The Journal of Systems and Software 140 (2018) 3–16

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

AppSpear: Automating the hidden-code extraction and reassembling of packed android malware

Bodong Li*, Yuanyuan Zhang, Juanru Li, Wenbo Yang, Dawu Gu

Lab of Cryptology and Computer Security, Shanghai Jiao Tong University, Shanghai, China

ARTICLE INFO

Article history:
Received 25 March 2017
Revised 6 January 2018
Accepted 20 February 2018
Available online 21 February 2018

Keywords:
Android security
Code packing technique
Code unpacking
Malware detection

ABSTRACT

Code packing is one of the most frequently used protection techniques for malware to evade detection. Particularly, Android packers originally designed to protect intellectual property are widely utilized by Android malware nowadays to hide their malicious behaviors. What's worse, Android code packing techniques are evolving rapidly with new features of Android system (e.g., the use of new Android runtime). Meanwhile, unpacking techniques and tools generally do not respond to the evolving of packers immediately, which weakens the effectiveness of new malware detection.

To address the unpacking challenge especially for Android packers with advanced code hiding strategies, in this paper we propose AppSpear, an automated unpacking system for both Dalvik and ART. AppSpear adopts a universal unpacking strategy that combines runtime instrumentation, interpreter-enforced execution, and executable reassembling to guarantee the hidden code is extracted and reconstructed as a complete executable. Our experimental evaluation with 530 packed samples shows that AppSpear is able to unpack protected code generated by latest versions of mainstream Android packers effectively.

1. Introduction

Designed to protect the intellectual property originally, Android packers are abused by malicious Android app to evade malware detection. According to a report of Symantec (2016), the ratio of packed malware has increased to 25% by August 2016. Worse still, recent Android malware analysis systems (Li et al., 2017; Wong and Lie, 2016; Mirzaei et al., 2017) often ignore packed malware. As a result, code packing has become a main obstacle against Android malware analysis.

To tackle the packed malicious code, a series of unpacking approaches have been proposed to help security analysis. A major target of existing unpacking tools (Yu, 2014; Android-Unpacker, 2014; Park, 2015) is to dump the Dalvik Executable (`dex`) data in memory and recover the original `dex` binary file. To achieve this target, some tools (e.g., Android-Unpacker, 2014) adopt a straightforward strategy to directly search `dex` file in memory. More advanced tools such as DexHunter (Zhang et al., 2015) and PackerGrind (Xue et al., 2017) recover the `dex` file by monitoring the process of code releasing to collect relevant information. More specifically, DexHunter monitors the class loading process to collect `dex` related data, and PackerGrind defines multiple data collection points to conduct a more comprehensive recovering.

Unfortunately, packers are evolving to thwart such unpacking tools. Countermeasures such as loading `dex` into dispersed memory regions, wiping or modifying part of `dex` are introduced to interfere with straightforward memory dump based unpacking: the malformed `dex` cannot be analyzed or dumped, making these unpacking tools unworkable. Some packers may even actively detect the unpacking behaviors and evade them. For instance, the Ijiami packer (Ijiami, 2017) adds some destructive classes into the original `dex`. When these classes are initialized by unpackers such as DexHunter, the packed app detects the unpacking behavior and stops execution.

Another issue for unpacking is the new execution model of Android. The prevalence of Android Runtime (ART) (Android Statistics and Facts, 2017) also leads to the upgrade of most Android packers. Since ART keeps two versions of code (a native version and a `dex` bytecode version), packers could permanently encrypt the `dex` bytecode version and only release the native version. In this situation, existing unpacking tools cannot deal with the encrypted bytecode in ART.

To study systematically Android code packers especially those new features against existing unpacking tools and propose corresponding new unpacking technique, we first conducted an investigation on 47,962 Android malware samples collected from 2012 to

* Corresponding author.
E-mail address: uchihal@sjtu.edu.cn (B. Li).

https://doi.org/10.1016/j.jss.2018.02.040
0164-1212/

2017, which contain 3258 packed samples and cover eight popular commercial Android packers. Through analyzing these samples, we summarized and classified the code packing techniques into four types: *dex protection, native protection, memory protection* and *code release protection*. Specially, we find many new advanced protections (i.e., *code release protection*) and no existing unpacker can solve them effectively.

Following the summarized features of mainstream Android packers, in this paper we propose AppSpear, an automated Android unpacking system for both Dalvik and ART. AppSpear adopts a universal unpacking strategy that combines runtime instrumentation, interpreter-enforced execution, and executable reassembling. Through a runtime instrumentation of both Dalvik VM and the *Fallback* interpreter of ART, AppSpear collects dex data from Dalvik Data Structs (DDS), which always provide accurate data. In addition, AppSpear enables an interpreter-enforced execution to force the releasing of all bytecode in ART. Since the native execution mode of ART only needs native code that compiled from the original dex bytecode, packers could encrypt the original bytecode after the compilation. To enforce the releasing of the bytecode in ART mode, AppSpear inserts an *execution converter* into ART to enable the interpretive execution for each method. Therefore, the original bytecode of the app is released and the collected DDS is accurate. Finally, AppSpear reassembles the collected DDS into a complete dex file, and this dex file can be sent to state-of-the-art static code analysis tools for further analysis.

To validate the effectiveness of AppSpear, we choose two sets of apps and conduct unpacking test. The first set of apps are 30 open-source apps from F-Droid (2017). The selected apps are packed by six mainstream packers, respectively. Then the 180 samples are sent to AppSpear for dex recovering. The second set are 350 representative packed malware. The results show that AppSpear recovered dex files successfully in both Dalvik and ART. With the ground truth built by the first set of packed samples, we verified that the unpacking results of AppSpear are accurate. More interestingly, we found that malicious behaviors of the malware samples were easily detected after the unpacking. This demonstrated that AppSpear can enhance existing malware detection significantly.

This paper makes the following contributions:

- With an investigation on 3258 packed malware samples, we summarize typical code packing techniques of Android packers especially a number of advanced protections that are not studied before. All protections are classified into four types: *dex protection, native protection, memory protection*, and *code release protection*. Among them, *code release protection* reflects new challenge to all known unpacking tools.
- We propose AppSpear, a universal and automated Android unpacking system for both Dalvik and ART. AppSpear can handle all four kinds of code protections effectively. Particularly, AppSpear is the first unpacker to tackle the *bytecode hiding protection* within ART mode.
- We evaluate AppSpear with two sets of packed samples. The results show that AppSpear conducted successful unpacking against code protected by mainstream Android packers, while existing unpacking tools such as Android-unpacker and DexHunter cannot handle a large portion of them.

2. Execution model of android

2.1. Android app

Android apps are mainly written in Java, C/C++ is also supported with Java native interface (JNI) to interact with apps and framework. When developing an app, developers compile Java source code into bytecode with the javac tool and convert it to dex bytecode with the dx tool. With other resources, the dex file will be zipped into an application package (.apk) file. During the installation of app, Android OS conducts further compilation or optimization locally. Then, the app will be executed within a specific runtime (Dalvik or ART), and each instance runs as an isolated process.

2.2. Dalvik VM

Dalvik virtual machine (Dalvik, 2017) (Dalvik VM) is a discontinued process virtual machine in Google's Android operating system that executes applications written for Android. Dalvik VM is an integral part of the Android software stack in Android versions 4.4 (KitKat) and earlier. While an app's installation, Dalvik can use a tool called dexopt to transform the dex file into optimized Dalvik executable (odex) file. During the execution, Dalvik interprets the dex bytecode. To improve performance, Dalvik also features modular interpretation and just-in-time compilation.

Dex file is the executable of Dalvik. It's composed of certain data sections (Dexformat, 2017), including header, method_id, class_def, data and so on. Dalvik loads the dex file and parses these data sections into corresponding data structs during the runtime. These data structs are defined as Dalvik Data Structs (DDS). Dalvik utilizes these DDS to execute the app. Although the data sections of dex file would be tampered by packers, DDS are crucial data for execution and always give accurate data.

2.3. Android runtime

With the release of Android 5 (Lollipop), Google thoroughly replaced Dalvik with Android Runtime (ART). Unlike the interpreter-based Dalvik, ART is a compilation based runtime, which provides significant performance improvement for Android apps. It first adopts ahead-of-time compilation to transform dex bytecode into native (platform specific) code and then executes the native code. During the installation, ART handles the dex file, which contains dex bytecode compiled from Java source code. ART uses a dex2oat compiler to produce an oat format executable, and then loads it for execution. In addition, necessary classes in boot path are also compiled and pre-initialized to a boot.art file, then mapped into memory during the boot process of the OS. Therefore, the execution is mainly based on compiled native code.

Oat file is a kind of ELF file, but contains two special data sections: *oat data section* and *oat exec section. Oat data section* contains the whole dex file and *oat exec section* contains the compiled native code. When ART loads the oat file into memory, the dex file in it is also parsed. Therefore, ART also maintains the same DDS as Dalvik does.

2.4. Fallback interpreter in ART

Notice that even for the latest version of ART, some dex bytecode is still not able to be complied. Hence the interpretation functionality is still reserved for specific cases. There are some specific methods which are only handled by the interpreter (Dalvik Executable Format, 2017). The main reason is that some mechanisms such as garbage collection, JNI, stack size, and bytecode verification are difficult to be compiled correctly using ART compiler. For example, ART verifies bytecode more strictly at install time than Dalvik does. Bytecode, which contains *invalid control flow, unbalanced moniterenter/moniterexit*, or *0-length parameter type list size*, is not able to be handled by ART and thus cannot be compiled into native code. In this case, the *Fallback* interpreter is adopted. In addition, ART does not compile the code of native methods, abstract methods, and static methods. Instead, ART executes these kinds of methods in the interpreter directly. As a result, the oat file still

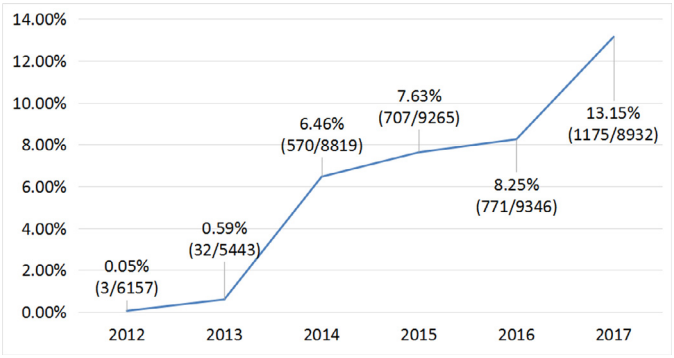

Fig. 1. The ratio of packed Android malware in our experimental samples collected from 2012 to 2017.

