s	0h, 43m, 10s	0h, 41m, 11s
DexX Lollipop	2h, 11m, 1s	0h, 0m, 10s	0h, 0m, 11s
DexX Marshmallow	2h, 30m, 12s	0h, 0m, 12s	0h, 0m, 10s
DexX Nougat	2h, 52m, 17s	0h, 0m, 10s	0h, 0m, 13s
DexX Oreo	3h, 35m, 50s	0h, 0m, 10s	0h, 0m, 11s

The first compiling is time consuming. The second compiling costs less since the compiler will take compiled code of not modified modules instead compiling again. DexHunter's compiling time consists of AOSP compiling and flashing kernel image to smart phone. Both of them are time-consuming work. DexX costs less time because of the double layer architecture. DexX's Framework Layer works in kernel space, its major function is to identify target application and launch DexX extractor. Without too many logics in it, the Framework

TABLE 3. Time consumption of unpacker compilation with 20 CPU cores.

	1st Compile	2nd Compile	3rd Compile
DexHunter	0h, 44m, 8s	0h, 10m, 10s	0h, 11m, 1s
DexX Lollipop	0h, 44m, 11s	0h, 0m, 10s	0h, 0m, 12s
DexX Marshmallow	0h, 57m, 1s	0h, 0m, 12s	0h, 0m, 11s
DexX Nougat	1h, 2m, 1s	0h, 0m, 12s	0h, 0m, 13s
DexX Oreo	1h, 20m, 5s	0h, 0m, 10s	0h, 0m, 12s

Layer seldom updates once compiled. Application Layer works at user space, the program logics are totally independent of the kernel runtime. This means we can develop DexX as developing a common Android application. When update DexX, you do not have to recompile the system image or reboot the system. Just install DexX as a common Android application when update. You can see the update works within few seconds. We tried to update Dexhunter, it costs at least 10 minutes before we see the results. However, DexX reduces the time to 10 seconds.

C. VALIDITY COMPARISON

To check validity, we use decompiler baksmali [31] to disassemble the extracted dex files. The dex file is marked as valid if it can be disassembled by baksmali successfully. The results in Table 4 show valid dex files extracted by DexHunter and DexX. For DexX, all the packers are supported and all the extracted dex files can be parsed by baksmali. The success of experiments also indicates that the architecture of DexX is compatible with most latest Android versions. We mark Ali and Baidu packers with ✓* for DexHunter because extracted dex files are not integrated and cannot be parsed by baksmali. However, we found most of dex items are extracted when viewing them with hex editor. We mark Qihoo packers with ✗* for DexHunter because it only extracts the stub classes of Qihoo packer. We mark Bangcle, Ijiami and Tencent with ✗ for DexHunter because no dex files are extracted.

With the rapid development of packing technology, DexHunter has failed to extract most of the latest packers. Three reasons lead to the failure of DexHunter in this experiment. (1). The core idea of DexHunter is to insert dex extraction code into class loading function in kernel space. However, for latest packers, they may use customized functions loading class instead of the kernel method. (2). DexHunter use *size_* and *location_* constants to locating the dex file. However, packers can wipe away these information. (3). DexHunter relies on feature strings to identify the target application. However, most packers evolve all the time. Even same packer vendor, different packer versions have different feature strings. Through experiments, we find except Bangcle all the packers change their feature strings. To complete the experiments on DexHunter, we analyze the feature strings manually.

D. ITEM RECOVER RATIO EVALUATION

In this paper, we introduce the first unpacker evaluation model **Item Recover Ratio (IRR)**. IRR model helps us understand the limitations of unpacking exploits. IRR model includes the recovered ratios of *String*, *Type*, *Proto Type*, *Field*, *Method*, *Code Item* and *Class Def* of extracted dex file by comparing the original dex. Before we explain the IRR, we introduce **Element Recover Ratio (ERR)** as

$$ERR = \frac{\sum_{j=1}^{m} \frac{E_{orig,j} \bigcap E_{recv,j}}{E_{orig,j}}}{m} \tag{1}$$

For each item, the corresponding elements can be referenced in dex structure. To calculate ERR, we compare the original and recovered elements, the intersection is the correctly recovered elements. Then for each element, we get the correctly recovered proportion. The ERR is the average recovered ratio for specific item. In Equation (1), m means the element amount. Once we get ERR for each item, we can calculate the IRR as

$$IRR = \frac{\sum_{i=1}^{n} ERR_i}{N} * 100\% \tag{2}$$

IRR is the average recovered ratio for specific item. For Equation (2), n is the number of recovered dex files and N is the number of total original dex files.

