2017 IEEE/ACM 39th International Conference on Software Engineering

# Adaptive Unpacking of Android Apps

Lei Xue*, Xiapu Luo*§, Le Yu*, Shuai Wang*, Dinghao Wu†
*Department of Computing, The Hong Kong Polytechnic University
†College of Information Sciences and Technology, The Pennsylvania State University
{cslxue,csxluo,cslyu,csswang}@comp.polyu.edu.hk, dwu@ist.psu.edu

***Abstract*—More and more app developers use the packing services (or packers) to prevent attackers from reverse engineering and modifying the executable (or Dex files) of their apps. At the same time, malware authors also use the packers to hide the malicious component and evade the signature-based detection. Although there are a few recent studies on unpacking Android apps, it has been shown that the evolving packers can easily circumvent them because they are not adaptive to the changes of packers. In this paper, we propose a novel adaptive approach and develop a new system, named *PackerGrind*, to unpack Android apps. We also evaluate *PackerGrind* with real packed apps, and the results show that *PackerGrind* can successfully reveal the packers' protection mechanisms and recover the Dex files with low overhead, showing that our approach can effectively handle the evolution of packers.**

## I. Introduction

With more than 2 million apps on the Google Play, Android has accounted for around 85% of all smartphone sales to end users [1]. At the same time, recent reports revealed that 97-99% of mobile malware runs on Android [2], [3], most of which are repackaged apps [4]–[7]. They are legitimate apps carrying malicious components injected by attackers. One root cause is the lack of binary protections, which is one of theOWASP mobile top ten risks [8], so that attackers can easily reverse engineer the apps and tamper their code.

To protect apps from being tampered and reverse engineered, a number of app packing services (or packers) emerge [9], which conceal and obfuscate the real code (i.e., Dex files) and prevent others from obtaining them [10], [11]. Unfortunately, attackers also utilize packers to hide malware for evading the signature-based detection and impeding the investigation of their malicious behaviors [12]. A recent report from Symantec reveals that the number of packed Android malware has increased from 10% to 25% [13]. Therefore, researchers proposed a few unpacking approaches recently to recover the Dex files from packed apps in order to facilitate the analysis of mobile malware [10], [11].

However, the arms race between packers and unpacking tools never ends. The latest version of packers could easily evade those unpacking tools. The key issue lies in the one-pass processing strategy adopted by the unpacking tools. In other words, they are not adaptive to the evolution of packers. In this paper, we propose a new adaptive approach, which employs an iterative process, to recover the Dex files from packed apps, and develop a new system named *PackerGrind* to automate most steps in the process.

§ The corresponding author.

Our iterative process consists of three major tasks including (1) *monitoring*, which captures how packed apps work, especially how it prepares the real code for execution, and then generates tracking reports, based on which we can determine the data collection points; (2) *recovery*, which collects the pieces of data in Dex files at the selected data collection points and reconstructs Dex files; (3) *analysis*, which determines whether new data collection points are needed to recover Dex files. Automating this process is non-trivial because we need to address two challenging research questions:

**RQ1:** *How to conduct cross-layer profiling of packed apps' behaviors in a smartphone?*

**RQ2:** *How to effectively recover the Dex files of apps packed by different packers?*

Answering RQ1 needs a system that can perform cross-layer monitoring of an app's behaviors and run in real smartphones. Note that with the support of Android framework, apps run in the runtime, which was the Dalvik Virtual Machine (DVM) before Android 5.0 and became the new Android runtime (ART) afterwards, and the runtime is on top of the modified Linux. Packed apps usually exploit the features of the Java language, Android framework, and native libs/instructions to hide the real code, detect emulator, and prohibit debugging [14]. Existing dynamic analysis systems for monitoring apps cannot address RQ1, because they either rely on emulator (e.g., QEMU) [15], [16] and debugging techniques [17], [18] or lack of the support of cross-layer profiling [16], [19]. To tackle RQ1, we propose and develop a novel cross-layer monitoring component for *PackerGrind*. By exploiting dynamic binary translation [20], it collects information from the runtime, the system, and the instruction layers and runs in smartphones. Moreover, it supports both DVM and ART.

Existing unpackers for Android apps cannot fully address RQ2 because of their one-pass processing strategy. To approach RQ2, we first identify basic Dex data collection points by scrutinizing how DVM and ART load and run apps. Since different packers employ various protection methods to modify the code and data in memory dynamically, *PackerGrind* provides detailed tracking reports as well as suggested criteria to recognize the protection patterns. Moreover, *PackerGrind* conducts static analysis on the Dex file obtained at each run to facilitate users to identify new data collection points if needed. Although this step might need manual inspection, the detailed information and scripts provided by *PackerGrind* could alleviate the workload. *PackerGrind* also has built-in rules to automatically unpack apps protected by existing packers

1558-1225/17 $31.00 © 2017 IEEE
DOI 10.1109/ICSE.2017.40

accessible to us. After that, *PackerGrind* will re-run the packed app, collect Dex data at selected collection points, and finally reconstruct the Dex file.