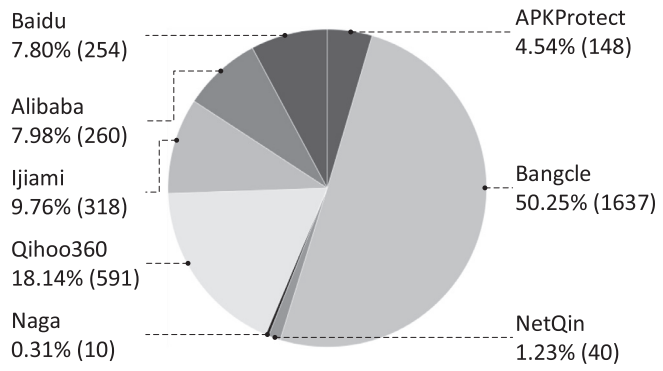

Fig. 2. Distribution of packers used by packed malware.

contains the entire `dex` bytecode. For each method in `oat`, there are two implementations: a native version and a bytecode version. Hence besides the compilation based execution mode, ART also supports interpretive execution mode.

3. Code packing technique

In this section, we summarize Android code packing techniques used by different packers. To understand Android packers and their code packing techniques, we conducted a large-scale investigation on 47,962 malware apps from 2012 to 2017, and found 3258 packed apps from all malware samples. After investigating those packed malware, we found seven new code packing techniques (details in Section 3.4), which have not been discussed by existing researches.

Figs. 1 and 2 show the details of our investigation. Among 47,962 malware apps collected from 2012 to 2017 (we collected most malware from the (Sanddroid, 2017) online Android app analysis system and the other ones from the wild), we obtained 3258 packed apps with signature matching. By learning eight popular Android packers (i.e., Bangcle, 2017; Ijiami, 2017; Qihoo360, 2017; Alibaba, 2017; Baidu, 2017; Naga Security, 2017; Netqin Security, 2017, and `APKProtect`[1]), we built a database of packers' signatures. In particular, we use the unique native `.so` library brought by each packer as its signature.

As Fig. 1 shown, the ratio of packed malware increases significantly year by year. Fig. 2 shows the distribution of different packers used by malware. Among all packers, Bangcle is the most popular one, which corresponds to its market share in Android code protection field.

[1] APKProtect is an obsolete Android packer, which has closed its packing service already.

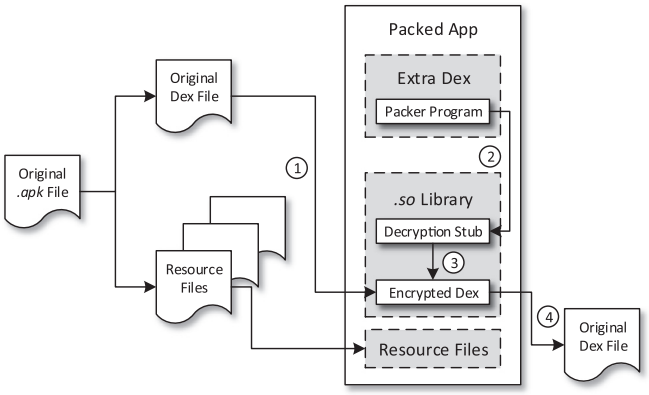

Fig. 3. Process of packed app's production and execution: (1) The packer encrypts the entire `dex` file and hides it into `.so` library. (2) When app starts, the packer program invokes the decryption stub in `.so` library at first. (3) The decryption stub decrypts the encrypted `dex` file in the next. (4) The original `dex` is released and executed at last.

By analyzing packed malware manually, we summarized all the code packing techniques. With the frequent evolution of the packer and the adoption of ART, seven new advanced techniques were found that challenge existing unpackers. We then classified all code packing techniques into four types: *dex protection, native protection, memory protection*, and the recently emerged *code release protection*.

3.1. Dex protection

Early Android packers only protect the `dex` file. Fig. 3 shows a production and execution process of the packed app. To prevent `dex` file from being decompiled by automated tools such as apktool (2017), Android packer firstly encrypts the entire `dex` file and hides it into a `.so` file. Then the packer adds an extra `dex` file, which contains the packer program. At last, the `.so` file, extra `dex` file, and original resource files will be combined into a packed `.apk` file. When the packed app starts to execute, the packer program invokes the decrypting stub in the `.so` file and decrypts the encrypted code dynamically. Then Dalvik VM or ART loads the decrypted `dex` with the help of `DexClassLoader` and executes it at last.

Although *dex protection* prevents the decompilation effectively, it has a major weakness: the protection of native `.so` file is weak. An analyst can conduct a code reverse engineering to decrypt the packed app manually. For the same packer, the decryption algorithm is similar. The analyst can also design a static unpacker which decrypts packed apps automatically in quantities.

3.2. Native protection

To resist manual analysis and static unpackers, Android packers adopt *native protection*. Packers protect the native `.so` file in three aspects: 1) the packer changes cipher key and encryption algorithm periodically, which increases the cost of analysis obviously; 2) an expansive code obfuscation is added to the native `.so` file, which causes an explosion of code logic and hinders static analysis significantly; 3) an extra packer is added to the native `.so` file, i.e., the native `.so` file will also be encrypted entirely. In a word, static unpacking technique cannot handle the frequent evolution of packers with dynamic code encryption/decryption.

In comparison, the original bytecode is still able to be dumped directly from memory using dynamic analysis. No matter how the native `.so` file is protected, during the execution the original `dex`

is decrypted and loaded into memory. Hence many unpackers find the decrypted dex data by searching the magic number of dex ("dex.035" or "dex.036"). Besides, the analyst can obtain the decrypted dex data by monitoring the process of optimization or compilation. After an .apk file is installed, dex file will be optimized into odex file in Dalvik and be compiled into oat file in ART as mentioned in Section 2. Most packers start to work only after the Android system completes this process. By patching Android system's optimization tool (dexopt) or compilation tool (dex2oat), the analyst can obtain the original dex directly. Another valid method is system interface hooking. By hooking the system interfaces for opening a dex file or oat file (*dvmDexFileOpenPartial* in Dalvik or *OatFile::Open* in ART), the decrypted code can be obtained.

3.3. Memory protection

To resist memory dumping based unpackers, Android packers utilize several countermeasures to protect the dex data in memory. We sum up them as follows:

3.3.1. Fake information

Data fields of dex file are crucial to static analysis while some of them are less relevant to dynamic execution. Packers may modify the contents of this kind of data fields to trick unpackers. This modification often confuses unpackers to dump incorrect data from memory. For instance, data fields of DexHeader DDS are often modified by packers:

- magic: As mentioned in Section 3.2, the magic number can be used to locate the start point of dex file. Modifying the magic increases the difficulty of searching dex file.
- fileSize: This data field gives the size of dex file. Without this important information, unpackers have to make a deep analysis of dex file's structure to collect complete dex data.

Data fields in DexMapList DDS can be tampered to confuse unpackers as well. DexMapList is always used to locate other DDS by static analysis tools. But Android system does not need the information in DexMapList. Wrong data in DexMapList would not impact the normal execution. Therefore, some packers tamper DexMapList to confuse unpackers and guarantee app's normal execution as well.

3.3.2. Dispersed code

Some packers release decrypted dex data into dispersed memory regions, which makes the dumped dex data incomplete and malformed. Since DexCode DDS contains dex bytecode directly, most packers give them more protections. Packers would load DexCode into dispersed memory regions and change relevant pointers into exceptional data (i.e., a number larger than dex file size, a negative number, or zero). While the initialization of each Java class, the packers repair the changed pointers in DexClassData, then the system can find each dispersed DexCode in memory.

3.3.3. Environment detection

Packers usually detect the execution environment to prevent unpackers from dumping the real code.

- *Emulator detection*: Since most dynamic analysis tools are based on Android emulator or virtual machine. Packers will check whether the packed app is running in a real mobile device. Most packed apps refuse to execute in an emulator or a virtual machine.

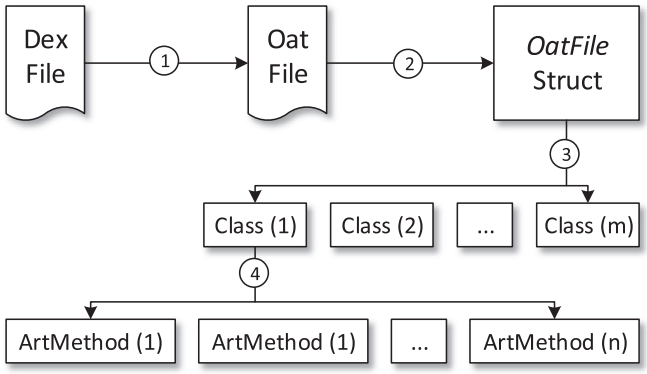

Fig. 4. Code release points in ART. The figure shows the process of parsing an oat file in ART. Packer's code release points are marked: (1) compiling bytecode (2) parsing oat file (3) loading classes (4) loading methods.

- *Anti-debugging*: Some unpackers work as debuggers that attach to the process of the packed app and obtain the bytecode during the execution. The packer would stop the debugger to attach through invoking some system interfaces (e.g., *ptrace*) beforehand.

3.4. Code release protection

Since unpackers can still obtain the actual bytecode by monitoring the code release stage of the app dynamically, *memory protection* is not adequate for hiding the information. Therefore, new packers utilize a *code release protection* to impede the unpacking. By analyzing the latest packed samples, we revealed how latest code packing techniques protect real code during the code release stage. As far as we know, no existing unpackers can handle *code release protection* effectively.