Since DexHunter fails to extract valid dex files. We only calculate the IRR for DexX in Table 5. The results show that DexX can recover most dex files items. One thing we should point out is that the IRRs of Method and Code Item for Ali packer do not achieve ideal scores. The reason is that Ali packer re-implements nearly 56% methods to native ones. If one method is turned to native method, the ERR of its Code Item will be zero because its Code Item structure is empty. And the ERR of its Method will be 80% because access flags will be modified as native. Although DexX can recover the dex files from packed applications, it has the following limitations and we will tackle them in future work. DexX can only recover Java class data. For those native code protected packers, DexX is not able to extract. DexX is not designed to reverse-engineer native code. However, we will keep on working for the automatic and universal solutions for native code packers.

V. RELATED WORK

Android packing technology evolves all the time. Meanwhile, more and more solutions are proposed to tackle with new challenges [12]–[14]. However, most of advanced unpacking exploits are tightly coupled with the Android kernel and not flexible. Therefore they always fall back and the latest packers can easily evade them.

AppSpear [12] modifies the kernel method of loading class and embeds dex extraction code. It collects required data and reconstructs the dex file when kernel parsing dex file. However, related parsing methods can be bypassed if packers use its own functions. Since AppSpear recovers the dex file

TABLE 4. Valid Dex extracted by DexHunter and DexX.

	Ali	Baidu	Bangcle	Ijiami	Qihoo	Tencent
DexHunter	✓*	✓*	✗	✗	✗*	✗
DexX Lollipop	✓	✓	✓	✓	✓	✓
DexX Marshmallow	✓	✓	✓	✓	✓	✓
DexX Nougat	✓	✓	✓	✓	✓	✓
DexX Oreo	✓	✓	✓	✓	✓	✓

TABLE 5. Item recover ratio evaluation for DexX.

	String	Type	Proto Type	Field	Method	Code Item	Class Def
Ali	99.03%	100%	100%	98.84%	82.98%	43.87 %	100 %
Baidu	99.99%	99.94%	99.99%	99.28%	100%	97.88 %	98.99%
Bangcle	100%	100%	100%	100%	100%	100%	100%
Ijiami	100%	99.96%	100%	98.93%	99.99%	100%	100%
Qihoo	99.99%	99.93%	99.99%	99.24 %	99.86 %	98.75 %	100 %
Tencent	100%	100%	100%	100%	100%	99.94 %	100%

according the dex structure and does not consider dynamic class loading issues [13], [14], [33], it may also dump invalid dex data.

DexHunter [13] takes a similar approach to extract dex file as AppSpear. The significant difference is that DexHunter does not simply extract dex file items according the dex structure, it also loads and initializes all the class objects since some packers would hide class data at first and release them only in class initialization phase. However, some advanced packers hook and bypass system class loading methods and using customized ones. Therefore, the dex extraction logics of DexHunter may not be executed. DexHunter also needs to provide the feature strings to identify target application, but the patterns of feature strings change all the time.

PackerGrind [14] is similar with DexHunter since it also inserts code into kernel functions. But it is more powerful since it monitors all the possible kernel functions. DexHunter only monitors *DefineClass* in ART runtime. PackerGrind monitor following functions, *DexFile* constructor, *DefineClass*, *LoadMethod* and *Invoke*. The benefit is that PackerGrind collects more comprehensive information. The limitation is that the functions monitored by PackerGrind may be bypassed. PackerGrind is based on Valgrind [36], packed applications may detect the existences of PackerGrind and then cease releasing the real code [14]. Another problem is PackerGrind consists 21.3K lines of C/C++ code (Valgrind not included) and 2.5k lines of Python code. We implement DexX with no more than 3000 lines of source code and all the Java source code is reusable. Compared with DexX, PackerGrind is a large project and not easy to maintain and develop.

Existing advanced unpacking exploits embed extraction logics in the kernel space, which makes the unpacker complicated and not efficient to update. However, packers are evolving all the time and most unpacking techniques only work for a limited period or certain versions of packers. The biggest problem is that once the packers aware the methods unpackers used, they will use alternative way to bypass them.

VI. CONCLUSION

In this paper, we design and implement an effective and universal unpacker named DexX for Android dex file extraction. DexX fully addresses three challenges mentioned in introduction section when developing an effective and universal unpacking tool. Since Android malwares use packing technology to escape anti-virus engine, DexX provides a general unpacking solution which facilitates the analysis of malwares. DexX is the first double layer unpacking framework that works for most latest packers. The success of experiments indicate that the architecture of DexX is compatible with most latest Android versions. We also propose the first systematic evaluation model IRR for unpacking exploits. Experiments demonstrate that DexX is able to unpack most Android applications packed by advanced packers.