In summary, our major contributions include:

- We propose a new iterative process to unpack Android apps. This process as well as the new system, *PackerGrind*, is adaptive to the evolution of packers.
- We design *PackerGrind* that automates most steps in the iterative process. It can conduct cross-layer monitoring and Dex file recovering in real smartphones. Moreover, it supports both DVM and ART. To our best knowledge, it is the *first* system that can address the above two challenging research questions simultaneously.
- We implement *PackerGrind* with 21.3K lines of C/C++ code (not include Valgrind) and 2.5K lines of Python code, and compare it with the state-of-the-art unpacking tools with real apps packed by popular packers. The results show that *PackerGrind* can unpack all these apps with low overhead whereas DexHunter [10] recovered a few and Android-unpacker [21] unpacked none.

The rest of the paper is organized as follows. Section II introduces background knowledge and a motivating example. Section III describes the basic Dex data collection points. Section IV details the design and implementation of *PackerGrind* and Section V reports the experimental results. After discussing the limitations of *PackerGrind* and future work in Section VI, we introduce the related work and conclude the paper in Section VII and Section VIII, respectively.

## II. Background

### A. Dex File

The bytecode of an Android app is contained in the Dex file which is a highly structured data file consisting of different Dex data items [22](e.g., `proto_id_item`, `code_data_item`). A Dex file has three major sections including *header* section, *data identifiers* section, and *data* section. The *header* section includes a summary of the Dex file (e.g., checksum, size, and offsets). The *data identifiers* section contains 6 identification lists for defined classes, namely, `string_ids`, `type_ids`, `proto_ids`, `field_ids`, `method_ids`, and `class_defs`, each of which contains multiple items. For example, `string_id_item` contains the offset from the start of the Dex file to the corresponding `string_data_item`. The *data* section contains the information related to bytecode, including `map_list`, `type_list`, `class_data_item`s, `code_data_item`s, `debug_info_item`s, `encoded_array_item`s, and four annotation data items.

### B. Android App Packing

Packers usually protect apps' code from three aspects, namely, hiding Dex files, impeding the dumping of Dex files in memory, and hindering the reverse-engineering of Dex files.

**Hiding Dex files.** Packers often use three approaches to hide Dex files. *(1) Dex file modification.* Packed apps use native code to modify Dex files in the memory when the app is running. For example, apps packed by Baidu packer in 2015 fill a special method with valid instructions just before the method is called and erase them after execution. *PackerGrind* can capture such behaviors and dump the correct instructions at the right moment. *(2) Dynamic class loading.* Packers put the bytecode of selected functions in separated Dex files, and load them when the functions are invoked. They even encrypt the Dex files and decrypt them before loading the required classes. *PackerGrind* can dump the Dex files after they are loaded because it traces the runtime's functions. *(3) Native method.* Packers could turn the selected Dex functions into native methods and then invoke them through Java native interface (JNI) from the Dex file. Although *PackerGrind* is not designed to reverse engineer the native code for regenerating the bytecode, it can still provide useful information about the native methods thanks to its cross-layer monitoring component.

**Impeding the dumping of Dex files.** Packers usually employ three approaches to prevent unpackers from dumping the real code in memory. *(1) Emulator detection.* Since many dynamic analysis systems rely on Android emulator, packers employ advanced techniques [14] to determine whether a packed app is running in an emulator. If so, the app will exit. *PackerGrind* does *not* rely on the emulator. Instead, it exploits dynamic binary translation [20] to perform cross-layer monitoring and recovers Dex files. *(2) Anti-debug.* If an unpacker attaches to the packed apps as a debugger, it can monitor the apps and obtain the Dex files. To impede such method, the packed apps often launch multiple threads and let one thread attach to another using *ptrace*, because a process can only be attached by one process. *PackerGrind* does *not* use this approach. *(3) Hooking.* To prevent unpackers from accessing and dumping the Dex files in memory, packed apps often hook the functions related to file and memory operations to prohibit unpackers from using them. *PackerGrind* can disable these hooks.

**Anti reverse-engineering of Dex files.** Packers commonly employ various techniques (e.g., obfuscation [23], etc.) to raise the bar for understanding the internal logics through static code analysis. Handling them is out of the scope of *PackerGrind*.

### C. Motivating Example

Existing unpackers are not adaptive to the changes of packers, and hence can be easily circumvented. In particular, they usually perform one-pass processing based on the developer's knowledge for obtaining the Dex files. Therefore, packers can modify their behaviors accordingly to defeat such unpackers. We argue that unpackers should be adaptive to the changes of packers by monitoring and learn their behaviors.

We use an app packed by Baidu packer (in DB-15) [24] as a motivating example. As show in Fig. 1, the original code of *onCreate()* in MainActivity is replaced by those at Line 23-26, and *onCreate001()* is empty and called between two JNI methods (i.e., *A.d()* and *A.e()*). By monitoring the packed app, we find that when *A.d()* is invoked, it fills *onCreate001()* with correct instructions, which will be erased after *A.e()* is called.

