AppSpear: Bytecode Decrypting and DEX Reassembling for Packed Android Malware*

Yang Wenbo[1(✉)], Zhang Yuanyuan[1], Li Juanru[1], Shu Junliang[1]
Li Bodong[1], Hu Wenjun[2,3], Gu Dawu[1]

[1] Computer Science and Engineering Department,
Shanghai Jiao Tong University, Shanghai, China
[2] Xi'an Jiaotong University, Xi'an, Shaanxi, China
[3] Palo Alto Networks, Singapore

Abstract. As the techniques for Android malware detection are progressing, malware also fights back through deploying advanced code encryption with the help of Android packers. An effective Android malware detection therefore must take the unpacking issue into consideration to prove the accuracy. Unfortunately, this issue is not easily addressed. Android packers often adopt multiple complex anti-analysis defenses and are evolving frequently. Current unpacking approaches are either based on manual efforts, which are slow and tedious, or based on coarse-grained memory dumping, which are susceptible to a variety of anti-monitoring defenses.

This paper conducts a systematic study on existing Android malware which is packed. A thorough investigation on 37,688 Android malware samples is conducted to take statistics of how widespread are those samples protected by Android packers. The anti-analysis techniques of related commercial Android packers are also summarized. Then, we propose AppSpear, a generic and fine-grained system for automatically malware unpacking. Its core technique is a bytecode decrypting and Dalvik executable (DEX) reassembling method, which is able to recover any protected bytecode effectively without the knowledge of the packer. AppSpear directly instruments the Dalvik VM to collect the decrypted bytecode information from the Dalvik Data Struct (DDS), and performs the unpacking by conducting a refined reassembling process to create a new DEX file. The unpacked app is then available for being analyzed by common program analysis tools or malware detection systems. Our experimental evaluation shows that AppSpear could sanitize mainstream Android packers and help detect more malicious behaviors. To the best of our knowledge, AppSpear is the first automatic and generic unpacking system for current commercial Android packers.

Keywords: Code protection, Android malware, DEX reassembling

* This work is partially supported by the National Key Technology Research and Development Program of China under Grants No.2012BAH46B02, the National Science and Technology Major Projects of China under Grant No.2012ZX03002011, and the Technology Project of Shanghai Science and Technology Commission under Grants No.13511504000 and No.15511103002.

1 Introduction

As Android malware emerges rapidly, more and more malware detection techniques leverage in-depth program analysis to help understand program and detect malicious behaviors automatically. A range of static and dynamic analysis approaches (e.g., using machine learning techniques to detect malware, using code similarity comparison to classify malware families) have been proposed for detecting malicious Android apps and the progress is significant [30] [7] [11]. To thwart program analysis based automated malware detection, malware authors gradually adopt code protection techniques [8] [26]. Although these techniques are initially designed to counter reverse engineering and effectively resist many program tampering attempts, they are becoming a common measure of malware detection circumvention. Among various code protection techniques, the most popular one is the code packing technique, which transforms the original app to an encrypted or obscured form (a.k.a "packed app"). According to the report [3] released by AVL antivirus team, among over 1 million Android malware samples they detected, the number of code packed malware is about 20,000. Unfortunately, current program analysis techniques and tools do not consider the code packing issue and could not perform effective analysis task on packed code, and thus are not able to detect those kinds of packed malware statically and automatically. In addition, anti-debugging code stubs are frequently injected into a packed app to interfere dynamic analysis based sandbox detection system. In a word, code packing is becoming a main obstacle for the state-of-the-art automated Android malicious code analysis.

To response, this paper conducts a systematic study of packed Android malware, and our work examines the feasibility of universal and automated unpacking for Android applications. The goals and contributions of this paper are twofold: First, we conduct a thorough investigation on large-scale Android malware samples to take statistics of how widespread those malware samples are protected by Android packers. We start the investigation from studying 10 popular commercial Android packers used by malware authors frequently, which cover the majority of existing Android packing techniques, and summarizing the anti-analysis techniques of those commercial Android packers. We then conduct the investigation among 37,688 Android malware samples, which contain 490 code packed malware. The dataset is accumulated from an online Android app analysis system–SandDroid [5] lasting for more than three years in collecting related packed malware samples. To the best of our knowledge, this is the first in-depth investigation on code packed Android malware. Second, to address the challenge of analyzing code packed malware, we propose **AppSpear**, a generic and fine-grained unpacking system for automatically bytecode decrypting and Dalvik executable (DEX) reassembling. As our investigation demonstrates, commercial Android packers are evolving rapidly. Packers' ongoing evolution leads to an endless arms race between packers and unpackers, and it requires a non-trivial amount of efforts for security analysts to tackle this problem. Since the amount of packer and malware increases at a significant speed, manual unpacking is not feasible for large-scale packed malware analysis. To avoid decrypting packed

code through manually comprehending different packing algorithms, AppSpear directly instruments the execution to extract all runtime Dalvik Data Structs (DDS) in memory and reassembles them into a normal DEX file. The purpose of AppSpear is to automatically rebuild the code packed app into its normal form so that this rebuilt app is able to be analyzed by program analysis tools. A bytecode decrypting and DEX reassembling process for code packed malware is executed to automatically reverse code protection techniques of Android packers.

Previous unpacking approaches [28] [25] [17] mainly focus on dumping the loaded DEX data in memory directly to recover the original DEX file. To thwart such memory dump based unpacking, new advanced packers would reload the DEX data into inconsecutive memory regions and modify relevant pointers that point to the data, which leads to a malformed dumped data. Moreover, some information in DEX is crucial to static analysis tools but is irrelevant to dynamic execution (e.g., metadata in DEX file Header). Packers could wipe or modify this kind of information in memory, which makes it difficult to locate a DEX file in memory. Hence, AppSpear adopts a more comprehensive runtime information reassembling approach rather than the simple memory dumping approach. It rebuilds a packed malware through three main steps: First, AppSpear leverages Dalvik VM introspection to circumvent anti-debugging measures of the packer and transparently monitors the execution of the packed app. During the monitoring, it records execution traces and runtime Dalvik Data Structs (DDS) in memory as raw materials for the next step. Second, AppSpear makes use of a proposed DEX reassembling technique to reassemble the collected materials into a normal DEX that is suitable for static analysis. Third, AppSpear makes use of an APK rebuilding technique to re-generate an APK file with inserted anti-analysis code resected.

To validate AppSpear, we first employ all code protection techniques of seven currently available online Android app packing services to pack our test app, and then use AppSpear to unpack the packed samples. AppSpear is able to decrypt every sample and output the reassembled app that corresponds to the original test app well. Further, among the 490 packed malicious apps in all collected 37,688 samples, we select 31 representative samples that are able to execute and use AppSpear to unpack them. All of those samples can be decrypted and reassembled by AppSpear, and the rebuilt apps expose obvious malicious behaviors that could not be detected by static app analysis tools (e.g., AndroGuard) before.

This paper makes the following contributions:

- We perform a thorough investigation on both existing mainstream Android packers and code packed Android malware in the wild. We further summarize typical anti-analysis defenses of Android packers.
- We propose a bytecode decrypting and DEX reassembling technique to rebuild protected apps. Our APK rebuilding process transforms a code packed malware to an unpacked one, which is a feasible form for commodity program analysis.