REFERENCES

[1] IDC Corporation, Framingham, MA, USA. *Smartphone Market Share*. Accessed: Apr. 13, 2018. [Online]. Available: https://www.idc.com/promo/smartphone-market-share/os

[2] Qihoo Corporation, Beijing, China. *Special Report for Android Malwares in 2017*. Accessed: Apr. 13, 2018. [Online]. Available: http://blogs.360.cn/360mobile/2018/03/01/review_android_malware_of_2017/

[3] J. Garcia, M. Hammad, and S. Malek, "Lightweight, obfuscation-resilient detection and family identification of Android malware," *ACM Trans. Softw. Eng. Methodol.*, vol. 26, no. 3, pp. 1–11, Jan. 2018.

[4] M. Zhang, Y. Duan, H. Yin, and Z. Zhao, "Semantics-aware Android malware classification using weighted contextual api dependency graphs," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur. (CCS)*, Nov. 2014, pp. 1105–1116.

[5] Y. Du, J. Wang, and Q. Li, "An Android malware detection approach using community structures of weighted function call graphs," *IEEE Access*, vol. 5, pp. 17478–17486, 2017.

[6] W. Wang, Z. Gao, M. Zhao, Y. Li, J. Liu, and X. Zhang, "DroidEnsemble: Detecting Android malicious applications with ensemble of string and structural static features," *IEEE Access*, vol. 6, pp. 31798–31807, 2018.

[7] Y. Duan *et al.*, "Things you may not know about Android (Un)packers: A systematic study based on whole-system emulation," in *Proc. Symp. Netw. Distrib. Syst. Secur. (NDSS)*, San Diego, CA, USA, Feb. 2018, pp. 18–21.

[8] D. Bird, "Mobile threat," *ITNOW*, vol. 59, no. 4, pp. 46–47, Dec. 2017.

[9] E. Root, A. Polkovnichenko, and B. Melnykov, "ExpensiveWall: A dangerous 'packed' malware on Google play that will hit your wallet," Check Point Softw. Technol., San Carlos, CA, USA, Tech. Rep. 1. Accessed: Jan. 11, 2018. [Online]. Available: https://blog.checkpoint.com/2017/09/14/expensivewall-dangerous-packed-malware-google-play-will-hit-wallet/

[10] S. Aimoto, "Five ways Android malware is becoming more resilient," Semantec Corp., Mountain View, CA, USA, Tech. Rep. 1. Accessed: Jul. 1, 2018. [Online]. Available: https://www.symantec.com/connect/blogs/five-ways-android-malware-becoming-more-resilient

[11] AVL Team. Antiy annual report of mobile security in 2017. Antiy Mobile Security Co., Ltd, Wuhan, China. Accessed: Jul. 1, 2018. [Online]. Available: http://blog.avlsec.com/2018/03/5150/2017-annual-report/

[12] W. Yang *et al.*, "Appspear: Bytecode decrypting and dex reassembling for packed Android malware," in *Proc. Int. Symp. Res. Attacks, Intrusions Defenses (RAID)*, 2015, pp. 359–381.

[13] Y. Zhang, X. Luo, and H. Yin, "DexHunter: Toward extracting hidden code from packed Android applications," in *Proc. Eur. Symp. Res. Comput. Secur. (ESORICS)*, Sep. 2015, pp. 293–311.

[14] L. Xue, X. Luo, L. Yu, S. Wang, and D. Wu, "Adaptive unpacking of Android apps," in *Proc. IEEE/ACM 39th Int. Conf. Softw. Eng. (ICSE)*, May 2017, pp. 358–369.

[15] M. Backes, S. Bugiel, O. Schranz, P. von Styp-Rekowsky, and S. Weisgerber, "ARTist: The Android runtime instrumentation and security toolkit," in *Proc. Secur. Privacy (EuroS&P)*, London, U.K., Apr. 2017, pp. 481–495.

[16] S. Liang, *The Java Native Interface: Programmer's Guide and Specification*. Reading, MA, USA: Addison-Wesley, 1999, pp. 4–5.

[17] X. Ugarte-Pedrero, D. Balzarotti, I. Santos, and P. G. Bringas, "SoK: Deep packer inspection: A longitudinal study of the complexity of run-time packers," in *Proc. IEEE Symp. Secur. Privacy*, May 2015, pp. 659–673.

[18] M. Zheng, M. Sun, and J. C. Lui, "DroidTrace: A ptrace based Android dynamic analysis system with forward execution capability," in *Proc. IWCMC*, Nicosia, Cyprus, Aug. 2014, pp. 128–133.

[19] IBM Corporation, Armonk, NY, USA. *An Introduction to Class Loading and Debugging Tools*. Accessed: Jul. 1, 2018. [Online]. Available: https://www.ibm.com/developerworks/library/j-dclp1/index.html

[20] L. Gong, "Secure Java class loading," *IEEE Internet Comput.*, vol. 2, no. 6, pp. 56–61, Nov. 1998.

[21] *Android Lollipop*, Google Corp., Menlo Park, CA, USA, Jun. 2014.