The state-of-the-art unpackers (i.e, DexHunter [10] and AppSpear [11]) cannot obtain the instructions in *onCreate001()* effectively. DexHunter collects the Dex data in *dvmDefineClass()*. However, when this function is called, the correct instructions have not been written to *onCreate001()* yet, and thus DexHunter misses them. Similarly, AppSpear assumes that DVM's parsing methods (e.g., *dexGetCode()*) always provide expected results. However, in this example, *dexGetCode()* can only obtain the right instructions of *onCreate001()* when it is invoked between *A.d()* and *A.e()*. In other words, AppSpear cannot get the correct results if it misses the right moment. Since DexHunter and AppSpear are implemented within the runtime, they cannot monitor packed apps' behaviors at the system and the instruction levels to determine the right moments.

```
22 public void onCreate(Bundle bundle) {
23   String str = "LXXX;->onCreate001(Landroid/os/Bundle;)V";
24   A.d(str);
25   onCreate001(bundle);
26   A.e(str);
27 }
28 private void onCreate001(Bundle savedInstanceState) {
29 }
```

Fig. 1: *onCreate()* method of the app packed by the Baidu packer of DB-15.

*PackerGrind* can address this issue because it iteratively monitors packed apps at different layers, facilitates the determination of collection points, and recovers the Dex files. By analyzing the Dex file obtained in the first run, we can learn that the instructions of *onCreate001()* are modified during execution. According to the tracking report, we know that *A.d()* is called before *onCreate001()* to fill the instructions and *A.e()* is invoked after *onCreate001()* to erase them. Moreover, as shown in Fig. 1, the parameters of both method *A.d()* and *A.e()* are the name of *onCreate001()*. With such information, we add a new collection point between *A.d()* and *A.e()*, and then *PackerGrind* can reconstruct the correct Dex file automatically.

## III. Basic Dex Data Collection Points

As shown in Fig. 2, we divide the process from Dex files loading to method execution into four phrases, namely, parsing Dex files, loading classes, resolving methods, and executing methods. Consequently, we define four basic collection points for DVM and ART, respectively.

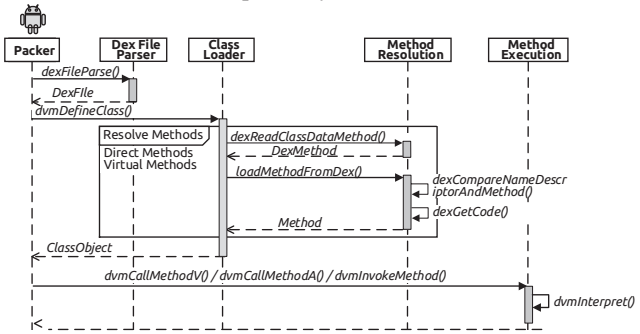


Fig. 2: The process from Dex file loading to method execution.

### A. Dalvik VM (DVM)

**Parsing Dex Files.** A Dex file can be loaded either from a file in storage through *openDexFileNative()* or a memory space through *openDexFile_bytearray()*. Both methods will call *dexFileParse()* to parse the Dex file and return the structure `DexFile` to represent this Dex file in runtime, as shown in Fig. 2. Since `DexFile` is initialized according to the Dex file header in *dexFileParse()*, we select *dexFileParse()* as the first Dex data collection point.

**Loading Classes.** A class can be loaded through *Dalvik_dalvik_system_DexFile_defineClassNative()*. In this function, *dvmDefineClass()* is called to load the class and return the structure `ClassObject` that contains the class's information (e.g., fields, methods, etc.). Moreover, the structure `class_def_item` is read from the Dex file, and then the structure `class_data_item` is parsed from the Dex file according to its offset in `class_def_item`. After that, `ClassObject` is initialized. Hence, we choose *dvmDefineClass()* as the second Dex data collection point.

**Resolving Methods.** When loading a class, the class loader will resolve each method to initialize `ClassOject` according to `class_data_item`. During such resolution, the class loader first obtains `DexMethod` from the Dex file by calling *dexReadClassDataMethod()*. Then, it creates a structure `Method` according to `DexMethod` in *LoadMethodFromDex()*. During the initialization of `Method`, *dexCompareNameDexcriptorAndMethod()* is called to check whether it is a *finalize* method, and then *dexGetCode()* is invoked to fetch the code information from the Dex file to populate `Method`. Since the symbols of inline functions and static functions are not exported in `libdvm.so`, we let *dexCompareNameDexcriptorAndMethod()* instead of *dexGetCode()* be the third Dex data collection point.

**Executing Methods.** Native code can invoke Java methods through Java reflection or JNI reflection using functions like *dvmInvokeMethod()*, *dvmCallMethodA()*, and *dvmCallMethodV()*. Since they call *dvmInterpret()* for both fast-interpreter and portable-interpreter, we select it as the fourth Dex data collection point.

### B. Android Runtime (ART)

During the installation of an app, ART invokes the tool `dex2oat` to compile the Dex file to the oat file, which is in ELF format but contains both the Dalvik bytecode and the compiled code. If an app without oat file is being launched, ART performs the same action. ART can execute a method in the interpreter mode, which is similar to DVM, or the compiled code mode. By default, if a method has compiled code, ART runs its compiled code. Otherwise, ART interprets its Dalvik bytecode. If a packed app uses `dex2oat` to compile Dex files containing real code into oat file, *PackerGrind* obtains the Dex file according to the arguments passed to `dex2oat`.

For the methods executed in the interpreter mode, *PackerGrind* also has four basic Dex data collection points.

**Parsing Dex Files.** Similar to DVM, `DexFile` represents the Dex file in runtime, which contains the information of classes and methods. The class constructor of `DexFile` (i.e., *DexFile()*) will read the Dex file in memory and parse it similar to *dexFileParse()* in DVM. Therefore, we choose *DexFile()* as the first Dex data collection point.