- We design an automated and generic unpacking system, AppSpear. AppSpear can deal with most mainstream Android packers and the unpacked apps can be validated by state-of-the-art analysis tools, which are not able to handle the packed form beforehand.

We detail on the investigation of existing Android packers and code packed malware in Section 2 and on our proposed unpacking technique in Section 3. The experimental evaluation is reported in Section 4. Before concluding in Section 7 we discuss related work and possible limitations in Section 6 and Section 5.

2 Code Packed Android Malware

The purpose of our investigation includes: **(a)** to find out the ratio of code packed malware in the wild, and **(b)** to understand the anti-analysis defenses used by those packers. We conduct a large-scale investigation on 37,668 malware samples collected from the SandDroid online Android app analysis system from 2012 to May 2015. Then we analyze and summarize the anti-analysis techniques used by popular commercial Android packers.

2.1 Investigation

To judge whether a malware sample is packed, and which packer it used to protect itself, we adopt a signature based identification strategy to detect code packed malware. We observe that each commercial Android packer brings its unique native *.so* library, which can be used as the signature of that packer. We first investigate 10 popular commercial Android packers (*Bangcle*, *Ijiami*, *Qihoo360*, etc) and build a signature database. Then, we collect 37,668 malware samples from 2012 to May 2015 using SandDroid, which detects malware according to the feedback results of 12 main virus scan engines from VirusTotal (F-Secure, Symantec, AntiVir, ESET-NOD32, Kaspersky, BitDefender, McAfee, Fortinet, Ad-Aware, AVG, Baidu-International, Qihoo-360). An app is regarded as malware if more than three virus scan engines detect it.

Table 1. Summary of Packed Android Malware

(a) Annual statistics

Year	Malware collected	Packed	Ratio
2012	16157	6	0.04%
2013	15443	89	0.58%
2014	5819	376	6.46%
2015	249	19	7.63%

(b) Distribution of packers

Packer	Number of Samples
APKProtect	37
Bangcle	402
NetQin	10
Naga	1
Qihoo360	23
Ijiami	27

As Table 1(a) shows, the amount of packed malware increases significantly since 2014. The distribution of packer type used by malicious apps is showed in

Table 1(b). Among those samples, *Bangcle* becomes the most welcome packer, which corresponds to its market share in Android code protection field.

Although most commercial Android packing service providers have stated that every submitted app is first checked by various antivirus products, we still find malware samples protected by those packers. We believe that no packing service provider could prove the accuracy of malware detection. Malicious app may not be detected at the time it was submitted due to the updating latency of the used antivirus products. In this situation, packing services may help produce code packed Android malware in the wild.

2.2 Anti-analysis Defenses

Android packers often use a variety of defenses to hinder analysis. To comprehend how Android packers obstruct program analysis, we manually analyze 10 commercial Android packers that provide public online packing services. Our analysis indicates that anti-analysis defenses employed by those packers can be classified into three categories. The first category of anti-analysis defenses involve functions that check the static and dynamic integrity of the app (i.e., whether the app is patched or injected with debugging routines). These measures can be easily circumvented if analysts know the tricks beforehand. The second category of anti-analysis measures involve source code level obfuscation, which requires the source code to employ the protection. The third category, which is most complex, involves bytecode hiding.

2.2.1 Integrity Checking Packers generally check the integrity of their packed apps to decide whether the apps are tampered. They check both the integrity of the static code and the dynamic process. For static code integrity checking, packers often calculate the checksum of the code part to determine whether the code is modified. Specifically, for Android app the certificate of the APK is validated by many packers. For dynamic process integrity checking, packers often calculate the checksum of DEX data loaded in memory at runtime. Moreover, they also detect the existence of debuggers or emulators. Besides the traditional anti-debugging tricks used in desktop Linux system (e.g., to fork subprocesses and $PTRACE$ one other, to check */proc/self/status* or */proc/-self/wchan*), some packers hook the *write* and *read* syscalls to thwart memory dump based DEX data acquiring. They check whether the code region is accessed or manipulated. If so, such operations will be abandoned.

2.2.2 Source Code Obfuscation Many developers would obfuscate their source code before their apps are released. Because most Android apps are written in Java, classic Java code obfuscation techniques can be directly employed on Android app. Those techniques mainly include:

- **Identifier mangling**: renaming class names, method names and variable names as meaningless strings or even non-alphabet unicode.
- **String obfuscation**: replacing static-stored strings with dynamic generated ones.
- **Reflection**: hiding method invoking using Java reflection mechanism.
- **Junk code injection**: injecting useless code to change original control flow.

- **Goto injection**: using *goto* to make control flow hard to understand.
- **Instruction replacing**: using a set of instructions to replace one instruction while keeping the semantic of the replaced instruction.
- **JNI control flow transition**: using JNI invoking to hide the real control flow.

Source code obfuscation requires the involving of developers during the development stage. The main problem for source code obfuscation is that it does not provide enough protection strength to counter bytecode level program analysis. Most source code obfuscation techniques only increase the comprehension complexity for manual reverse engineering. Malicious code, which mainly needs to hinder automated program analysis based detection, requires more sophisticated protection.

2.2.3 Bytecode Hiding When published, the Java source code of an Android app is first compiled with standard Java compiler into Java bytecode files (*.class* files), and these files are then transformed into a DEX file with the *dx* tool provided by Google. Information of bytecode is thus contained in this DEX file. To prevent the analyst from acquiring bytecode information from the app, packers modify the original executables to thwart state-of-the-art analysis tools. Typical measures include *metadata modification* and *DEX encryption*.

In Android app, many metadata could be modified without affecting the normal execution, but the modification significantly affects certain analysis tools. Packers would sabotage program analysis through modifying some crucial metadata of the APK file to create malformed executables, and leverage this as an effective defense to counter analysis. Typical metadata modification measures include:

- **Manifest cheating** [21]: modifying the binary form of the *Manifest.xml* directly and injecting name attribute into <application> with unknown id. Android system will ignore this attribute because the id is unknown. But when typical analysis tools (e.g., *Apktool*) repackage it, this name will be included and is not able to be correctly parsed. Packers can utilize the difference to prevent itself from repackaging.
- **Fake encryption** [1]: setting the encryption flag in ZIP file header though the file is actually not encrypted. Old version of Android(<=4.2) ignored this flag but decompression modules of APK static analysis tools often check it, which leads to an error.
- **Method hiding** [6]: modifying the *method_idx_diff* and *code_offset* of certain *encoded_method* in DEX file and pointing to another method. It would make the method invisible to most APK static analysis tools.
- **Illegal opcodes** [23]: injecting illegal opcodes or corrupted object in DEX file to break static analysis tools.
- **Anti decompilation** [23]: adding some non-existing classes to break decompiler and prevent them from converting Dalvik bytecode to JAVA.
- **Magic number tampering**: erasing or modifying the magic number of DEX files. It increases the difficulty of locating the DEX file in memory.

Notice that metadata modification measures are actually tricky defenses that do not really hide the bytecode information. Therefore, it is feasible to circumvent these defenses to acquire bytecode with refined static analysis. In addition, with the verification of DEX format becoming stricter, these tricky defenses are not available anymore.