3.4.1. Code release points in ART

With ART's popularity, most packers start to support ART code packing. Due to the differences in design, ART compiles and executes code in a completely different way from Dalvik. Packers need design specialized mechanisms for code's decryption in ART. As Fig. 4 shown, packers select different code release points to prevent this behavior from being monitored by unpackers uniformly. They usually release actual code on the following four occasions:

- *Compiling Bytecode*: During an .apk file's installation, dex file will be compiled into oat file. Some packers utilize dex2oat in system to compile bytecode, and they select to release real code before the compilation.
- *Parsing Oat File*: An oat file will be parsed into the OatFile struct in ART. Some packers select *OatFile::open()* as the code release point.
- *Loading Classes: ClassLinker::LoadClass()* is used to load class data and parse class data into the Class struct. It's also used as a code release point.
- *Loading Methods*: ART uses the ArtMethod struct to represent each Java method. Some packers release each method's code when *ClassLinker::LoadMethod()* is invoked.

3.4.2. Bytecode hiding in ART

Sometimes packers do not release all code at the code release points mentioned above, they would hide some bytecode until they are actually executed. Unlike Dalvik, ART maintains two versions of code (native code and dex bytecode) for each method. In a normal execution, native code is executed by default. As Fig. 5 shown, if method A wants to invoke method B (case 1), A finds ArtMethod B struct at first, then A fetches B's native code and

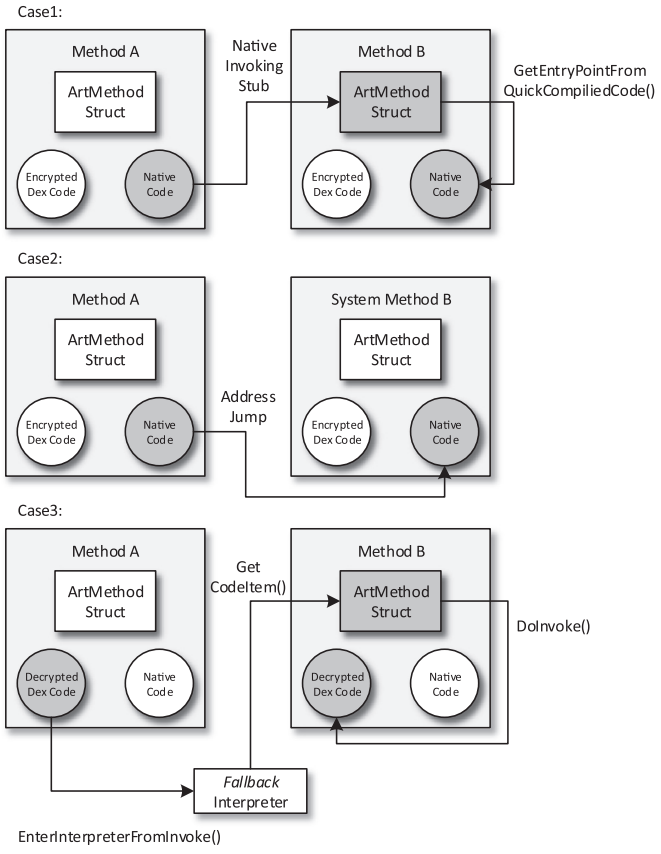

Fig. 5. Bytecode hiding in ART. Case 1 and Case 2 show that `dex` bytecode is hidden during normal executions in ART. Case 3 shows that hidden `dex` bytecode is released before an interpretive execution.

executes it next. Considering the Ahead-of-time (AOT) compilation and function inlining of ART, when B is a system method or a simple method (case 2), it can even be invoked by address jumping directly without accessing `ArtMethod` struct. In these cases, `dex` bytecode would never be invoked. Some packers utilize this feature and keep bytecode encrypted during the execution. Only when methods are executed in an interpretive mode (case 3), hidden code would be just decrypted. When A calls B, packers release the bytecode at first, then the interpreter executes the decrypted code. In this case, packers select *DoInvoke()* in *Fallback* interpreter as the code release point. All existing unpackers would not change app's execution mode, thus they would fail to obtain the real code hidden by packers in ART.

3.4.3. Stepwise code release

Releasing code in multiple steps is also used by packers to prevent the unpacker to obtain the entire code. When the packed app starts to run, only part of the `dex` data is decrypted in the first code release step. Some important data (e.g., instructions of a method) would be decrypted just before its execution. Some packers would even re-encrypt it after its execution. Here we give two examples:

- *Further code release while DefineClass()*: As Fig. 6 shown, in the first code release, the packer releases the original `dex` data, but replaces all methods' instructions with fake instruction `NOP` (opcode:0x00). A method's real instructions are hidden until this method's class object is initialized.
- *Further code release and re-encryption at inserted stubs*: As Fig. 7 shown, the packer inserts two stubs into *onCreate()* and keeps its real code encrypted while the first code release. When the first stub is invoked, the packer decrypts the real code. When the second stub is invoked, the packer would re-encrypt it.

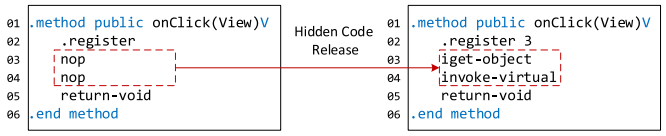

Fig. 6. The 1st example of stepwise code release: While parsing `oat` file, the packer releases `dex` data but replaces instructions with `NOP` (LEFT). When *DefineClass()* is invoked, the packer just releases the real instructions (RIGHT).

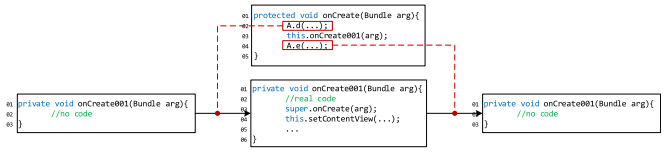

Fig. 7. The 2nd example of stepwise code release: The packer moves the real code of *onCreate()* into *onCreate001()* and inserts two stubs: *A.d()* and *A.e()* into *onCreate()*. The packer hides the real code until *A.d()* is invoked and re-encrypts the code just after *A.e()* is invoked.

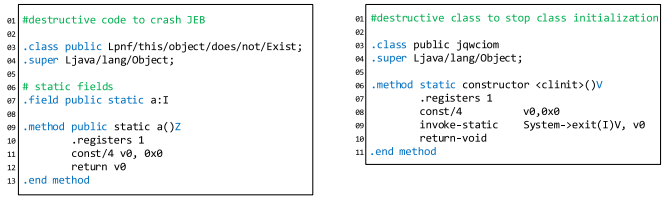

Fig. 8. Destructive code: (1) Code on the left contains an empty static filed named `a` and an empty static method also named `a`. When JEB parses the code, the bugs of JEB would make it to crash. (2) The class on the right contains *System → exit()* in its constructor. The app would exit while the initialization of this class.

3.4.4. Destructive code

Some packers add destructive code into the original `dex` data, which interferes some static and dynamic analysis tools and increases the difficulty of unpacking:

- *destructive code against static analysis*: Static analysis tools have different ways to parse `dex` data and some bugs may exist in their implementations. Packers insert some code to trigger these bugs. Even if the `dex` file is unpacked, it would not be analyzed by static analysis tools. Fig. 8 shows packer Alibaba (2017)'s destructive code (code on the left) which can make Jeb (2017) to crash.
- *destructive code against dynamic analysis*: Since some packers select to release code while class initializations, they inject some classes which can't be initialized to prevent packers from initializing all classes uniformly. These destructive classes will never be invoked in an app's normal execution, but once be initialized, the app will exit. Fig. 8 shows a destructive class we found in packer (Ijiami, 2017) (code on the right).

3.4.5. Native method cheating

Packers sometimes disguise a normal method as a native method to cheat unpackers. By setting the `access_flags` of `ArtMethod` to `ACC_NATIVE`, packers make the methods look like native ones without changing bytecode into native code actually. Meanwhile, the `code_off` is also set as 0. Fig. 9 gives an example of native method cheating used by the Alibaba (2017) packer. If the "native" method is invoked, the packer will revise the `access_flags` and `code_off` to make it back to a normal method.

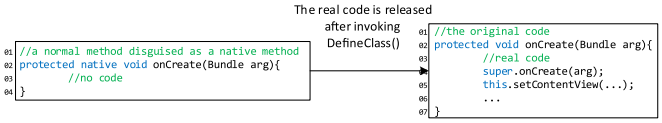

Fig. 9. Native method cheating: The method is disguised as a native method (code on the left). It would be changed back to a normal method after its class object is initialized.

3.4.6. Time checking

We also find time-out checking mechanism added by packers to prevent unpackers from monitoring code release. If the unpacking process introduces an obvious overhead, packers would terminate the execution due to the delay.

3.4.7. Class loader replacement

Packers may use their own class loaders to load classes. In this situation, an unpacker cannot monitor the real class loading through hooking the relevant system interfaces. Hence the actual code release is protected.

4. AppSpear

In this work, we present AppSpear, a universal and automated unpacking system. AppSpear is designed for both Dalvik and ART, and it effectively handles all the code packing techniques mentioned in Section 3. As far as we know, AppSpear is the first unpacking tool that is able to handle all latest protections of mainstream Android packers.

The design of AppSpear needs to address two main challenges: 1) how to overcome kinds of code packing techniques and guarantee its generality. 2) how to obtain `dex` bytecode from ART which is based on native code execution. To address the first challenge, we make use of Dalvik Data Structs (DDS) to obtain accurate `dex` data and `dex` re-writing technique to recover the original `dex` file. DDS are important data structs maintained by Android system. When an app is executed in Dalvik VM, the `dex` file will be parsed for initializing the `DexFile` struct and then the `DexFile` struct will be transformed into a series of DDS for further execution. Although ART does not rely on interpretive execution, it still reserves DDS in its *Fallback* interpreter as mentioned in Section 2.4. Since the execution of an app directly depends on DDS, AppSpear conducts a DDS based unpacking to circumvent protections of packers and guarantees the accuracy of unpacking against intentional code modification of packers. Then AppSpear re-writes the `dex` file through reassembling the collected `dex` data to avoid being confused by fake information generated by the packer.