**Loading Classes.** *DefineClass()* of class `ClassLinker` is used to load and parse each class in the Dex file and return an instance of class `Class` to represent the class in runtime. Hence, we select *DefileClass()* as the second Dex data collection point.

**Resolving Methods.** ART uses `ArtMethod` to represent each method of a class, and the instance of `ArtMethod` for a method is initialized in *LoadMethod()*. Therefore, we let it be the third Dex data collection point.

**Executing Methods.** *Invoke()* of the class `ArtMthod` in ART is invoked when a Java method is called by Java reflection and JNI reflection. Hence, we select *Invoke()* as the fourth Dex data collection point.

## IV. PACKERGRIND

### A. Overview

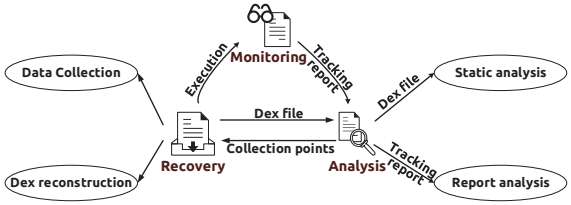


Fig. 3: The iterative process realized by *PackerGrind*.

To be adaptive to the evolution of packers, *PackerGrind* adopts the iterative process shown in Fig. 3 to recover Dex files. This process consists of three tasks in each run. More precisely, when running a packed app in smartphone, *PackerGrind* monitors its behaviors from three layers including runtime, system, and instruction, and generates a tracking report. At the same time, *PackerGrind* collects Dex data at specified collection points and reconstructs Dex files by the end of each run. Then, it performs static analysis on the recovered Dex files. Users determine whether new collection points are needed according to the tracking report and the result of static analysis, because the basic data collection points described in Section III may not be enough for *PackerGrind* to collect all data of the original Dex file. We propose basic protection patterns in Section IV-E to help users determine additional collection points if needed. Based on these patterns, we have identified all collection points for packers accessible to us as described in Section V. After adding the new collection points, *PackerGrind* will run the process one more time and repeat this procedure until the Dex file is correctly recovered.

Fig. 4 shows the architecture of *PackerGrind*. It consists of three components for finishing the three tasks in the iterative process. The monitoring component (Section IV-B) tracks the behavior of packed apps at three layers and generates the tracking report. The recovery component (Section IV-D) automatically gathers Dex data at selected collection points and reconstructs the Dex file. The analysis component (Section IV-E) performs static analysis on the Dex file dumped at each run and determines whether new data collection points are needed. We develop *PackerGrind* based on Valgrind [20] and therefore it runs in real smartphone instead of emulator.

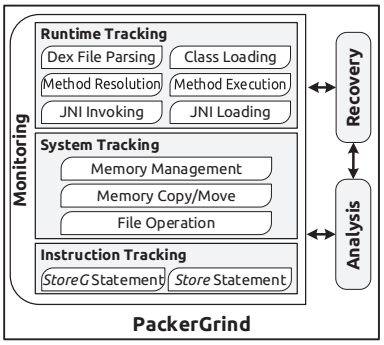


Fig. 4: Architecture of *PackerGrind*.

### B. Monitoring

*1) Runtime layer:* To locate the structure `DexFile`, which represents a Dex file in runtime, and collect Dex data from the four basic data collection points (Section III), *PackerGrind* monitors the arguments and the returns of the selected functions in Table I using the function wrapping technique [20]. For example, by wrapping *dexFileParse()* with the wrapper function *dexFileParse_wrapper()*, we can obtain the arguments passed to *dexFileParse()* and its return.

TABLE I: Wrappers for tracking Dex and DVM related events.

| Category | Wrapped Functions | Tracked Information |
|---|---|---|
| Dex Data | *dexFileParse()* | Dex file parsing |
| | *dvmClassDefine()* | Class loading |
| | *dexCompareNameDescriptorAndMethod()* | Method resolution |
| | *dvmInterpret(), dvmMterpStdRun()* | Method execution |
| | *dvmCallJNIMethod()* | JNI invocation |
| | *dvmInvokeMethod()* | Java reflection |
| | *dvmCallMethodV(), dvmCallMethodA()* | JNI reflection |
| Native Module | *dvmLoadNativeCode()* | Native code loading |

Table I lists two set of functions. One includes the functions related to the basic Dex data collection points and the function *dvmCallJNIMethod()*, because some packers use native code to modify Java methods through JNI. Moreover, it contains the functions related to Java reflection and JNI reflection (i.e., *dvmInvokeMethod()*, *dvmCallMethodV()*, *dvmCallMethodA()*), because they are used by some packers to invoke Java methods. The other set has *dvmLoadNativeCode()* because it will be called when *System.load()* or *System.loadLibrary()* is used to load native module. Since native modules allow packed apps to release or modify the Dex data, we wrap *dvmLoadNativeCode()* to track such behaviors.