To thoroughly hide bytecode and thwart static analysis, packers employ DEX encryption techniques. Similar to classic code packers on commodity desktop computer platforms, a DEX encryption scheme generally relies on a decrypting stub responsible for decrypting encrypted bytecode at runtime. Packers would place the decrypting stub in native code part of a protected app as an initializer. The encrypted bytecode is first decrypted by the decrypting stub in packer's native code, and then the Dalvik VM will load and execute the decrypted bytecode.

There are generally two types of code releasing strategies for DEX encryption schemes. The first strategy performs a full-code releasing, which decrypts the entire encrypted DEX file before the control flow reaches to it, and does not modify the released DEX file after the transition from unpacking routine to bytecode. The second strategy performs an incremental code releasing, which selectively decrypts only the portion of code that is actually executed, and may encrypt it again after the execution. This strategy is used as a mechanism to prevent memory dump based unpacking. Traditionally, one specific packer generally adopts only one code release strategy (e.g., full-code releasing adopted by *Bangcle* and *APKProtect*). Latest packers, however, start to adopt both kinds of encryption schemes to strength their protections. For instance, the *Baidu* packer will first release a decrypted DEX, which does not contain the original bytecode however. It contains a second decrypting stub responsible for decrypting original bytecode of a method once it is invoked. That is, the packer employs a two-layer encryption based code protection.

The decrypting stub of DEX encryption schemes could be implemented in either Java or native code. DEX file level encryption schemes in Java usually leverage the *DexClassloader* method or the *openDexFile* interface to fulfil a dynamic code loading based DEX releasing. The decrypting stub executes before the DEX file is loaded and releases the decrypted DEX file for the Dalvik VM. However, this kind of code releasing is easily monitored if analysts could hook certain interfaces of Android system services. Thus many schemes prefer the native code, which is more difficult to be analyzed, to fulfil a specific Dalvik bytecode hiding via DEX file encryption. Those schemes tend to encrypt the code at the method level and use native code to directly manipulate memory instead of invoking certain system APIs.

3 AppSpear

3.1 Overview

The target of AppSpear is to fulfil an automated unpacking process against most common Android packers. The involved issues of this process include anti-analysis defense circumvention, DEX decrypting, and executable rebuilding. The most difficult part of this process is how to overcome the deployed DEX hiding

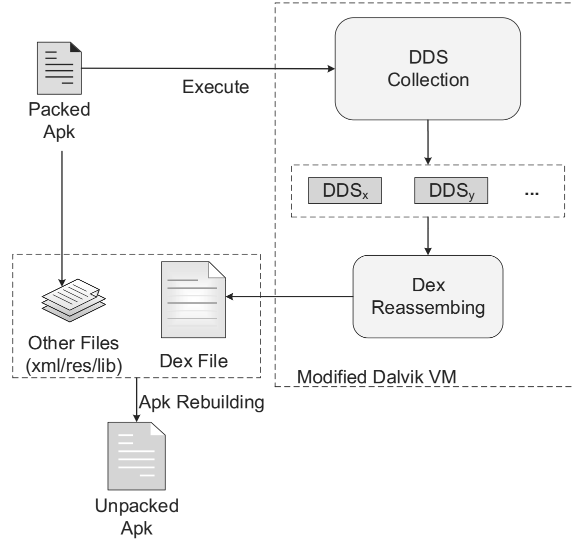

Fig. 1. An overview of AppSpear's unpacking process

techniques. Generally, most Android packers leverage the hybrid code execution style of Android app and implement bytecode decrypting stubs in native code, which are also heavily-packed and obfuscated, thus, difficult to be analyzed and comprehended. Current effective unpacking approaches require a manual reverse engineering to recover the decryption algorithm, and then develop corresponding tools to decrypt the packed bytecode. This process is time-consuming and is easily countered by the packer if it changes its encryption algorithm. To address this challenge, AppSpear adopts a universal Android code unpacking method that does not need to know the detail of the code encryption algorithm. The core intuition of our work is to make use of runtime Dalvik Data Structs (DDS) in memory to reassemble a normal DEX file. When an Android app is installed and executed, its APK file is first decompressed and the belonging DEX file is parsed into different structs of the Dalvik VM instance. The DEX file is a highly structured bytecode data file. Dalvik VM parses the DEX file to initialize the *DexFile* struct and then initializes a series of DDS in memory. These DDS are essential elements of app execution and thus are not allowed to be hidden or arbitrary tampered, otherwise the app will crash. Many packers intentionally modify the mapped DEX data in memory after the DDS initialization to prevent a memory dump based unpacking, but those DDS must be kept accurate to guarantee the stability of the execution. Hence, AppSpear collects those runtime DDS in memory to reassemble the decrypted DEX file.

The feasibility of our work is based on two observations: (1) the packer's functionality is implemented in an independent part of the app (e.g., as a dy-

namic library), and is responsible for initializing the app by releasing the original DEX bytecode before it is loaded by the Dalvik VM. For most Android packer, there exists a clear boundary between these two parts and a transition process from the packer's code to the original code. This is because the hybrid execution model of Android app restricts the arbitrary control flow transition between DEX execution and native code execution. Generally an app would fulfil the transition only through certain system services. Thus we can detect this boundary by monitoring certain JNI interfaces and determining when to start the DDS collection. (2) No matter how complex the packer encrypts the original data, it seldom modifies the semantic of the original bytecode. After the DEX loading process, it is expected to observe accurate content of the bytecode of the original app from the DDS.

Figure 1 illustrates the overview of AppSpear's unpacking process. In detail, AppSpear employs the unpacking through three main steps: First, to circumvent various anti-analysis measures of Android packers. AppSpear introspects the Dalvik VM to transparently monitor the execution of any packed app. Second, AppSpear collects DDS in memory and performs a reassembling process on the collected DDS with some modified methods fixed to re-generate a DEX file, Finally, AppSpear resects anti-analysis code and further synthesizes the DEX file with the manifest file and other resource files from the original packed APK as an unpacked APK. After those three steps, this unpacked app is expected to be analyzed by most regular Android app analyzing tools.

3.2 Transparent Monitoring

Android packers generally adopt complex anti-analysis measures to detect debuggers, emulators and static analysis tools. To effectively circumvent these anti-analysis measures, AppSpear adopts a transparent Dalvik VM instrumentation based bytecode monitoring and retrieving. AppSpear monitors the execution of the app at the Dalvik VM layer, thus is transparent to any bytecode level detection. It is also a very transparent code monitoring to native code level detection because our monitoring is a compilation time instrumentation code injection rather than runtime instrumentation code injection. A runtime instrumentation code injection heavily relies on system provided interfaces (e.g., ptrace) to perform the monitoring, and is easily detected by packers. Compared with them, AppSpear integrates its monitoring code with the Dalvik VM's interpreter and is thus very difficult to be aware of.

AppSpear performs a fine-grained bytecode level instrumentation. We modify the fast interpreter of Dalvik VM to insert an instrumenting stub in each instruction's interpreting handler. Our implementation inserts a function call stub at the very beginning of every opcode's interpretation code. This brings a flexible monitoring that guarantees AppSpear could start unpacking at an arbitrary point of the execution.