For the second challenge, we add an *execution converter* into ART to switch each method's execution into interpretive mode, which helps AppSpear overcome the issue of *bytecode hiding in ART* (Section 3.4.2). In the interpretive mode, ART is forced to execute `dex` bytecode, and thus the packer will release the encrypted bytecode. Then AppSpear could retrieve the released bytecode.

Fig. 10 depicts the overall unpacking process of AppSpear. In detail, AppSpear employs the unpacking through three main steps: bytecode recovering (in ART), DDS collecting and DDS reassembling. We provide two environments for unpacking: an instrumented Dalvik VM and an instrumented *Fallback* interpreter of ART. The first step is only performed in ART to solve the second challenge mentioned above. We design an *execution converter* to switch the execution of ART into interpretive mode, which guarantees the real bytecode to be released in ART. Then in the second step, AppSpear collects `dex` date from DDS uniformly in two environments. At last, AppSpear reassembles the collected dex data into a new `dex` file. Our `dex` re-writing technique guarantees the real code's integrity and removes destructive code added by packers. The new `dex` file is then able to be analyzed by most Android app analysis tools and can also be repackaged into an `apk` file.

4.1. Bytecode recovering in ART

To recover original `dex` data in ART is more difficult and challenging than in Dalvik. ART maintains each method with `ArtMethod` struct. `ArtMethod` contains two versions of code (native code and `dex` bytecode). The bytecode comes from the original `dex` file and the native code is generated from the bytecode while the app's installation. To invoke a method, ART utilizes a native invoking stub to find the method's `ArtMethod` struct, then obtains the method's native code. Under normal circumstances, only native code is executed in ART, which makes the bytecode easier to be hidden. Accordingly, packers have more flexible strategies to protect bytecode in ART. As mentioned in Section 3.4.1, various code release points can be selected in ART. Even packers can keep bytecode hidden during the whole execution (i.e., *bytecode hiding in ART* mentioned in Section 3.4.2). They set the bytecode release points into the entry of *Fallback* interpreter. Since *Fallback* interpreter is hardly used while normal executions, these release points are difficult to trigger. *Bytecode hiding in ART* is challenging for dynamic unpacking in ART.

AppSpear uses an *execution converter* to address this challenge. The *execution converter* can switch a method's execution mode between native and interpretive. By setting all methods be executed in the interpreter, the *execution converter* can force all bytecode be executed in ART. No matter how packers protect the bytecode, they have to release the code before the interpretive execution. In this way, the *execution converter* guarantees all bytecode to be released in ART.

The *execution converter* switches each method's execution mode by changing the entry points of method. As Fig. 11 shown, a method in ART has four entry points: two for interpretive invocation and two for native invocation. The default entry points lead the method into native execution (the blue lines in Fig. 11). By switching each method's entry points, the *execution converter* can make ART to execute `dex` bytecode (the green lines in Fig. 11).

Fig. 12 shows how the *execution converter* works. After ART loads the `oat` file, it does some preparations (i.e., loading class, loading method, linking code, etc.) before the following execution. While the period of linking code, the *execution converter* changes each method's execution into interpretive mode. This leads the execution flow into the *Fallback* interpreter. Then the bytecode release points at the entry of interpreter would be triggered and the hidden code would be decrypted and released. In this way, *execution converter* makes the bytecode released and helps the following DDS collecting in ART.

4.2. DDS collecting

AppSpear collects DDS and retrieves accurate `dex` data from them to reassemble unpacked `dex` file. Other existing unpackers mainly obtain the real code from data in memory or high-level data struct (e.g., `DexFile`). However, packers usually hide real code in memory and tamper the data in `DexFile` struct. Although the real code would be released before execution, it is difficult for unpackers to decide the time of unpacking due to packers' different code release occasions. As essential elements of app execution, DDS always provide more accurate data. Since Android system obtains the code to be executed directly through DDS, most DDS must be kept accurate to guarantee the stability of the execution. Although part of DDS are still possible to be tampered, AppSpear can pick them out and abandon the trustless data. In addition, since DDS are a group of independent data structs, the process of

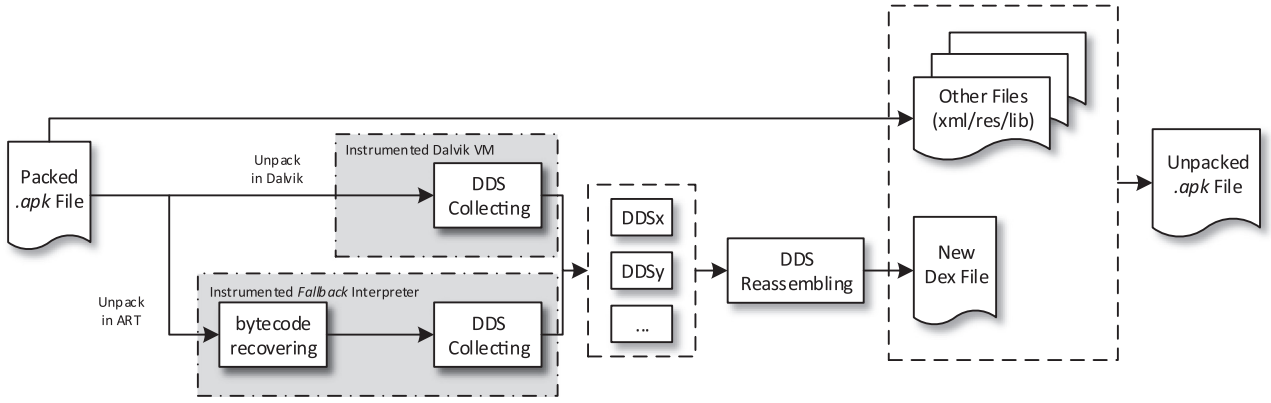

Fig. 10. An overview of AppSpear's unpacking process.

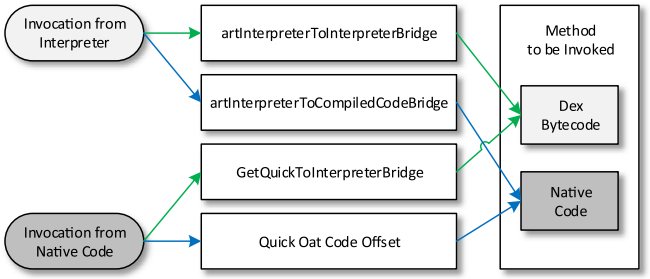

Fig. 11. Entry points of method in ART. The figure shows 4 entry points of method in ART. The native code is executed by default (blue lines). The *execution converter* of AppSpear makes ART to execute bytecode by changing entry points of method (green lines). (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

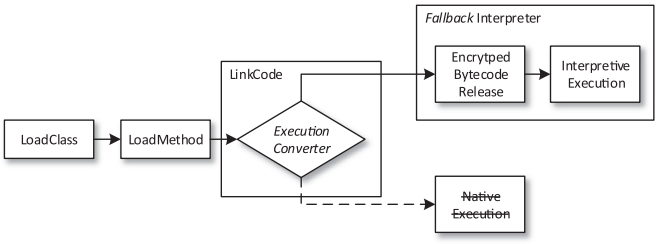

Fig. 12. *Execution converter* of AppSpear. The figure shows how the *execution converter* works. When the period of linking code, the *execution converter* switches all methods' execution mode and leads the execution flow into *Fallback* interpreter.

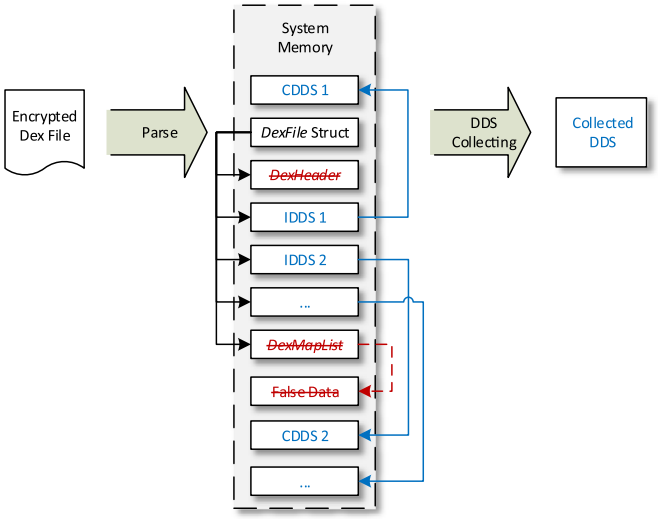

Fig. 13. DDS Accessing: AppSpear collects DDS from IDDS' data offsets (blue lines), and abandons the trustless data in `DexMapList` and `DexHeader` (red lines). (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

collecting DDS is flexible. AppSpear can collect DDS at different occasions to avoid omission. Therefore, AppSpear selects to retrieve `dex` data from DDS to guarantee its accuracy and integrity.

4.2.1. DDS classification

Both Dalvik and ART maintain 18 kinds of DDS. We classify them into two types: the index DDS (IDDS) and the content DDS (CDDS). The IDDS are used to index the real offset of the CDDS, and the CDDS store the detailed bytecode data. The IDDS include `DexHeader`, `DexStringId`, `DexTypeId`, `DexProtoId`, `DexFieldId`, `DexMethodId`, `DexClassDef`, and `DexMapList`. And the CDDS include `DexTypeList`, `DexClassData`, `DexCode`, `DexStringData`, `DexDebugInfo`, `DexEncodedArray` and four items related to Annotation. Since Annotation relevant DDS are seldom related to program's functionality and thus are less important for program analysis. On the contrary, `DexCode` is the most important one, which contains all the bytecode instructions directly. Packers usually give extra protections to it.