*2) System layer:* Packed apps can release and modify the Dex data in memory by calling system library functions and system calls through its native module. Since such behaviors cannot be monitored at the runtime layer, *PackerGrind* tracks them at the system layer by wrapping memory management functions (e.g., allocation, free, mapping, etc.), file operations (e.g., open, read, write, and close), and data movement functions (e.g., *memcpy()*, *strcpy()*, etc.). It also traces the invocation of some system calls (e.g., *sys_map()*, *sys_unmap()* and *sys_protect()*), because they can be used by packed apps to allocate memory, release memory, and change memory access permissions, respectively. *PackerGrind* maintains a surveillance memory list for the ranges of memory that may be used to store the

Dex data, and records the operations on them in tracking report.

*PackerGrind* also wraps some functions for special purposes. For example, some packers adopt timeout mechanism for anti-unpacking (e.g., Ijiami). More precisely, if the unpacking process takes a time longer than the packer's timeout threshold, the app crashes. To address this issue, we wrap the system call *sys_gettimeofday()* to modify the timestamps returned to the packer so that its timeout mechanism will not be activated. Users can use *PackerGrind* to track more functions if necessary.

*3) Instruction layer: PackerGrind* instruments *store* instructions to monitor operations for modifying Dex files, because packed apps can write and modify Dex data in memory directly through its native code instead of invoking memory copy or move functions. *PackerGrind* skips system libraries, because no system library functions except memory copy and move functions, which are wrapped at the system layer, will modify Dex files. *PackerGrind* maintains a system library memory list for the memory regions of system libraries, and uses it to determine whether an instruction belongs to system libraries.

To monitor memory modifications, *PackerGrind* inserts an intermediate representation (IR) of function invocation statement before `Ist_StoreG` and `Ist_Store` statements that are translated from packed apps' native code by Valgrind [20]. In this IR statement, the instruction tracking function will be called to check whether the target address is in the surveillance memory list. If so, *PackerGrind* records the target address, operand value, and instruction address in the tracking report.

### C. Tracking report

A tracking report contains three major types of information and its length depends on the app's execution time. **(1) Dex file.** When a new Dex file represented by `DexFile` is found, *PackerGrind* parses `DexFile` and records the memory information about the Dex file (e.g., Dex file header, classes, methods and codes). **(2) Memory modification.** *PackerGrind* maintains a Dex file list containing the memory ranges of all Dex files in the runtime. When functions and instructions for memory modification are identified, *PackerGrind* checks whether the target addresses are in the memory range of a Dex file. If so, the modification information is written to the tracking report. At the system layer, this information includes the invoked function, target address, the Dex structure to which the target address belongs (e.g., Dex header field), and the value written to the target address. At the instruction layer, this information includes instruction address, instruction types (i.e., `Ist_StoreG` and `Ist_Store`), target address, target address information, and the stored value. **(3) Method invocation.** At the runtime layer and the system layer, the invocation and the return of any wrapped function are logged into the tracking report with the parameters and the return values.

### D. Recovery

It collects the Dex data and reconstructs Dex files.

*1) Dex data collection:* In each run, *PackerGrind* starts collecting Dex data after a `DexFile` is identified because it represents a Dex file. Once the Dex file is located through `DexFile`, *PackerGrind* initializes a shadow memory for storing the collected Dex data items belonging to this Dex file, and then copies the data items to the shadow memory. When a new Dex data item is collected, *PackerGrind* firstly checks whether the shadow memory for this data item exists. If so, *PackerGrind* copies this data item to the shadow memory. Otherwise, *PackerGrind* creates a new shadow memory for this data item, copies it to the shadow memory, and changes the corresponding offset to this item in the shadow memory.

*2) Dex file assembling:* After collecting Dex data, *PackerGrind* assembles them into a Dex file. Since a packer can release Dex data in discontinuous memory areas, there will be more than one shadow memory allocated for storing the collected Dex data. Therefore, *PackerGrind* allocates a continuous memory and assembles the collected Dex data together to reconstruct Dex files. Specifically, *PackerGrind* performs a two-step Dex file construction. First, it divides the collected Dex data items into different groups according to their types. For example, *PackerGrind* groups all `class_def_item`s together to assemble `class_defs`. After that, these groups of data will be put together according to the Dex format. By doing so, *PackerGrind* can obtain the offsets of the Dex data items and the sizes needed for such data structures.

Second, *PackerGrind* allocates a continuous memory region and copies the collected data to it starting from their group offsets. For each data structure, *PackerGrind* updates its members according to the offsets of the data structures. For example, when a `class_data_item` is copied into the continuous memory region, we will update the corresponding `class_def_item.class_data_off`. *PackerGrind* will recalculate the meta-data of Dex header after all data structures are copied into the continuous memory region to ensure the validity of Dex file. Eventually, *PackerGrind* dumps the memory region and outputs the Dex file.

### E. Analysis

We analyze the dumped Dex file and the tracking report to achieve three purposes. First, since packed apps usually use JNI methods (i.e., native code) to dynamically modify Dex files in memory and the dumped Dex file may contain unexplored paths to JNI methods, we conduct static bytecode analysis to look for such paths. Second, we inspect the tracking report to determine whether new data collection points are needed. Third, we identify more information about the discovered JNI methods from the result of cross-layer monitoring.