To evade typical emulator detecting of packers, AppSpear is deployed on a standard Android device, Google's Nexus phone, instead of an emulator. This guarantees a very trustworthy analyzing environment: if the malware or the packer refuses to execute on this device, then it is not compatible with most

other Android devices. The deployment of AppSpear is simple. It only modifies the Dalvik VM's library (/system/lib/libdvm.so) in system, and is compatible to many mainstream Android devices.

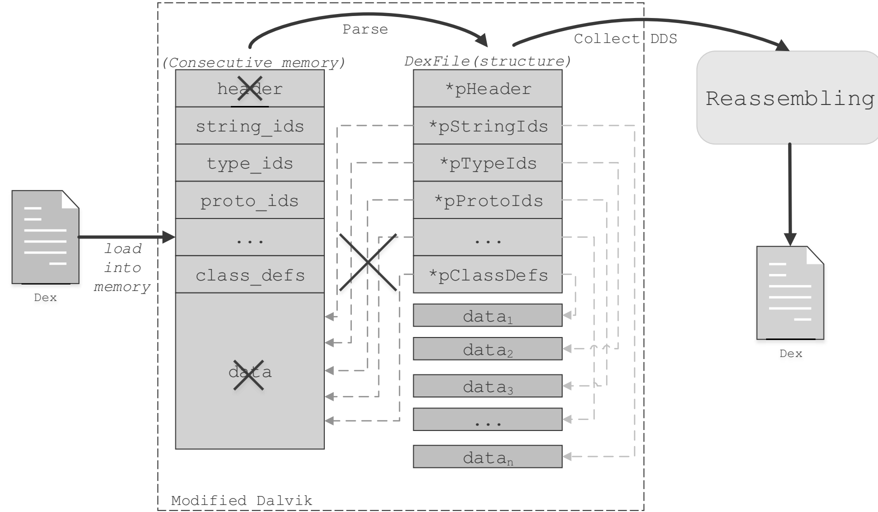

Fig. 2. DEX reassembling

3.3 Dex Reassembling

DEX reassembling of AppSpear is a reverse process of the DEX loading procedure. A Dalvik Data Struct (DDS) is a crucial data structure for the execution of the Dalvik VM. A basic fact for Android app's execution is that the runtime DDS in memory contain the actual execution code and data information of an app. AppSpear leverages this to employ the DEX reassembling process. Dalvik VM maintains 18 DDS parsed from a DEX file during runtime. Those DDS can be classified into two types in our definition: The first type is the index DDS (IDDS) including *Header*, *StringId*, *TypeId*, *ProtoId*, *FieldId*, *MethodId*, *ClassDef* and *MapList*. The main functionality of IDDS is to index the real offset of the second type of DDS: CDDS, which refers to the content DDS (CDDS) including *TypeList*, *ClassData*, *Code*, *StringData*, *DebugInfo*, *EncodedArray* and four items related to *Annotation*. This type of DDS mainly store raw data of bytecode content information. Since *Annotation* relevant DDS are seldom related to program's functionality and thus are less important for program analysis, AppSpear currently ignores these parts of items in the process of reassembling. We leave it for future work.

As Figure 2 shows, in normal DEX loading process, DEX is mapped in consecutive memory. IDDS in initialized *DexFile* struct point to CDDS in the mapped

data space. However, packers may modify raw DEX file or data in memory to produce some malformed data structures and lead to an inaccurate analysis. For instance, packers may modify some metadata in DEX file header and set incorrect offset value of certain CDDS. Some packers even re-map different CDDS to new separated memory space and modify the offset value in IDDS to point to the new addresses. Therefore, AppSpear needs to collect DDS in memory and rewrite a new DEX file other than just dumping the mapped DEX in memory to complete the whole unpacking process.

Then we describe the two phases in detail: DDS collection and DEX rewriting.

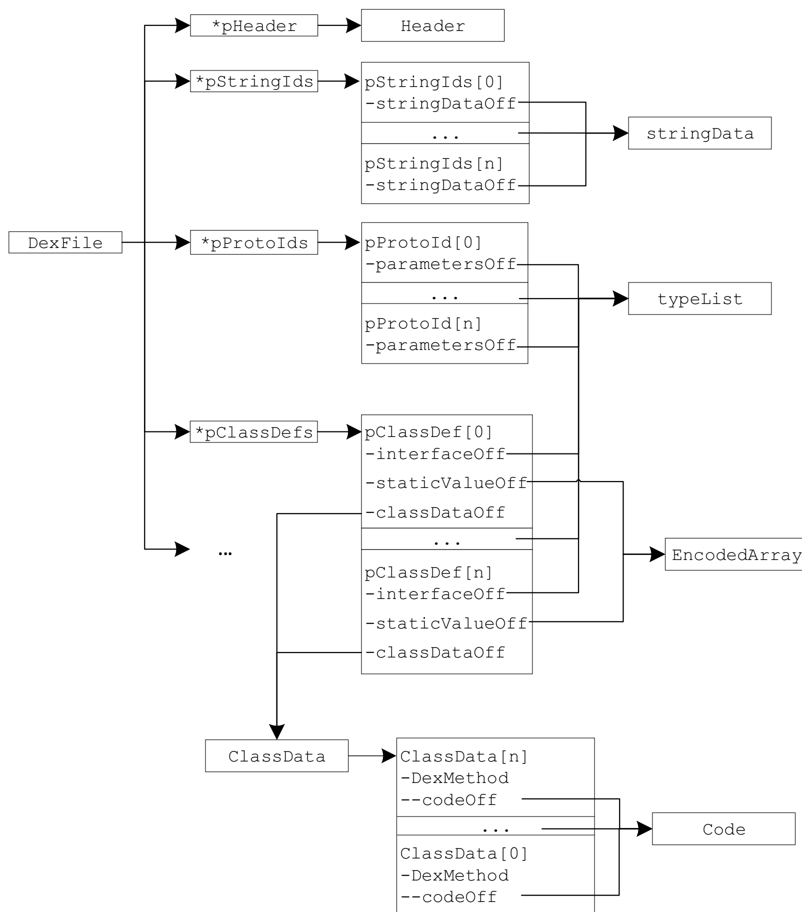

Fig. 3. DDS collection

3.3.1 DDS collection AppSpear collects necessary DDS information to help rebuild a normal decrypted DEX file. However, after the DEX loading process a set of information in the original DEX file is either lost or intentionally modified by packer. The main difficulty involves how to precisely acquire the data content of DDS. AppSpear evades these obstructions by reusing the Dalvik VM's parsing methods (e.g., *dexGetXXX* methods in *DexFile.h* [4]), which always provide accurate results.

To collect DDS, AppSpear first introspects the Dalvik VM instance to access the *DexFile* struct through *Method->clazz->pDvmDex->pDexFile* when instructions are being interpreted. Figure 3 depicts the DDS collection process. AppSpear starts the DDS collection from locating the *DexFile* struct and then accessing certain IDDS including *pStringIds*, *pTypeIds*, *pProtoIds*, *pFieldIds*, *pMethodIds*, *pClassDefs* in *DexFile* struct. These IDDS are fixed size structs thus their contents are directly read. Notice that the *DexHeader* struct also contains pointers of CDDS, but AppSpear avoids accessing them directly because of the potential modification of packers. Specifically, AppSpear traverses all attributes of IDDS to collect accurate offset of CDDS including *StringData*, *TypeList*, *ClassData*, *EncodedArray*, *Code*, etc. After determining the offset, AppSpear further accesses the *size* attribute of each DDS to determine the length.