4.2.2. DDS accessing

As Fig. 13 shown, the hidden `dex` file would be decrypted and parsed into `DexFile` struct at first. Notice that the data in `DexFile` struct would be intentionally modified by packers. For example, the data offsets in `DexMapList` DDS are likely tampered into false data (red lines in Fig. 13). And the real CDDS data would be released into dispersed spaces in memory (blue lines in Fig. 13).

AppSpear accesses DDS in following steps: We first access `DexFile` struct through *Method* → *clazz* → *pDvmDex* → *pDexFile* in Dalvik and *ArtMethod* → *GetDexFile* in ART. Then, from `DexFile` struct, we obtain certain IDDS' pointers. These IDDS are fixed size structs thus we can obtain their complete data according to their pointers and fixed sizes. Here we abandon IDDS `DexHeader` and `MapList` directly. At last, we traverse all attributes of IDDS to collect accurate offset of CDDS. We need further access the size attribute of each CDDS to determine their real sizes. Notice that the `DexFile` struct also contains pointers of CDDS, but we avoid accessing them directly because of the potential modifications of packers. We show the detailed attributes we used in Fig. 14.

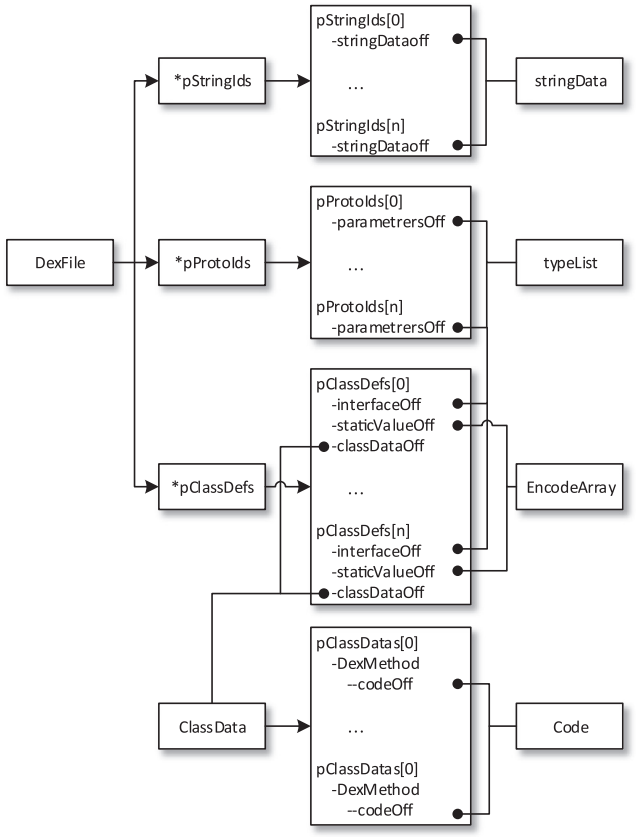

Fig. 14. Detailed attributes we used to access the data content of DDS.

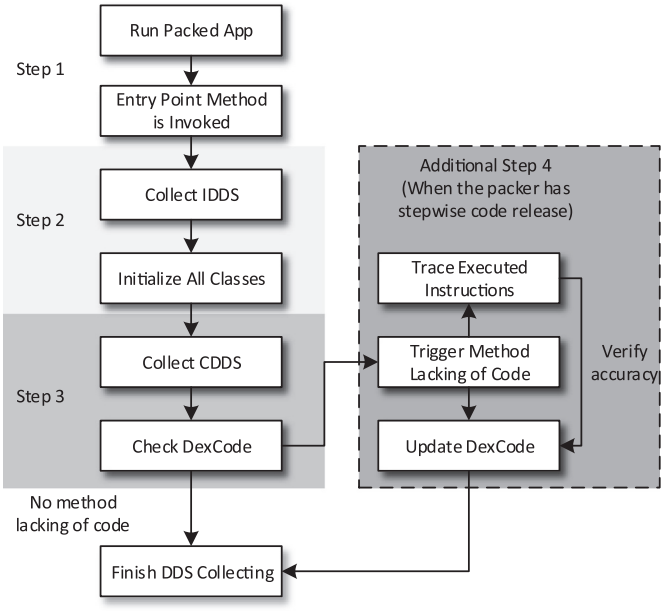

Fig. 15. Process of DDS collecting.

4.2.3. The process of DDS collecting

The process of DDS collecting is shown in Fig. 15. We give the details by steps:

(1) Setting entry point method. We first decide the certain point of execution (denoted as unpacking point) to start our DDS collecting. AppSpear performs instruction-level instrumentations in both Dalvik and ART. By monitoring the `INVOKE` instruction, we could choose arbitrary point to perform collecting, which is significant for fighting against self-modified packers. The default unpacking point is determined by the `manifest` file of app. We choose the main activity as default unpacking point because packers are not allowed to modify the original four components in Android although they can add new `<application>` to the manifests file. Once the interpretation meets the main activity's entry point method (e.g. *onCreate()*), we try to access the `DexFile` struct and start the collection.

(2) Collecting IDDS and initializing classes. At beginning, we first collect IDDS from `DexFile` struct. From the `DexClassDef`, we obtain the names of all classes in the original `dex` file. As mentioned in Section 3.4, packers adopt kinds of *code release protections*. According to our observation, real code is always released after the initialization of class. Therefore AppSpear would enforce to initialize all classes before our CDDS collecting (by *dvmDefineClass()* in Dalvik, *ClassLinker* → *FindClass()* and *ClassLinker* → *EnsureInitialized()* in ART). While class initialization, AppSpear needs to detect and bypass some special classes to prevent crashes. As mentioned in Section 3.4.4, some packers may inject some destructive classes against dynamic analysis which would crash the app. In addition, normal classes with special static code would also cause a crash while enforced initialization, such as static code for thread or dynamic loading operations. By checking the static code of each class before the initialization, AppSpear can effectively bypass all special classes and destructive classes.

(3) Collecting CDDS and checking DexCode. After class initialization, AppSpear starts to collect CDDS. Up to now, we have collected all the DDS. Considering that packers are likely to add extra protections to `DexCode`, AppSpear would check whether any collected `DexCode` contains illegal instructions (e.g. `NOP`) or no instruction. If all the `DexCode` are regular, we would finish the collection. If not, we may add a further step to update the illegal `DexCode`.

(4) Triggering illegal methods and updating DexCode. If AppSpear finds any method lacking of code, it will feedback the information to the analyst by system logs. Then the analyst tries to trigger this method and AppSpear collects the corresponding `DexCode` in the meanwhile. AppSpear also traces the executed instructions of this method by instrumented stubs in Dalvik and ART. AppSpear uses the traced instructions to check the accuracy of new collected `DexCode`. If the instructions in `DexCode` do not correspond to that in traces, it would be repaired using accurate result in traces. At last, AppSpear updates the `DexCode` with new collected ones and finishes the DDS collecting.

Here the method trigger needs manual involvements. Fortunately, there are only few methods that need to be triggered and the process of triggering is not complex. According to our investigation, these special protected methods are always few in number and inserted into the entries of app (*onCreate()* of each activity), which are easy to be triggered. Because packers do not have the source code of the original app. It's really complex for packers to insert code decryption stubs deep into the program. And too many special protected methods would bring more performance loss. Actually, we can trigger most special protected methods by manual operations and Android Debug Bridge (`adb`) tool. Our experimental result in Section 6.1.1 also proves it. Considering the more complex cases we would missed, AppSpear is also compatible with most Android input generators (Wong and Lie, 2016; Choudhary et al., 2015; Mirzaei et al., 2016), which can help to increase our code coverage rate. We will further discuss this in Section 8.

4.3. DDS reassembling

During DDS Reassembling, AppSpear resects destructive code against static analysis, combines all DDS into a new `dex` file, and rebuilds an unpacked `.apk` file in the last. The new generated `dex` file and `.apk` file can be analyzed by most Android app analyzing tools.

4.3.1. Anti-analysis code resecting

As mentioned in Section 3.4.4, packers would leverage bugs of some static analysis tools to inject destructive code that obstruct the normal analysis but do not impact the normal execution. Before we rebuild a `dex` file, we resect these destructive code to help static analysis at first. Since the destructive code is specific and aims at certain analysis tools, it usually has obvious features and is easily detected. We find them by class name checking. For the example on the left of Fig. 8, we first locate it by its specific class name: *"pnf.this.object.does.not.Exist"*, then we check its code and resect the redundant method `a` and field `a`.

4.3.2. Dex rewriting

After resecting the destructive code, AppSpear combines the other DDS together. According to the order of `dex` items defined in *DexFile.h*, we re-order the collected DDS and write them back to the `dex` file in order.

During the rewriting, an important issue is the offset adjustment. DDS maintain many pointers that point to other DDS and the contents of these pointers are reloaded values that represent the offsets at runtime. When this DDS is written back into `dex` file, we should adjust this offset value to a new one that represents the actual offset in `dex` file. We check every pointer of DDS to adjust this offset value when performing `dex` rewriting.

As mentioned in Section 4.2.2, we abandon `DexMapList` and `DexHeader` while collection due to the possible modifications of packers. Therefore we need re-generate these two DDS while `dex` rewriting. The entire `DexMapList` struct stores offsets and sizes of other DDS, we re-calculate all metadata in `DexMapList` during rewriting process. In addition, in case that the packer's modification of certain value, we directly use known knowledge to fill them in `DexHeader` (e.g., size of file, magic number of header).

What's more, during the `dex` rewriting, we should also consider the type difference between `dex` file and DDS. First, in `dex` file the data is 4-byte aligned. Thus during the rewriting, we fill the gap with `NULL` byte if the size of DDS is not 4-byte aligned. Second, in `dex` file the size attribute of `DexClassData` is generally encoded in ULEB128, but its corresponding attribute in DDS is directly stored in a 32-bit variable. The rewriting should transform this 32-bit value with ULEB128 encoding. Finally, in `DexClassData` the id of method and field is the actual value, but when rewriting they should be adjusted into a relative offset to the first id in each `DexClassData`. We would automatically calculate these differences to generate a rewritten `dex` file.