*1) Static bytecode analysis:* We employ IntelliDroid [25] to determine how to trigger the JNI methods in the dumped Dex files through statical analysis. Given an app and a set of targeted JNI methods, IntelliDroid can help us find the execution paths leading to these methods as well as the corresponding input. Thus, we first extract JNI methods from the Dex files and let them be the target methods, and then use IntelliDroid to look for the execution paths leading to them with event handlers as the entry-points. After that, we drive the app to execute the target JNI methods following the corresponding paths.

```
001 Syscall: open() /data/dalvik-cache/data@app@demo.killerud.gestures-1.apk@classes.dex flag=0x00020000 fd=50          // First dex file is opened and tracking starts
002 .........
003 Invoke: dvmLoadNativeCode() /data/app-lib/demo.killerud.gestures-1/libmobisec.so                                  // Load the native library named libmobisec.so
004 Return: dvmLoadNativeCode() /data/app-lib/demo.killerud.gestures-1/libmobisec.so
005 Invoke: dvmCallJNIMethod() pDexFile=0x05f3ef40 mth: Lcom/ali/mobisecenhance/StubApplication; attachBaseContextIT(VL)    // JNI method attachBaseContextIT(VL) is invoked
006         Syscall: open() 0x61c6faf0(/data/data/demo.killerud.gestures/files/libmobisecx1.so) flag=0x00020042 fd=54      // File named libmobisecx1.so is opened and file handler is 54
007         Syscall: mmap() off_0x00000000 -> 0x375b6000-0x375b7a58 flat=rw- prot=0x2 fd=54                             // File lbmobisec1.so is mapped to memory range 0x375b6000-0x375b7a58
008         Invoke:dexFileParse() file: 0x375b6000-0x375b7a58 flag: kDexParseDefault pDexFile=0x00000000               // dexFileParse() is invoked to parse memory range 0x375b6000-0x375b7a58
009         Return:dexFileParse() file: 0x375b6000-0x375b7a58 flag: kDexParseDefault pDexFile=0x05f44970               // dexFileParse() returns results: pDexFile=0x05f44970
010         Syscall: close() fd=54                                                                                     // File lbmobisec1.so is closed
011         3759DC87: Executable | STORE *(a_0x375b654c) <- v_0x24 | interfacesOff (pDexFile=0x05f44970 ClassIdx=5)   // The interfaceOff (ClassIdx=5) value is modified by native code
012         3759E1F3: Executable | STORE *(a_0x375b6b2a) <- v_0x80 | class_data_item (pDexFile=0x05f44970 ClassIdx=0)  // The code_data_item of the class (ClassIdx=0) is modified by native code
013         3759E1F9: Executable | STORE *(a_0x375b6b2b) <- v_0x80 | class_data_item (pDexFile=0x05f44970 ClassIdx=0)  // The code_data_item of the class (ClassIdx=0) is modified by native code
014         .......
015         3759E205: Executable | STORE *(a_0x375b6bc7) <- v_0x80 | class_data_item (pDexFile=0x05f44970 ClassIdx=9)  // The code_data_item of the class (ClassIdx=9) is modified by native code
016         3759E209: Executable | STORE *(a_0x375b6bc8) <- v_0x00 | class_data_item (pDexFile=0x05f44970 ClassIdx=9)  // The code_data_item of the class (ClassIdx=9) is modified by native code
017 Return: dvmCallJNIMethod() pDexFile=0x05f3ef40 mth: Lcom/ali/mobisecenhance/StubApplication; attachBaseContextIT(VL)   // JNI method attachBaseContextIT(VL) returns
018 Invoke: dvmDefineClass() pDexFile=0x05f44970 class: Lhiof/enigma/android/gestures/GesturesDemoActivity;              // dvmDefineClass() is invoked to define class GesturesDemoActivity;
019 Return: dvmDefineClass() pDexFile=0x05f44970 class: Lhiof/enigma/android/gestures/GesturesDemoActivity;   // dvmDefineClass() returns.
```

Fig. 5: Tracking report for an app packed by the Ali packer.

*2) Tracking report analysis:* We provide Python scripts to analyze the tracking report in order to recognize the protection patterns and determine whether new collection points are needed. By exploiting the insight that a portion $P$ (e.g., Dex header, methods, etc.) of a Dex file should be valid right before it is being used, we define four basic protection patterns for $P$: (1) it is changed to valid value before its first use (**FmT**); (2) it is modified to invalid value after its last use (**TmF**); (3) it is altered to valid value before being used and turned to invalid after the use (**FmTmF**); (4) it is always valid (**T**). Although the basic protection patterns are by no means comprehensive, they cover all packed samples accessible to us. Users can define new patterns after studying the tracking report.

To recognize the protection patterns, we first collect the method invocation and the memory modification information from the tracking report. According to the target address information of each modification operation $M$, we identify which portion of the Dex file is modified by $M$. By checking the method invocation information, we determine when the portion is used. Given each portion $P$, we regard its content is valid when it is being used. A quick approach to detect invalid portions is to apply static analysis tool to the dumped Dex file and see whether there is any parsing error.

During the first run, *PackerGrind* collects Dex data at the basic data collection points, and hence the dumped Dex file may include invalid portion. If so, we infer the protection pattern and select new data collection points (i.e., when its content is valid). After that, we execute *PackerGrind* again to collect more valid portions.

We use an app packed by Ali packer as an example to illustrate this process. Fig. 5 shows the tracking report. At line 003, *dvmLoadNativeCode()* is called to load the native library `libmobisec.so`. Then, the JNI method *attachBaseContextIT(VL)* of class *Lcom/ali/mobisecenhance/StubApplication* is called by *dvmCallJNIMethod()* (line 005), and it returns at line 017.