3.3.2 DEX Rewriting After acquiring the size and offset of DDS in memory, the next step is to determine how to place each DDS into a re-created DEX file. According to the order of *map item type codes* defined in *DexFile.h*, AppSpear re-orders the collected DDS and writes them back to the DEX file in order.

During the rewriting an important issue for AppSpear is the offset adjustment. A DDS in memory maintains many pointers that point to other DDS and the contents of these pointers are reloaded values that represent the offset at runtime. When this DDS is written back into DEX file, AppSpear should adjust this offset value to a new one that represents the actual offset in DEX file. AppSpear checks every pointer of DDS to adjust this offset value when performing DEX rewriting. Because the entire *MapList* struct stores offset and size of other DDS, AppSpear re-calculates all metadata in *MapList* during rewriting process. In addition, in case that the packer's modification of certain value, AppSpear directly uses known knowledge to fill them in DEX file (e.g., size of header, magic number of header) instead of reading them from raw DDS in memory.

What's more, during the DEX rewriting, AppSpear 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, AppSpear fills the gap with null byte if the size of the DDS is not 4-byte aligned. Second, in DEX file the size attribute of *ClassData* 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 *ClassData* struct 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 *ClassData*. AppSpear would automatically calculate these differences to generate a rewritten DEX file.

3.3.3 Multiple Unpacking AppSpear needs to collect DDS at certain point of execution (denoted as unpacking point) to guarantee the effectiveness of DEX reassembling. The instruction-level instrumentation of AppSpear proves that it could choose arbitrary point to perform collecting, which is significant for fighting against self-modified packers.

The default unpacking point of AppSpear is determined by an APK's manifest file. 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 Dalvik VM's interpretation meets the main activity, AppSpear starts the collection.

A particular difficult point is that an app may load multiple encrypted Dalvik executables at runtime. As a result, our unpacking should also employ DDS collection at each point when a new DEX file is loaded. AppSpear introspects the execution of Dalvik VM and monitors the context. When certain context (e.g., *DexClassloader* is invoked by the app or a new *DexFile* struct is met) is encountered, a DDS collection procedure is triggered. In this way, AppSpear guarantees that any runtime loaded DEX file could also be captured.

One issue for our dynamic analysis based unpacking is that if an encrypted method is not executed during runtime, it would not be able to be decrypted and reassembled into the re-generated app. To handle this, AppSpear traces executed instructions for multiple times, trying to trigger the hidden methods as much as possible. After the tracing phase, AppSpear performs an offline comparison between methods in DEX file and methods in traces. If one method in DEX file does not correspond to that in traces, it will be repaired using the accurate result in traces. Although dynamic analysis will meet the incompleteness issue, AppSpear tries to approach a practically acceptable result. Moreover, if a malicious method is not executed in our device, it is not expected to be triggered in real world devices.

3.4 APK Rebuilding

Many Android app analysis tools require a complete APK file instead of a sole DEX file to perform analysis. Moreover, in our reassembled DEX file there still exists a small amount of anti-analysis code injected by packers to obstruct analysis. AppSpear performs a last step APK rebuilding to solve these issues.

3.4.1 Anti-analysis Code Resecting Packers usually leverage bugs of some analysis tools to inject code stubs that obstruct the normal analysis. AppSpear resects those code stubs to help analysis. Because those code stubs are very specific and aim at certain analysis tools, they usually have obvious features and are easily detected. AppSpear maintains an empirical database of this kind of code and automatically resects any code stubs in database when encountering.

3.4.2 APK Repackaging AppSpear combines the reassembled DEX file with materials from the existing packed app including *manifests.xml* and resource files to repackage the app. The manifests 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 the Dalvik VM. AppSpear would fix this entry point hijacking with the original entry point of the DEX file.

4 Experimental Evaluation

To evaluate the effectiveness of AppSpear, we test malware samples packed by 10 mainstream commercial Android packers, which cover the latest and most complex Android packing techniques. To illustrate and evaluate the effectiveness of our approach on malware detection, 31 packed malware are manually chosen from the collected 490 packed samples of SandDroid to test AppSpear. These 31 samples can run without crashes or exceptions before unpacking and are all of different package names. In other words, we avoid to choose the packed malware from the same original app. The chosen packed malware set covers 6 packers (*Bangcle*, *Ijiami*, *Qihoo360*, *Naga*, *NetQin*, and *APKProtect*). Notice that latest online Android packing services claim that they do not provide malicious code packing service and there exist no such packed samples detected, we also want to ensure if their countermeasures do take effect and there are no such potential packed malicious apps. So we develop a home brewed malicious app that requires many permissions and collects sensitive data. The test app contains all four main components (*Activity*, *Service*, *Broadcast Receiver*, *Content Provider*) and an *Application* class. We submit this test app to 7 online packing services of *Bangcle* (a.k.a *Secneo*), *Ijiami*, *Qihoo360*, *Baidu*, *Alibaba*, *LIAPP* and *Dex-Protector*, (*NetQin* and *APKProtect* which appear in malware are not available since the first quarter of 2015) and actually get different packed versions. In a word, we believe that all those 10 packers (7 in existing malware samples and 3 extra online packing services) could help protect malicious apps.

We execute the packed samples on our devices implemented with AppSpear. In our experimental evaluation, AppSpear is deployed on two devices, Galaxy Nexus and Nexus 4 respectively, and the versions of Android operating system are 4.3 and 4.4.2. We build a modified Dalvik VM (in the form of libdvm.so) based on the AOSP source code and replace the default Dalvik VM with our AppSpear integrated one. Notice that our deployment leverages a third-party *Recovery* subsystem (e.g., *CWM* Recovery) to fulfil the system lib replacement and does not require a privilege escalated Android (a.k.a rooted Android), which may fail to pass the integrity checking of some packers.

In our experiment, AppSpear successfully circumvents all anti-debugging and integrity checking measures of these packers, and all of the packed samples on our devices execute stably without occurring exceptions or crashes. Using the default setting, AppSpear conducts the unpacking as soon as the *Main Activity* class invokes the *onCreate* method. Almost all of the samples are unpacked automatically and the corresponding unpacked APK files are generated. As a contrast, existing methodology such as memory dumping either fails and breaks on the halfway due to the various anti-analysis techniques or gets the broken DEX files that cannot be parsed correctly by other analysis tools and need further fix.