4.3.3. APK repackaging

In the end, we combine the reassembled `dex` file with materials from the existing packed app including `manifests.xml` and resource files to repackage the app. The `manifest` file of an app declares the permissions and the entry points of the app. The declared permissions are directly used in our repackaged app while the entry points should be adjusted. Some packers may modify the main entry point to their decrypting stubs so that they could perform `dex` decryption before the interpretation of interpreter. We would fix this entry point hijacking with the original entry point of the `dex` file.

4.4. Code packing technique solution

As a universal unpacking system, AppSpear can deal with most mainstream Android packers without the detail of their code encryption algorithms. It can solve all existing code packing techniques mentioned in Section 3 effectively.

As a dynamic system, AppSpear obtains decrypted code while app's execution. We do not worry about the complex statical protections: **dex protection** and **native protection** added by packers. Since AppSpear monitors the code release by interpreter instrumentations dynamically instead of dumping data from memory, it can also evade **memory protection**. AppSpear can abandon *fake information* (Section 3.3.1) while DDS collecting and re-generate it while DDS reassembling. Although the real data may be not dex-formated in memory due to *dispersed code* (Section 3.3.2), AppSpear can always find the real code by attributes of DDS. To circumvent *environment detection* (Section 3.3.3) of packers, AppSpear is designed to be deployed on a standard Android device instead of an emulator. In addition, AppSpear adopts a transparent bytecode monitoring and retrieving based on Dalvik VM and ART *Fallback* interpreter instrumentations. Since we monitor at an execution layer without changing the source code, it's transparent to any code level detection. We also do not rely on system provided interfaces (e.g., *ptrace*) to perform debugging and monitoring, which is easily detected by packers. Therefore, AppSpear provides a trustworthy analyzing environment.

AppSpear also performs well when resisting **code release protection**. AppSpear collects real code from DDS. No matter how packers perform code release in Dalvik or ART (Section 3.4.1), the real code is always released while the initialization of DDS as mentioned in Section 4.2. This guarantees that we can obtain the accurate data all the time. AppSpear solves *bytecode hiding in ART* (Section 3.4.2) by the *execution converter* as mentioned in Section 4.1. It can change each method's execution into interpretive mode and enforce the bytecode to be executed. This guarantees the hidden code to be decrypted in ART. AppSpear solves *stepwise code release* (Section 3.4.3) from two aspects: On one hand, AppSpear initializes all classes before the CDDS collecting (the 2nd step of DDS collecting in Section 4.2.3). This can trigger the further code release while *DefineClass()*. On the other hand, if a method's code is still not decrypted after class initialization, AppSpear would perform an additional step to trigger the method and collect its real code (the 4th step of DDS collecting in Section 4.2.3). For *destructive code* (Section 3.4.4), AppSpear would check class's static code and bypass the destructive code against dynamic analysis when class initialization in DDS collecting (the 2nd step of DDS collecting in Section 4.2.3). Then AppSpear would further resect the destructive code against static analysis while DDS reassembling (in Section 4.3.1). According to our observation, the fake "native" methods would be changed back to normal ones while their classes' initializations. When AppSpear initializes all classes in DDS collecting, it solves *native method cheating* (Section 3.4.5) simultaneously. To evade *time detection* (Section 3.4.6), AppSpear utilizes threads to perform the time-consuming operations (e.g., collecting, calculating, and saving data). The instrumented stubs are only used to monitor kinds of events, which provide ignorable overhead. Here we select to create internal threads instead of normal threads because normal thread is not allowed to attach some interpreter-related system functions. This guarantees that AppSpear would not block the main thread of app. Neither packers nor system would detect any delay to exit the app. Since AppSpear does not rely on monitoring the process of class loading, *class loader replacement* (Section 3.4.7) would not impact our unpacking work. Instead, AppSpear starts its unpacking when the entry point method is invoked as mentioned in the 1st step of DDS collecting (Section 4.2.3).

Table 1
Detailed interfaces used by AppSpear while DDS collecting.

Type	DDS	Dalvik	ART
IDDS	DexHeader	Generated from collected data	
	DexStringId	DexFile → pStringIds[i]	DexFile → GetStringId(i)
	DexTypeId	DexFile → pTypeIds[i]	DexFile → GetTypeId(i)
	DexProtoId	DexFile → pProtoIds[i]	DexFile → GetProtoId(i)
	DexFieldId	DexFile → pFieldIds[i]	DexFile → GetFieldId(i)
	DexMethodId	DexFile → pMethodIds[i]	DexFile → GetMethodId(i)
	DexClassDef	DexFile → pClassDefs[i]	DexFile → GetClassDef(i)
	DexMapList	Generated from Collected Data	
CDDS	DexTypeList	dexGetProtoId → parametersOff	ProtoId.parameters_off_
		dexGetClassDef → interfacesOff	ClassDef.interfaces_off_
	DexClassData	DexClassDef → classDataOff	ClassDef.class_data_off_
	DexCode	DexClassData → directMethods[i].codeOff	
		DexClassData → virtualMethods[i].codeOff	
	DexStringData	DexStringId → stringDataOff	StringId.string_data_off_
	DexEncodedArray	DexClassDef → staticValuesOff	ClassDef.static_values_off_
	DexDebugInfo Annotation	Ignored	

5. Implementation

We implement AppSpear in about 20,000 lines of C++ code, we have released the source code on Github (Appspear Source Code, 2017). The implementation has two parts: Dalvik version and ART version. In Dalvik, we add some instrumenting stubs into *dalvik/vm/mterp/out/InterpC-portable.cpp*, in which there are many interpreting handlers for instructions. These stubs can transparently monitor the methods' execution and trace executed instructions. To collect DDS, we select to use internal threads to avoid obvious delay by *dvmAttachCurrentThread()*. The added threads can access the `DexFile` struct through *Method* → *clazz* → *pDvmDex* → *pDexFile*. At the same time, we reuse the Dalvik VM's parsing functions in *dalvik/libdex/DexFile.h* as Table 1 shows.

In ART, we first add an *execution converter* to dynamically switch each method's execution mode from native to interpretive. The *execution converter* works when *ClassLinker::LinkCode()* is invoked. In the period of linking code, the *execution converter* links all methods' native entry points to *GetQuickToInterpreterBridge()* and interpretive entry points to *artInterpreterToInterpreterBridge*, which guarantees that the control flow will go into the interpreter. After certain method's execution, the *execution converter* links this method's native entry point to its own native code entry (code offset in `oat`), and links the method's interpretive entry point to *artInterpreterToCompiledCodeBridge*, which changes the method back to native for less monitoring performance loss.

Similar to Dalvik, there are many interpreting handlers for instructions in *art/runtime/interpreter/interpreter_switch_impl.cc* and *interpreter_goto_table_impl.cc* of ART. We also add some instrumenting stubs into them to transparently monitor the methods' execution and trace executed instructions. To collect DDS in ART, we also create internal threads by *art::Locks::mutator_lock_* → *SharedLock()*. We select to access the `DexFile` struct through *ArtMethod* → *GetDexFile* in ART. At the same time, we reuse the ART's parsing methods in *art/runtime/dex_file.h* as Table 1 shows.

The deployment of AppSpear is simple. For Android system version 4.4 and earlier, only the Dalvik VM's library (*/system/lib/libdvm.so*) is modified. For Android system version 5.0 and later, only the ART's library (*/system/lib/libart.so*) is needed to be modified. Our implementation is compatible to various mainstream Android devices.

6. Evaluation

We use 180 packed samples based on a set of open-source apps from F-Droid (2017) (in Section 6.1) and 350 packed malware apps from Sanddroid (2017) (in Section 6.2) to evaluate the effectiveness of AppSpear. We also validate the performance of AppSpear in Section 6.3. We deploy AppSpear on two devices: Nexus 4 with Android OS 4.4.2 (Dalvik) and Nexus 5 with Android OS 5.0.1 (ART). All packed samples are then tested using these two devices, respectively.

6.1. Experiments with open-source apps

We randomly download 30 apps' source code from F-Droid (2017) and build them into apps. Then we upload the apps to six online commercial Android packing services (i.e., Alibaba, 2017; Baidu, 2017; Bangcle, 2017; Ijiami, 2017; LIAPP, 2017; Qihoo360, 2017) in December 2017 and obtain 180 packed apps. Then we use AppSpear to unpack them in Dalvik and ART, respectively. The result is listed in Table 2.

6.1.1. Unpacking result

As Table 2 shows, AppSpear successfully unpacks all the packed apps both in Dalvik and ART (*Line "Success or Not in Unpacking"*). To check the correctness of our recovered `dex` files, we first parse them by static tool Baksmali (2017) and then compare the detailed `dex` data (`Java` class, `Java` method, bytecode instruction, etc.) with the `dex` files of original apps. Finally, all the recovered `dex` files are successfully parsed by `baksmali` without any exception, and each recovered `dex` file contains the exact classes, methods and instructions of the original app. The result demonstrates the effectiveness of our unpacking approach.

We also give the details while unpacking different packers in Table 2. *Line "Packer Added Class"* shows the average number of extra classes added in each app. In details, the recovered `dex` files of packer Alibaba, Baidu, and Ijiami contain extra `Java` classes. We check the extra classes and find that packers use them to perform their packer programs.

Line "Resected Class" shows the average number of destructive classes resected in each app. While DDS reassembling, AppSpear finds and resects destructive code from recovered `dex` files of packer Alibaba and Ijiami. Packer Alibaba adds two destructive classes[2] to trigger the bugs of JEB and `dex2jar`. Packer Ijiami adds three destructive classes, which are named randomly, to exit the app while the initializations of these classes.

Line "Additional Step" shows whether AppSpear performs an additional step (the 4th step in DDS collecting in Section 4.2.3) while unpacking. *Line "Further Triggered Methods"* shows the average number of methods that need to be triggered in the additional step. AppSpear can finish the unpacking after the first DDS

[2] class: *pnf.this.object.does.not.Exist* and class: *z.z.z.z0*.

Table 2
Experiment on 180 packed apps generated from open source apps in F-Droid.