The information about the instruction layer modifications is from line 011 to line 016. At line 011, *"3759DC87"* and *"Executable"* are the instruction address and the executable permission of the address, respectively. *"STORE"* indicates the instruction type, which is `Ist_Store` at line 011. *"a_0x375b654c"* and *"v_0x24"* are the target address and the stored value, respectively. *"interfacesOff (pDexFile=0x05f44970 ClassIdx=5)"* denotes that the target address is the field *interfacesOff* of the 5th class

```
1 public void onCreate(Bundle savedInstanceState) {
2   super.onCreate(savedInstanceState);
3   setContentView(C0000R.layout.main);
4   this.display = ((WindowManager) getSystemService("window")).getDefaultDisplay();
5   this.mLibrary = GestureLibraries.fromRawResource(this, C0000R.raw.gestures);
6   if (!this.mLibrary.load()) {
7     finish();
8   }
9   findViewById(C0000R.id.gestures)).addOnGesturePerformedListener(this);
10 }
```

(a) The original *onCreate()*.

```
1 public void onCreate(Bundle savedInstanceState) {
2   A.V(0, this, new Object[]{savedInstanceState});
3 }
```

(b) The *onCreate() in an packed app.*

```
001 Invoke:dvmCallJNIMethod() pDexFile=0x05f41a90 mth: Lcom/baidu/protect/A; V(VILL)
002         JNI_Reflection: Landroid/app/Activity; onCreate(VL)
003         JNI_Reflection: Landroid/app/Activity; setContentView(VI)
004         JNI_Reflection: Landroid/app/Activity; getSystemService(LL)
005         JNI_Reflection: Landroid/view/WindowManagerImpl; getDefaultDisplay(L)
006         JNI_Reflection: Landroid/gesture/GestureLibraries; fromRawResource(LLI)
007         JNI_Reflection: Landroid/gesture/GestureLibraries$ResourceGestureLibrary; load(Z)
008         JNI_Reflection: Landroid/app/Activity; findViewById(LI)
009         JNI_Reflection: Landroid/gesture/GestureOverlayView; addOnGesturePerformedListener(VL)
010 Return:dvmCallJNIMethod() pDexFile=0x05f41a90 mth: Lcom/baidu/protect/A; V(VILL)
```

(c) Tracking report of *A.V()*.

Fig. 6: The method *onCreate()* before and after packing and the tracking report of *A.V()*.

in the Dex file, which is represented by a `Dexfile` in memory address *0x05f44970*. At line 11, the field *interfaceOff* of the 5th classes (*ClassIdx=5*) in the Dex file is modified. From line 012 to 016, the `class_data_item`s of 10 classes (from *ClassIdx=0* to *ClassIds=9*) in the Dex file are modified by the STORE (i.e., `Ist_Store`) instruction. Finally, the class *GesturesDemoActivity* is defined by *dvmDefineClass()* at line 018 and 019.

The tracking report shows that the `class_data_item`s of all classes are filled with valid values in the JNI method *attachBaseContextIT(VL)* before *dvmDefineClass()*. Since the runtime loads classes in *dvmDefineClass()* based on `class_data_item`s, the `class_data_item`s' contents are valid after calling *attachBaseContextIT(VL)*. That is, `class_data_item` follows the **FmT** protection pattern. Hence, we can collect the Dex data after *attachBaseContextIT(VL)* returns. Since *PackerGrind* collects Dex data when it is defined by *dvmDefineClass()* by default, the collected content is valid and no more run is needed.

*3) Native methods inspection:* A packer can re-implement an app's Java methods in the native module, and then call them through JNI. Although *PackerGrind* is not designed to reverse-engineer the native code for reconstructing the bytecode, it can still provide useful information about the native methods

thanks to its cross-layer monitoring capability. More precisely, packed apps have to use JNI reflection (i.e., *dvmCallMethodV()* or *dvmCallMethodA()*) to invoke Android framework APIs in Java. Since *PackerGrind* has wrapped these functions, it can provide rich information about the native methods.

For example, Fig. 6 shows the method *onCreate()* in the original app and that in the app packed by Baidu. It shows that the method *onCreate()* has been re-implemented in native code, which invokes Android framework APIs through JNI reflection. In other words, after packing, the original implementation of *onCreate()* is replaced with the invocation of the native method *A.V()*. From the tracking report of *A.V()* shown in Fig. 6c, we can infer the original implementation of *onCreate()*. For example, the method *onCreate()* of class *android/app/Activity* is invoked at Line 002. Correspondingly, as shown in Fig. 6a, the method *onCreate()* of the MainActivity's super class (*android/app/Activity*) is invoked at Line 2. Note that other unpackers (e.g., [10], [11]) cannot profile such behavior.

### F. Implementation on ART

We adopt similar methods to monitor packed apps running in ART. Different from DVM that provides only two functions to invoke a Java method in native code, ART provides *Call**TYPE**Method()* functions and *CallStatic**TYPE**Method()* functions to call non static methods and static methods, respectively. ***TYPE*** indicates the type of the method's return value.