4.1 Accuracy of DEX reassembling

We first evaluate the accuracy and feasibility of the newly generated DEX files. We choose 5 popular static tools to validate the reassemled DEX files. They are DEXTemplate for 010Editor, Baksmali, Enjarify, IDA Pro and AndroGuard. The reason why we choose these five tools is that they are all widely-used and can parse a DEX file to extract information from it. They have their own parsing engines and have no dependency with each other. We consider the failure of DEX parsing as the following conditions:

DEXTemplate for 010Editor is a DEX file parsing template. It will raise errors if the format of a DEX file is invalid. Baksmali is a widely-used disassembler for DEX files. When disassembling invalid DEX files, it will throw exceptions. IDA Pro also supports DEX file disassembling. If it prompts windows indicating parsing error or can not identify the files as DEX when opening the reassembled DEX files, then we regard this condition as DEX parsing failure. Enjarify, which is provided by Google, is a tool for translating Dalvik bytecode to equivalent Java bytecode, aiming to replace dex2jar. When the translation process of a DEX file ends, Enjarify will give the result such as how many classes are translated successfully and how many fail. As long as one class in the whole DEX file fails to be translated, we regard it as an parsing failure. The DEX parsing engine of Androguard is implemented by Python and remains active in open source community. We regard the DEX parsing failure of AndroGuard occurs once it raises errors or exceptions while it is being used to do further static analysis (such as sensitive API extraction in DEX).

The testing set consists of 7 home brewed samples submitted to online packers and 31 malware samples from the collected 490 packed samples, which covers 10 different packers altogether.

DEXTemplate	Baksmali	Enjarify	IDA Pro	AndroGuard
38/38	37/38	34/38	38/38	38/38

Table 2. success rate of parsing reassembled DEX

The result in Table 2 shows that DEXTemplate for 010Editor, IDA Pro and AndroGuard successfully parse all reassembled DEX files. However, Baksmali fails to parse only one sample and raises exceptions. The reason is that some illegal instructions, which cause the parsing failure, are intentionally inserted into 9 of 350 classes in that sample. But the exceptions in those 9 classes do not affect the parsing of other 341 classes. Four samples fail when Enjarify tries to translate them to JAR file. Among the 4 samples, 1% classes on average in each sample appear the parsing errors. After manually checking the reason, we find that these failure result from the limitations presented by Enjarify itself on its homepage. The result proves that the success rate of parsing the reassembled DEX files is high and those few failure cases are mainly caused by the implementation problem of the static tools themselves.

4.2 Unpacking Code Packed Malware

Since AppSpear's target is to rebuild a packed malware into its normal form so that program analysis tools or automated malware detection systems are able to analyze its real behaviors, we implement an in-depth static sensitive behavior analysis tool based on AndroGuard to further evaluate the unpacking results. The tool extracts the sensitive permissions of an APK and counts the number of sensitive behaviors (our tool simply regard the sensitive API calls as sensitive program behaviors) related to those permissions (referring to the map of API and permissions in AndroGuard [2]) before and after unpacking. Since packers do not change the permissions declared in the manifest file, the number of permissions used by samples remains still.

Packer Name	Malware Sample	Permissions	Packed	Unpacked
Qihoo360	CE8668B81420CF6843DA4D2EB846C314	6	5	52
	9EC616C1BC4470EE03C4E299C3A616D6	16	5	76
	878CF954DAE814D83BFFC5374E8BF423	12	5	60
Bangcle	6D3D891FC3459CA2A9911D8438966B20	8	0	84
	3FF42BF94C39A9E4B2F0EA50747670B4	6	0	30
	3F6487D723F60B4C80AC7EAB7F22BBCC	13	0	332
	03CA02466849847A26A926D6605927D0	15	0	174
	1D44FA56473B5EC27E75C734062102CA	8	0	13
	020D37EE843411AB749CADF17FC43006	15	0	108
	5F5D6F391148A4E3ACDFF3C57B8EA6EA	10	0	100
	2E06E5A5350EF54342D1328DF216D261	7	0	213
	1AF5B2D290902EE0124239F4315F4B40	9	0	91
	4DA607EDB8D7689B604C775670E5DA6F	6	0	81
	5B5674C8BA87CBC163328B27EFF24392	11	0	230
	07D9EB10587722E26BA93CB47D598641	7	0	65
	3AC41F02613FE1436564AD1C30226416	16	2	6
	9A10E7A615589B0E949F9FF9CBDFE50E	12	0	176
	4E478E2BE20EAC9C0B939FA6ADA60CE5	6	0	30
	0FA57B3D98C24EABB32C47CA3C47D38A	6	102	89
	4B9762D0B4F00E6F1A42D4AA6E984301	21	4	109
NetQin	2C3EB7833619F35A54C91166BAAE5FCD	11	5	11
	C63AE255C1F3A22DAC47E8BFB400615B	5	5	6
	2A7CADAB7FC61508C70B146B496BDA12	17	5	41
Naga	91815F6F381DB7CA793885873AFFA782	7	0	25
Ijiami	FFB08850111C1D8B061792953588CB88	21	0	109
	A1B22DE648076B8B9515F77326D9DB13	19	0	80
APKProtect	A58DB782081C0A41BE7556FD662F9F09	23	21	30
	67EC17C3B482AC5C1E896A2BB2C64353	5	4	13
	3094CF944D45E48201A8E8EC4C742CD0	2	1	14
	FC272EA7F6A5FE21ED4EADAD8EF34155	9	43	45
	7BB4FB90B8C37311DC6C35AAA15F58C6	2	1	1

Table 3. Sensitive behaviors before and after AppSpear's unpacking

AppSpear conducts the unpacking work on 31 packed malware and the details of our unpacking and analysis results are shown in Table 3. The third column of Table 3 indicates the number of sensitive permissions extracted by our static analysis tool. The fourth column and fifth column indicate the number of sensitive behaviors our static analysis tool counts before and after the AppSpear's unpacking respectively. As Table 3 shows, before the unpacking packers can hide almost all sensitive behaviors of malware which can evade the detection of static

analysis tools. After the unpacking of AppSpear, the number of detected sensitive behaviors in unpacked malware increases significantly. This proves that an effective unpacking process is crucial to current malware detection.

Several noticeable observations are also revealed by our experiment: First, the samples packed by *APKProtect* generally possess higher number of sensitive behaviors compared with other samples. After a manual analysis, we find that the packed code of *APKProtect* is not entirely encrypted. Instead, the DEX file is only partially encrypted by *APKProtect*. As a result, some sensitive behaviors are not hidden by the packer before the execution and our static tool can detect them. However, after the unpacking of AppSpear, more hidden sensitive behaviors appear. Second, a malware sample packed by *Bangcle* packer (MD5 sum: `0FA57B3D98C24EABB32C47CA3C47D38A`) presents an unusually high number of sensitive behaviors before unpacking. After manually checking the packed sample, we find that it consists of several third-party libraries which contain those sensitive behaviors. Because the packer only packs the main DEX file of the malware, the 102 sensitive behaviors refers to those in third-party libraries. After the unpacking of AppSpear, another 89 sensitive behaviors in the unpacked DEX file are detected by our tool, which indicate the real malicious behaviors of this malware sample.

4.3 Home brewed Samples

Considering the fact that some newly-born packers do not appear in the above malware samples and also even some packers appear, they improve their code packing strength all along, we submit our home brewed sample to 7 latest online packing services to further evaluate the effectiveness of AppSpear. Different from the malware in wild, we have the original DEX file of our home brewed sample before packing, so we can manually verify those unpacked apps by comparing their decrypted and reassembled DEX file with the original one. The content of DEX file in each unpacked app contains the exact components and classes of the original app, which demonstrates the effectiveness of our approach.