	Alibaba		Baidu		Bangcle		Ijiami		LIAPP		Qihoo360	
	D	A	D	A	D	A	D	A	D	A	D	A
Success or Not in Unpacking	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Packer Added Classes	59	59	62	62	0	0	5	5	0	0	0	0
Resected Classes	2	2	0	0	0	0	3	3	0	0	0	0
Additional Step	×	×	✓	✓	×	×	×	×	×	×	×	×
Further Triggered Methods	0	0	6.44	6.44	0	0	0	0	0	0	0	0

D: Dalvik A: ART

Table 3
Code packing techniques used by different packers.

	Alibaba	Baidu	Bangcle	Ijiami	LIAPP	Qihoo360
Dex Protection	✓	✓	✓	✓	✓	✓
Native Protection	✓	✓	✓	✓	✓	✓
Fake Information		✓		✓		
Dispersed Code			✓			✓
Environment Detection	✓	✓	✓	✓	✓	✓
Code Release Points in ART	✓	✓	✓	✓	✓	✓
Bytecode Hiding in ART	✓					✓
Stepwise Code Release	✓	✓		✓		✓
Destructive Code	✓			✓		
Native Method Cheating	✓					
Time Detection				✓		
Class Loader Replacement						✓

collecting for most packers. Due to the code release stubs added by Baidu, AppSpear needs to perform an additional step to trigger these stubs and collect the missing code. Fortunately, Baidu only adds code release stubs into *onCreate()* method of each `Activity`. On average, each app has 6.44 methods that need to be triggered. We force to start each activity by adb tool and trigger the code release stubs. Most activities are started normally. There are 4 exceptional activities which need to receive certain `intent`. Although these four apps exit while their activities are forced to start, we still collect the missing code because the packer releases the code before the execution of the original code in *onCreate()*. Finally, we successfully recover all samples of packer Baidu with an additional step of DDS collecting.

Table 2 shows the same results in Dalvik and ART. It demonstrates that packers adopt similar protection strategies in two execution environments. However, there are still some differences in their code release process. For example, packer Alibaba and Qihoo360 perform a *bytecode hiding in ART* (Section 3.4.2), but not in Dalvik.

6.1.2. Analysis of used packers

Using AppSpear, we learn and sum up the detailed code packing techniques used by each packer in Table 3. In general, all the six packers perform strong *dex protection, native protection*, and *memory protection*, which invalidate the unpacking approaches by static ways or memory dumping, and all packers have supported ART. We give the details of each packer below:

- **Bangcle, LIAPP** These two packers adopt a full-code releasing style. LIAPP releases code into continuous memory and Bangcle adopts a *dispersed code* release. AppSpear can collect all relevant DDS just when the app is started.
- **Ijiami**. This packer adds *destructive code* against dynamic analysis, which would crash the app within their initializations. We have given an example in Section 3.4.4 on the right of Fig. 8. Although these classes are randomly named, we can detect them by checking the static code of each class as mentioned in the 2nd step of DDS collecting in Section 4.2.3. This packer also adds a *time detection* while code release. We collect DDS with internal threads to evade its detection as mentioned in Section 4.4.
- **Alibaba**. This packer adds *destructive code* against static analysis, which triggers the bugs of some static tools (e.g., JEB, `dex2jar`). AppSpear can resect it while DDS reassembling. This packer not only hides real code with *native cheating*, but also performs *bytecode hiding in ART*. As far as we know, AppSpear is the only unpacking system which can solve it in ART.
- **Baidu**. This packer adopts a further code release with the stubs added into the *onCreate()* method of each `Activity` class. An example is showed in Section 3.4.3 (Fig. 7). AppSpear solves it by an additional step of DDS collecting to obtain the specially protected code.
- **Qihoo360**. This packer adopts *bytecode hiding in ART* as well. Bytecode is only released while interpretive executions in ART. In addition, it makes a *class loader replacement* to prevent unpackers from monitoring the process of class loading.

6.1.3. Comparison with other unpacking tools

To further illustrate the effectiveness of AppSpear, we compare AppSpear with two representative unpackers: Android-Unpacker (2014) and Zhang et al. (2015). Android-unpacker is an unpacking tool based on dumping data from memory. DexHunter is a general unpacker which is based on code release monitoring. We use the two unpackers to unpack the same samples as AppSpear. Since DexHunter is also suitable for both Dalvik and ART, we also deploy it on devices with different runtime environments. The result is shown in Table 4.

Due to the strong *memory protections* of modern packers, Android-unpacker fails to solve all the packers. DexHunter can solve packer Bangcle and LIAPP in both Dalvik and ART. It can also solve Alibaba in Dalvik but fails in ART due to the *bytecode hiding in ART*. All methods recovered by DexHunter in ART contain no instructions for packer Alibaba. For Baidu, DexHunter provides incorrect data because Baidu adds *fake information* in DDS `DexHeader`. For Ijiami, DexHunter can not work normally due to the *time detection*. For Qihoo360, DexHunter provides code of packer program instead of app's real code, because DexHunter is started by monitoring the class loading process, the packer re-

Table 4
Comparison with Android-Unpacker (2014) and DexHunter (Zhang et al., 2015).

	Alibaba	Baidu	Bangcle	Ijiami	LIAPP	Qihoo360
Android-unpacker	×	×	×	×	×	×
DexHunter in Dalvik	✓	×	✓	×	✓	×
DexHunter in ART	×	×	✓	×	✓	×
AppSpear in Dalvik	✓	✓	✓	✓	✓	✓
AppSpear in ART	✓	✓	✓	✓	✓	✓

Table 5
Experiment on 350 packed malware. Row "Packed" (and "Unpacked") shows the average number of sensitive behaviors in the `dex` file before (or after) AppSpear's unpacking.

Packer	Alibaba	APKProtect	Baidu	Bangcle	Ijiami	Naga	NetQin	Qihoo360	Total
Number of Samples	50	50	50	50	50	10	40	50	350
Packed	0.00	13.64	0.00	6.35	0.88	0.00	5.00	5.00	4.27
Unpacked	86.54	21.36	94.64	113.59	95.04	34.10	85.45	69.74	81.69

places the original `ClassLoader` which confuses DexHunter. Compared with them, AppSpear can successfully solve all packers which proves our effectiveness and applicability.

6.2. Experiments with packed malware

To further evaluate AppSpear's effectiveness and examine whether AppSpear helps malware analysis, we conduct an experiment on 350 packed malware of different packers, which are selected from 3258 packed samples of our investigation in Section 3. We specially select the samples that can be executed in both Dalvik and ART. The number of samples for each packer is shown in Table 5.

We run the packed samples in Dalvik and ART with AppSpear, respectively. All the samples run normally and we successfully obtain 700 recovered `dex` files (350 for Dalvik and 350 for ART). We first use five popular static tools (i.e., DEXTemplate for 010Editor, 2017; Baksmali, 2017; Enjarify, 2017; IDA Pro, 2017 and AndroGuard, 2017) to validate the recovered `dex` files. All the 700 recovered `dex` files are successfully parsed by these five tools. Since we do not have the source code, we compare the `dex` of Dalvik version with that of ART version. We found two versions of `dex` files contain the same code.

We further implement an in-depth static sensitive behavior analysis tools based on AndroGuard (2017). The tool can count the number of malware's sensitive behaviors before and after unpacking. Our tool simply regards the sensitive API calls as sensitive program behaviors (referring to the map of API and permissions in `AndroGuard` (Api permissions.py, 2017)).

The analysis results are shown in Table 5. From the table, we can see that most malware contain few sensitive behaviors before unpacking. Specially, malware packed by APKProtect show the most sensitive behaviors before unpacking. Because APKProtect adopts a part-encryption strategy, part of the original code is exposed even before the unpacking. Every sample packed by NetQin or Qihoo360 has 5 sensitive behaviors before unpacking. This is a coincidence because the two packers' packer program both contain 5 sensitive API calls. However, all the samples show more sensitive behaviors after unpacking. Each sample has 77.42 more sensitive behaviors on average. It proves that AppSpear can effectively help the malware analysis.

6.3. Performance

AppSpear tries to keep well balance between performance and functionality. On the premise of unpacking work, we do not affect the app's execution as far as possible. No obvious lags are felt and no app behaves abnormally while unpacking. To evaluate the overhead, we select three apps and pack them with six different Android packers respectively. These apps' original `dex` files have different file sizes. Then we unpack these 18 packed apps in both Dalvik and ART, and record the time of unpacking process. For each same unpacking process, we perform 10 times and calculate the average value.

Table 6 shows the result of performance evaluation. The second column of Table 6 indicates the file size of original `dex`. And the third column indicates the number of `Java` classes. They are positively associated. The three samples have obvious differences in file size. Column 4–9 give each sample's consuming time of unpacking process. Form the result, we get three conclusions: First, the same sample's consuming time does not change much among different packers. Due to our design principle, we unpack the protected app in an unified way without considering different packers. Therefore, it is reasonable. Second, as `dex` file becomes larger, the consuming time grows obviously. Because while DDS collecting, we need to calculate the size of each DDS. While `dex` rewriting, we need to calculate the new address of each DDS. These two steps take up most of the time. Therefore, a larger `dex` file has more DDS, so needs more time to unpack. Third, unpacking in ART is more efficient than in Dalvik. Because ART itself provides a significant performance improvement as mentioned in Section 2.3. Generally speaking, our unpacking process gives an acceptable consuming time. On the premise of unpacking's accuracy, AppSpear can also provide a perfect performance.

7. Related work

The previous unpacking approaches and tools (e.g., Polyunpack (Royal et al., 2006), Omniunpack (Martignoni et al., 2007), Renovo (Kang et al., 2007), Pandoras Bochs (Böhne, 2008), and Eureka (Sharif et al., 2008)) mainly concern about packers of desktop platforms. Compared with classic Windows and Linux code packers, Android packers are more complex because they involve both native code and `dex` bytecode, which means a packer should consider both aspects and keep the balance between protection strength and stability.