[22] P. Barros *et al.*, "Static analysis of implicit control flow: Resolving Java reflection and Android intents (T)," in *Proc. Automat. Softw. Eng. (ASE)*, Lincoln, NE, USA, Nov. 2015, pp. 669–679.

[23] F-Droid Community. *F-Droid*. Accessed: Jul. 1, 2018. [Online]. Available: https://f-droid.org

[24] Alibaba Corporation. *Ali Packer*. Accessed: Jul. 1, 2018. [Online]. Available: http://jaq.alibaba.com/

[25] Baidu Corporation. *Baidu Packer*. Accessed: Jul. 1, 2018. [Online]. Available: http://app.baidu.com/jiagu

[26] Bangcle Corporation. *Bangcle Packer*. Accessed: Jul. 1, 2018. [Online]. Available: http://www.bangcle.com/

[27] Ijiami Corporation. *Ijiami Packer*. Accessed: Jul. 1, 2018. [Online]. Available: http://www.ijiami.cn/

[28] Qihoo Corporation. *Qihoo Packer*. Accessed: Jul. 1, 2018. [Online]. Available: http://jiagu.360.cn/

[29] Tencent Corporation. *Tencent Packer*. Accessed: Jul. 1, 2018. [Online]. Available: https://yaq.qq.com/

[30] J. Horton, *Android Programming for Beginners*. Birmingham, U.K.: Packt, 2015, pp. 1–2.

[31] JesusFreke. *Smali/Baksmali*. Accessed: Oct. 14, 2018. [Online]. Available: https://github.com/JesusFreke/smali

[32] *Android Open Source Project*, Google Corp., Menlo Park, CA, USA, 2010.

[33] S. Poeplau, Y. Fratantonio, A. Bianchi, C. Kruegel, and G. Vigna, "Execute this! Analyzing unsafe and malicious dynamic code loading in Android applications," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, vol. 14, Feb. 2014, pp. 23–26.

[34] Android Community. *Dalvik Executable Format*. Accessed: Jul. 1, 2018. [Online]. Available: https: //source.android.com/devices/tech/dalvik/dex-format

[35] Y. Shao, X. Luo, C. Qian, P. Zhu, and L. Zhang, "Towards a scalable resource-driven approach for detecting repackaged Android applications," in *Proc. 30th Annu. Comput. Secur. Appl. Conf.*, Dec. 2014, pp. 56–65.

[36] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," *ACM SIGPLAN Notices*, vol. 42, no. 6, pp. 89–100, Jun. 2007.

CAIJUN SUN was born in Yuyao, Zhejiang, China, in 1990. He received the B.S. degree in software engineering from Hangzhou Normal University, China, in 2013. He is currently pursuing the Ph.D. degree in computer science with the Beijing University of Posts and Telecommunications, China. A primary thrust for his current and future work focuses on malware analysis, and security of mobile systems and apps.

HUA ZHANG received the B.S. degree in telecommunications engineering from Xidian University in 1998, the M.S. degree in cryptology from Xidian University in 2005, and the Ph.D. degree in cryptology from the Beijing University of Posts and Telecommunications in 2008. She is currently an Associate Professor of the Beijing University of Posts and Telecommunications. Her research interests include cryptography, information security, and network security.

SUJUAN QIN was was born in Shijiazhuang, Hebei, China, in 1979. She received the B.S. degree in computer science and technology from the Shijiazhuang Railway Institute in 2003 and the Ph.D. degree in cryptography from the Beijing University of Posts and Telecommunications (BUPT) in 2008. From 2008 to 2010, she held a post-doctoral position in information and communications engineering from BUPT. Since 2011, she has been an Assistant Professor with the Institute of Network Technology, BUPT. She has authored one book and over 70 articles. Her research interests include information security, network security, cryptography, and quantum cryptography.

NENGQIANG HE received the B.S. degree in communication engineering from Beijing Jiaotong University, in 2007, and the Ph.D. degree in electronic engineering from Tsinghua University in 2012. In 2012, he was an Engineer with the National Computer Network Emergency Response Technical Team and Coordination Center of China, where has been a Senior Engineer since 2014. He holds three patents. He was a recipient of the Best Student Paper Award in the 10th International Conference on Mobile and Ubiquitous Multimedia in 2011.

JIAWEI QIN was born in Benxi, Liaoning, China, in 1993. He received the B.S. degree in computer science from Shenyang Aerospace University, China, in 2015. He is currently pursuing the Ph.D. degree in computer science with the Beijing University of Posts and Telecommunications, China. His research interests include malware analysis and security of Android systems.

HONGWEI PAN was born in Hechuan, Chongqing, China, in 1991. He received the B.S. degree in security engineering from Xidian University, China, in 2013. He is currently a Security Engineer with China Minsheng Bank, China. His research interests include information security and vulnerability mining.