To thoroughly understand the detailed advanced packing techniques and prove that AppSpear does defeat the anti-analyses defenses used by those packers, we conduct an in-depth study on verifying the results manually. We find that due to its design principle, AppSpear unpacks the protected app in a unified way without considering certain anti-analyses defenses. For instance, some packers hook system calls (e.g, *write()*) in their own process space to prevent the DEX data from being dumped. If the source address of *write()* is located in the memory scope of the mapped raw DEX data, the content will be modified by the hooking function. AppSpear evades memory dump measures through reassembling and generating a new DEX file instead of reading the raw DEX data, and the unpacking results are accurate. In addition, we manually check each packer's DEX hiding schemes to validate the correctness of our unpacking strategy. We find that the DEX hiding schemes of *Qihoo360*, *LIAPP* and *Bangcle* adopt a full-code unpacking style. Even the encryption algorithms of their DEX hiding schemes are unknown to AppSpear, our DEX reassembling approach easily col-

lects relevant DDS and recovers exactly the same DEX as the original one.

Ijiami. This packer modifies the DEX header to erase the magic number of the DEX file. The measure is used to thwart memory dump based unpacking method, because it is difficult to automatically locate the target DEX data through the memory space of the process without the help of the magic number. Since AppSpear focuses on the *DexFile* struct rather than the raw DEX data in memory, it is not affected by this counter-measure. What's more, *Ijiami* modifies the attribute of *headerSize* in DEX header to a larger value, which crashes some static analysis tools when parsing the unpacked DEX. However, AppSpear has already considered this modification and always uses the correct value to rewrite the reassembled DEX file in the DEX rewriting process.

Alibaba. This packer also applies modifications to the original DEX file. The reassembled DEX file from the sample packed by *Alibaba* packer contains more classes than that in the original DEX file. Some of these injected classes will cause the failure of some static analysis tools such as *dex2jar*. AppSpear considers these classes as anti-analysis code and resects them directly. The DEX hiding scheme also re-maps the *DexCode* of every *DexMethod*, and modifies the *codeOff* attribute in *DexCode* struct to a negative value, AppSpear ignores those modifications and directly acquires data of every DDS to reassemble a new DEX file, thus the reassembled DEX file is accurate.

Baidu. This packer adopts an incremental packing style. It erases the DEX header and inserts native methods as wrapper of some specific 'target' methods (e.g, *onCreate* of *MainActivity*). The code of these methods are patched as `NOP` until they are executed. When executed, the wrapping native method before the patched method will be first executed to recover the actual bytecode. After the invoking, the bytecode is immediately erased by the wrapping native method after the patched method. AppSpear deals with this situation by adopting instruction level tracing and gets the real bytecode of the target method, and repairs those patched methods using the traced information.

Dexprotector. This packer splits the original DEX file to several DEX files and packs them. AppSpear can monitor dynamic DEX file loading in one process and recover multiple packed DEX files. Besides the packing part of the packer, the recovered DEX files also indicate that *Dexprotector* applies heavy code obfuscation to the original DEX file. AppSpear focuses on hidden code unpacking instead of code de-obfuscation. Source code obfuscation may increase the comprehending complexity for reverse engineering, but it seldom affects the malware detection because the obfuscation is not able to hide privilege API invoking, which directly exposes the malicious intention. Although AppSpear is not able to de-obfuscate this DEX file (de-obfuscation is out of our work's scope), the multiple unpacked DEX files still contain all bytecode information of the original app and are adequate to a later program analysis or malware detection.

5 Discussion

AppSpear is based on dynamic analysis, which means it would suffer from the code coverage issue. If a method is not invoked during runtime, packers would not decrypt the bytecode of this method, then AppSpear can not recover this part of code. Fortunately, AppSpear is deployed on real devices and tries to trigger the hidden methods as much as possible, which can mitigate this shortcoming. From the other side, it is less meaningful for packers to hide a malicious method that is seldom invoked during runtime. From the results of our experiment, we find out that most hidden methods locate on entry point classes or can be easily triggered.

Malware can employ various anti-analysis techniques for emulator or VM evasion [18]. It is feasible that packers can use similar ways to detect AppSpear and then hide the decrypting procedure to defeat our unpacking approach. They can utilize some code features or fingerprint of AppSpear to avoid being analyzed by AppSpear. To thwart such evasion, AppSpear can also use similar anti-detection measures as emulator evading detection proposed in [14].

Besides commercial Android packers, there also exist some home brew packers and some malware may use them to protect the code. Although the methodology of our proposed approach is universal on monitoring and unpacking most kinds of DEX encryption schemes, due to the lack of sample for testing, we can not guarantee that AppSpear could handle those home brew packers perfectly. We leave this as future work. Particularly, some advanced packers transform bytecode into obfuscated native code executables on Android. AppSpear can not de-obfuscate native code obfuscation. Packers may even pack the native code in original APKs even though this kind of code packing technique is still not prevalent on Android. This is one of the limitations of our work and it is possible to be addressed in future with advanced de-obfuscation strategy.

6 Related Work

The topic of code packing have been thoroughly studied in the literature, and several solutions have been proposed for code unpacking [13]. Pedrero et al. [26] present a very comprehensive study on commodity runtime code packers. Their work studied the runtime complexity of packers and evaluated on both off-the-shelf packers and custom packed binaries on desktop computer systems. On the commodity desktop system, a series of automatic unpacking approaches and tools have been proposed. **Polyunpack** [20] performs automatic unpacking by by emulating the execution of the program and monitoring all memory writes and instruction fetches, and considers all instructions fetched from previously written memory locations to be successfully unpacked. **Omniunpack** [16] is a real-time unpacker that performs unpacking by looking for written-then-execute pattern. **Renovo**[15] also uses the written-then-execute pattern to perform the unpacking. It instruments the execution of the binary in an emulator and traces the execution at instruction-level. **Pandoras Bochs** [9] is also an emulator based automatic malware unpacking tool, which uses the full system emulator bochs as its engine. **Eureka** [22] uses coarse-grained NTDLL system call monitoring

for automated malware unpacking, is only available for Windows packers. These unpacking approaches and tools 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 Dalvik bytecode, which means a packer should consider both aspects and keep the balance between protection strength and stability. Meanwhile, the analysis tools (e.g., emulators or code instrumentation tools) on Android platform are less powerful. To the best of our knowledge, our work is the first one to study Android packers systematically, and can handle every available commercial packer.

Before our work, there was a range of summarization work that introduces the feature and anti-analysis technique of certain Android packers. Strazzere introduced anti static analysis and anti dynamic analysis code protection techniques in [23] and [24] separately. Detecting emulator is also an important anti-analysis measure of many packers and is thoroughly discussed in [27]. However, only a few work discuss the bytecode encryption issue in a generic perspective. As far as we know, current bytecode decryption techniques or tools either directly copy DEX data in memory, which is not feasible for unpacking state-of-the-art Android packers, or rely on encryption algorithm reverse engineering, which involves substantial manual efforts. The main shortcoming of many unpacking approaches is that they heavily rely on the specific memory dump based methodology. For instance, Park [17] leverages *wait-for-debug* feature of Android platform to circumvent anti-debugging and then performs a memory dump based unpacking. Yu [28] and Strazzere [25] make some assumptions of the packer's features and leverage these features to locate bytecode, which are already unavailable due to the evolution of the packer. DexHunter [29] mainly focuses on how to locate and dump the DEX in memory. Our proposed DEX reassembling technique settles this deficiency and leads to a more universal unpacking.