Before our work, a few studies on Android unpacking were proposed. Android-Unpacker (2014), ZjDroid (2016) and some other works (Yu, 2014; Park, 2015) mainly focus on dumping `dex` data from memory. Packers can easily use *memory protecion* to evade these memory dump based unpackers. DexHunter (Zhang et al., 2015) and DWroidDump (Kim et al., 2015) recover `dex` file by monitoring the process of packer's code releasing. DexHunter exploits the class loading process and DWroidDump monitors the process of opening `dex` file. PackerGrind (Xue et al., 2017) updates to multiple data collection points. However, they can also be

Table 6
Consuming time of AppSpear's unpacking process.

Sample	File Size of Original Dex	Number of Java Classes	Average Time of Unpacking in Dalvik(ART) (s)					
			Bangcle	Ijiami	Qihoo360	Baidu	Alibaba	LIAPP
1	457KB	462	1.09(1.08)	1.14(1.03)	1.06(0.98)	1.02 (1.00)	0.98(0.83)	1.18(0.94)
2	2850KB	3018	9.70(2.03)	8.84(1.86)	8.87(2.18)	9.32(2.20)	9.45(1.98)	8.92(2.06)
3	5769KB	5501	19.52(4.80)	18.38(4.44)	17.29(4.53)	17.12(4.88)	18.03(4.67)	18.37(4.79)

Table 7
Comparison of unpackers' capabilities.

		Android-unpacker	DWroidDump	DexHunter	PackerGrind	AppSpear
Supporting ART		×	×	✓	✓	✓
Dex Protection		✓	✓	✓	✓	✓
Native Protection		✓	✓	✓	✓	✓
Memory Protection	T3-1	×	×	×	✓	✓
	T3-2	×	✓	✓	✓	✓
	T3-3	×	✓	✓	✓	✓
Code Release Protection	T4-1	×	×	✓	✓	✓
	T4-2	×	×	×	×	✓
	T4-3	×	×	×	✓	✓
	T4-4	×	×	×	×	✓
	T4-5	×	×	✓	✓	✓
	T4-6	×	×	×	✓	✓
	T4-7	×	✓	×	✓	✓

T3-1: *Fake Information* T3-2: *Dispersed Code* T3-3: *Environment Detection*
T4-1: *Code Release Points in ART* T4-2: *Bytecode Hiding in ART*
T4-3: *Stepwise Code Release* T4-4: *Destructive Code* T4-5: *Native Method Cheating*
T4-6: *Time Detection* T4-7: *Class Loader Replacement*

evaded by latest packers because of *code release protection*. Most of these unpackers don't support ART. Although DexHunter and PackerGrind claim to support ART, they still mainly focus on unpacking in Dalvik and can not deal with *bytecode hiding in ART*. Table 7 summarizes the differences between AppSpear and other unpackers. Compared with them, AppSpear doesn't focus on any code release point, which is easily evaded by packers. Instead, AppSpear collects dex data from DDS while app's execution. As essential elements of bytecode execution, DDS always provide accurate data. At the same time, with the help of *execution converter*, AppSpear can still obtain accurate data in ART despite packers' bytecode hiding strategy. Therefore, AppSpear can successfully solve all code packing techniques effectively.

8. Discussion

AppSpear is based on dynamic analysis, which means it would suffer from the code coverage issue. If packers adopt *stepwise code release* strategy, AppSpear needs to trigger all the code release stubs to collect dex data. Fortunately, according to our investigation, only one packer (i.e., Baidu, 2017) inserts extra code release stubs and the inserted stubs are few in number. These stubs are inserted into fixed positions (*onCreate()* of Activity). Since packers do not have the source code of the app to be protected, it's really complex for packers to insert stubs deep into the program. And too many extra stubs would bring more performance loss. Therefore, we successfully trigger all additional stubs by manual clicks and adb tool. Considering that the packers keep evolving, it's possible to make the stubs difficult to trigger. We can also utilize IntelliDroid (Wong and Lie, 2016) to trigger these stubs. IntelliDroid is a targeted input generator for dynamic analysis of Android malware. It's paired with full-system dynamic analysis systems such as TaintDroid (Enck et al., 2014), as well as AppSpear. As they claimed, AppSpear can have a 93.3% code coverage rate with the help of IntelliDroid.

Malware can employ various anti-analysis techniques for emulator or VM evasion (Petsas et al., 2014). It is feasible that packers can use similar ways to detect our implementation and then hide the decrypting procedure to defeat our unpacking approach. They can utilize our implementation's code features or fingerprint to avoid being analyzed by us. To thwart such evasion, we can also use similar anti-detection measures as emulator evading detection (Hu and Xiao, 2014).

9. Conclusion

This paper describes a systematic study of Android code packing techniques. We summarize and classify the packing techniques into four categories: *dex protection, native protection, memory protection* and *code release protection*. Then we propose a universal and automated unpacking system, AppSpear, which employs a novel bytecode decrypting and dex reassembling approach to replace traditional manual analysis and memory dump based unpacking. AppSpear is designed to support both Dalvik and ART. We further implement AppSpear and evaluate with open-source apps and packed malware apps. Experiments demonstrate that AppSpear is able to resist most latest Android packers' protections with a low performance loss, and it is expected to become an essential supplement of current Android malware analysis.

References

Alibaba Security,2017. http://jaq.alibaba.com/.
Androguard, 2017. https://github.com/androguard.
Android Statistics and Facts, 2017. https://www.statista.com/topics/876/android/.
Android-Unpacker, 2014. https://github.com/strazzere/android-unpacker.
An automatic android application analysis system, 2017. http://sanddroid.xjtu.edu.cn/.
Api permissions.py in androguard, 2017. https://github.com/androguard/androguard/blob/master/androguard/core/bytecodes/api_permissions.py.
Appspear source code, 2017. https://github.com/UchihaL/AppSpear.
A tool for reverse engineering android apk files, 2017. https://github.com/iBotPeaches/Apktool.
Baidu security, 2017. http://apkprotect.baidu.com/.
Bangcle security, 2017. https://www.bangcle.com/.
Böhne, L., 2008. Pandora's bochs: automatic unpacking of malware. phdthesis, University of Mannheim.
Choudhary, S.R., Gorla, A., Orso, A., 2015. Automated test input generation for android: are we there yet? Automated Software Engineering (ASE).
Dextemplate for 010editor, 2017. https://github.com/strazzere/010Editor-stuff/blob/master/Templates/DEXTemplate.bt.

Dalvik executable format, 2017. https://source.android.com/devices/tech/dalvik/dex-format.
Enck, W., Gilbert, P., Han, S., Tendulkar, V., Chun, B.-G., Cox, L.P., Jung, J., McDaniel, P., Sheth, A.N., 2014. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. ACM Transactions on Computer Systems (TOCS).
enjarify, 2017. https://github.com/google/enjarify.
F-droid, 2017. https://f-droid.org/.
Hu, W., Xiao, Z., 2014. Guess where i am-android: detection and prevention of emulator evading on android. HitCon.
Ida pro, 2017. https://www.hex-rays.com/products/ida/.
Ijiami security, 2017. http://www.ijiami.cn/.
Important issues that work on dalvik do not work on art, 2017. https://developer.android.com/guide/practices/verifying-apps-art.html.
Jeb, 2017. https://www.pnfsoftware.com/.
Liapp on-site, 2017. https://liapp.lockincomp.com/.
Kang, M.G., Poosankam, P., Yin, H., 2007. Renovo: a hidden code extractor for packed executables. In: Proceedings of the 2007 ACM workshop on Recurring malcode.
Kim, D., Kwak, J., Ryou, J., 2015. Dwroiddump: executable code extraction from android applications for malware analysis. International Journal of Distributed Sensor Networks.
Li, Y., Jang, J., Hu, X., Ou, X., 2017. Android malware clustering through malicious payload mining. RAID.
Martignoni, L., Christodorescu, M., Jha, S., 2007. Omniunpack: fast, generic, and safe unpacking of malware. ACSAC.
Mirzaei, N., Garcia, J., Bagheri, H., Sadeghi, A., Malek, S., 2016. Reducing combinatorics in gui testing of android applications. Software Engineering (ICSE).
Mirzaei, O., Suarez-Tangil, G., Tapiador, J., de Fuentes, J.M., 2017. Triflow: triaging android applications using speculative information flows. AsiaCCS.
Naga security, 2017. http://www.nagain.com/.
Netqin security, 2017. http://cn.nq.com/.
Park, Y., 2015. We can still crack you! general unpacking method for android packer (no root). Black Hat Asia.
Petsas, T., Voyatzis, G., Athanasopoulos, E., Polychronakis, M., Ioannidis, S., 2014. Rage against the virtual machine: hindering dynamic analysis of android malware. In: Proceedings of the Seventh European Workshop on System Security.
Qihoo360 security, 2017. http://jiagu.360.cn/.
Royal, P., Halpin, M., Dagon, D., Edmonds, R., Lee, W., 2006. Polyunpack: automating the hidden-code extraction of unpack-executing malware. ACSAC.
Smali and baksmali, 2017. https://github.com/JesusFreke/smali.
Symantec Report: "Five ways Android malware is becoming more resilient", 2016. https://www.symantec.com/connect/blogs/five-ways-android-malware-becoming-more-resilient.
Sharif, M., Yegneswaran, V., Saidi, H., Porras, P., Lee, W., 2008. Eureka: a framework for enabling static malware analysis. European Symposium on Research in Computer Security.
The dalvik runtime is no longer maintained or available [in latest versions of android and its byte-code format is now used by art.], 2017. https://source.android.com/devices/tech/dalvik/index.html.
Wong, M.Y., Lie, D., 2016. Intellidroid: a targeted input generator for the dynamic analysis of android malware.. NDSS.
Xue, L., Luo, X., Yu, L., Wang, S., Wu, D., 2017. Adaptive unpacking of android apps. In: Proceedings of the 39th International Conference on Software Engineering.
Yu, R., 2014. Android packers: facing the challenges, building solutions. In: Proceedings of the 24th Virus Bulletin International Conference.
Zhang, Y., Luo, X., Yin, H., 2015. Dexhunter: toward extracting hidden code from packed android applications. European Symposium on Research in Computer Security.
Zjdroid. 2016. https://github.com/halfkiss/ZjDroid.