The ART runtime has been introduced since Android 4.4.2 to support a more efficient app execution. Although AppSpear is based on the Dalvik VM of Android and focuses on DEX reassembling, which means it is not compatible for those Android versions without Dalvik VM, all apps can also be executed on Dalvik VM even though new ART runtime is supported because of the backward compatibility requirement, So our approach is still effective for a expected long period.

Android malware detection is an active research area and many methods [10] [30] [12] [7] [19] have been proposed to address the large-scale malware analysis issue. However, seldom work considers the code packing issue. Our work is a solution to enhance current malware analysis. The unpacked APK from AppSpear can help program analysis tools, especially those static analysis tools, perform a more accurate malware detection.

7 Conclusion

This paper describes a systematic study of code packed Android malware. Commercial Android packers are analyzed and relevant anti-analysis techniques are summarized. An investigation of 37,688 Android malware samples is then conducted and 490 code packed apps are analyzed with the help of our proposed

AppSpear, an automated code unpacking system. AppSpear employs a novel bytecode decrypting and DEX reassembling approach to replace traditional manual analysis and memory dump based unpacking. Experiments demonstrate that our proposed AppSpear system is able to unpack most malware samples protected by popular commercial Android packers, and it is expected to become an essential supplementary process of current Android malware detection.

Acknowledgments

We would like to thank our shepherd, Elias Athanasopoulos, and the anonymous reviewers for their insightful comments that greatly helped improve the manuscript of this paper.

References

1. An APK Fake Encryption Sample. `https://github.com/blueboxsecurity/DalvikBytecodeTampering`.
2. API_permissions.py in AndroGuard. `https://github.com/androguard/androguard/blob/master/androguard/core/bytecodes/api_permissions.py`.
3. AVL Malware Report 2014. `http://blog.avlyun.com/2015/02/2137/malware-report/`.
4. libdex/DexFile.h - platform/dalvik - Git at Google. `https://android.googlesource.com/platform/dalvik/+/android-4.4.2_r2/libdex/DexFile.h`.
5. SandDroid - An automatic Android application analysis system. `http://sanddroid.xjtu.edu.cn/`.
6. Axelle Apvrille. Playing Hide and Seek with Dalvik Executables. *Hacktivity*, 2013.
7. Daniel Arp, Michael Spreitzenbarth, Malte Hübner, Hugo Gascon, Konrad Rieck, and CERT Siemens. Drebin: Effective and explainable detection of android malware in your pocket. In *Proc. of Network and Distributed System Security Symposium (NDSS),21st*, 2014.
8. Leyla Bilge, Andrea Lanzi, and Davide Balzarotti. Thwarting real-time dynamic unpacking. In *Proc. of European Workshop on System Security,4th*, 2011.
9. Lutz Böhne. *Pandoras bochs: Automatic unpacking of malware*. PhD thesis, University of Mannheim, 2008.
10. Iker Burguera, Urko Zurutuza, and Simin Nadjm-Tehrani. Crowdroid: behavior-based malware detection system for android. In *Proc. of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, 2011.
11. Jonathan Crussell, Clint Gibler, and Hao Chen. Andarwin: Scalable semantics-based detection of similar android applications. In *Proc. of the 18th European Symposium on Research in Computer Security (ESORICS)*, 2013.
12. Michael Grace, Yajin Zhou, Qiang Zhang, Shihong Zou, and Xuxian Jiang. Riskranker: scalable and accurate zero-day android malware detection. In *Proc. of the 10th international conference on Mobile systems, applications, and services*, 2012.
13. Fanglu Guo, Peter Ferrie, and Tzi-Cker Chiueh. A study of the packer problem and its solutions. In *Proc. of the 11th Recent Advances in Intrusion Detection (RAID)*, 2008.
14. Wenjun Hu. Guess Where I am: Detection and Prevention of Emulator Evading on Android. *HITCON*, 2014.
15. Min Gyung Kang, Pongsin Poosankam, and Heng Yin. Renovo: A hidden code extractor for packed executables. In *Proc. of the 5th ACM workshop on Recurring malcode*, 2007.

16. Lorenzo Martignoni, Mihai Christodorescu, and Somesh Jha. Omniunpack: Fast, generic, and safe unpacking of malware. In *Proc. of the 23rd Computer Security Applications Conference*, 2007.
17. Yeongung Park. We Can Still Crack You! General Unpacking Method for Android Packer (not root). *Black Hat Asia*, 2015.
18. Thanasis Petsas, Giannis Voyatzis, Elias Athanasopoulos, Michalis Polychronakis, and Sotiris Ioannidis. Rage against the virtual machine: hindering dynamic analysis of android malware. In *Proc. of the 7th European Workshop on System Security (EuroSec)*, 2014.
19. Siegfried Rasthofer, Steven Arzt, Marc Miltenberger, and Eric Bodden. Harvesting runtime data in android applications for identifying malware and enhancing code analysis. *Technical report*, 2015.
20. Paul Royal, Mitch Halpin, David Dagon, Robert Edmonds, and Wenke Lee. Polyunpack: Automating the hidden-code extraction of unpack-executing malware. In *Proc. of the 22nd Computer Security Applications Conference*, 2006.
21. Patrick Schulz and Felix Matenaar. Android Reverse Engineering and Defenses. `http://bluebox.com/wp-content/uploads/2013/05/AndroidREnDefenses201305.pdf`.
22. Monirul Sharif, Vinod Yegneswaran, Hassen Saidi, Phillip Porras, and Wenke Lee. Eureka: A framework for enabling static malware analysis. In *Proc. of the 13th European Symposium on Research in Computer Security (ESORICS)*, 2008.
23. Tim Strazzere. Dex education: Practicing safe dex. *Black Hat, USA*, 2012.
24. Tim Strazzere. Dex education 201: anti-emulation. *HITCON*, 2013.
25. Tim Strazzere and Jon Sawyer. ANDROID HACKER PROTECTION LEVEL 0. *DEF CON 22*, 2014.
26. Xabier Ugarte-Pedrero, Davide Balzarotti, Igor Santos, and Pablo G Bringas. SoK: Deep Packer Inspection: A Longitudinal Study of the Complexity of Run-Time Packers. In *Proc. of IEEE Symposium on Security and Privacy,36th*, 2015.
27. Timothy Vidas and Nicolas Christin. Evading android runtime analysis via sandbox detection. In *Proc. of ACM symposium on Information, computer and communications security,9th*, 2014.
28. Rowland Yu. Android Packers: Facing the challenges, Building solutions. In *Proc. of the 24th Virus Bulletin International Conference*, 2014.
29. Yueqian Zhang, Xiapu Luo, and Haoyang Yin. Dexhunter: Toward extracting hidden code from packed android applications. In *Proc. ESORICS*, 2015.
30. Yajin Zhou, Zhi Wang, Wu Zhou, and Xuxian Jiang. Hey, You, Get Off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets. In *Proc. of the 19th Network and Distributed System Security Symposium (NDSS)*, 2012.