To call a native method through JNI, ART invokes the functions *artInterpreterToCompiledCodeBridge()* or *art_quick_generic_jni_trampoline()* depending on whether the native method contains compiled code or not. Both of them eventually establish the JNI calling environment according to the JNI call convention in ART. Before the execution of native code, the functions *JniMethodStart()* or *JniMethodStartSynchronized()* will be called depending on whether the native method is synchronized or not. We wrap these two functions to track the invocation of JNI methods and get the name of JNI methods in ART.

Since ART is implemented in C++, all Dex data structures are stored in class objects. We parse such class objects and recover Dex data structures from them. *PackerGrind* currently supports the ART in Android 6.0 and it reuses the modules at the system layer and instruction layer for DVM.

## V. Evaluation

We conduct extensive experiments to evaluate *PackerGrind* by answering the following five questions.

**Q1:** *Can PackerGrind be adaptive to the evolution of packers and identify their protection mechanisms?*
**Q2:** *Can PackerGrind correctly recover Dex files?*
**Q3:** *Is PackerGrind better than other available unpackers?*
**Q4:** *Can PackerGrind facilitate the analysis of malware?*
**Q5:** *What is the overhead of PackerGrind?*

### A. Data Set

We use two sets of packed apps to evaluate *PackerGrind*. The first set has 480 packed apps with ground truth. More precisely, we download 40 randomly selected open-source apps from F-Droid [26] and then upload them to 6 online commercial packing services (Qihoo [27], Ali [28], Bangcle [29], Tencent [30], Baidu [24], Ijiami [31]) in Mar. 2015 (denoted as DB-15) and Mar. 2016 (denoted as DB-16) to construct 480 packed apps. The second set consist of 200 packed malware samples from Palo Alto Networks [32]. These samples were packed by eleven packers including Ali [28], APKProtect [33], Baidu [24], Bangcle [29], Ijiami [31], Naga [34], Qihoo [27], Tencent [30], LIAPP [35], Netqin [36], and Payegis [37].

We conduct the experiments in both Android 4.4 with DVM and Android 6.0 with ART on a Nexus 5 smartphone [38]. *PackerGrind* monitors the protection patterns of these packed apps and recover their Dex files.

### B. Protection Mechanisms

Using *PackerGrind*, we reveal the protection mechanisms adopted by 6 packers, each of which has two versions for DB-15 and DB-16, individually. As shown in Table II, packers are evolving with new techniques and hence unpackers should be adaptive to the evolution. *PackerGrind* can unpack apps protected by all mechanisms except the *Re-implement Method*.

TABLE II: Protection mechanisms adopted by six packers in DB-15 and DB-16. The symbol before (or after) "—" denotes whether a packer in DB-15 (or DB-16) uses the mechanism or not.

| Packer | Qihoo | Ali | Bangcle | Tencent | Baidu | Ijiami |
|---|---|---|---|---|---|---|
| Dynamically Release Dex | ✓—✓ | ✓—✓ | ✓—✓ | ×—✓ | ✓—✓ | ✓—✓ |
| Dynamically Modify Dex | ×—✓ | ×—✓ | ×—✓ | ×—✓ | ✓—× | ✓—✓ |
| Customized Dex Parsing | ×—✓ | ×—× | ×—× | ×—× | ×—× | ×—✓ |
| Re-implement Method | ×—× | ×—× | ×—× | ×—× | ×—✓ | ×—× |
| Anti-Debug (e.g., *ptrace*) | ×—× | ×—× | ✓—✓ | ×—× | ×—× | ×—✓ |

All but Tencent packer of DB-15 release Dex files to memory dynamically. In DB-16, all packers except Baidu dynamically modify selected structures in the Dex file. For example, Ali packer changes the `class_data_item` of each class in the loaded Dex file from invalid value to valid one before the class is defined. Moreover, Ijiami packer sets the Dex file header with valid value before *dexFileParse()* is called, and changes it to invalid value after using *dexFileParse()*.

Qihoo packer and Ijiami packer of DB-16 use their own functions rather than the standard runtime functions to parse certain structures of the Dex file. Qihoo packer invokes the native code in its library libjiagu.so instead of *dvmDefineClass()* to load classes. Ijiami packer parses the methods of the loaded classes again using the native code in its library libexec.so, and changes the instruction offsets of those methods to valid values right before *dvmDefineClass()* returns. Baidu packer of DB-16 re-implements all *onCreate()* functions using native code with the same functionality. Bangcle packers uses *ptrace()* to protect the app process from being attached by debugging tools while Ijiami packer of DB-16 periodically searches for the string "@com.android.reverse-" to detect ZjDroid [39].

***Answer to Q1:*** *PackerGrind* is adaptive to the evolution of packers and can identify the protection mechanisms adopted by various packers.

### C. Recovering Dex Files

*1) Number of runs required for determining all Dex data collection points:* Table III shows that *PackerGrind* needs one run for Ali, Bangcle, Tencent, and the new version of Baidu packers (i.e., DB-16). It takes two runs to handle Qihoo, Ijiami, and the old version of Baidu packers(i.e., DB-15). Once the Dex data collection points for a packer are identified, *PackerGrind* can recover the Dex files in one run.

TABLE III: Number of runs required for determining all Dex data collection points.

| Packer | Qihoo | Ali | Bangcle | Tencent | Baidu (DB-15) | Baidu (DB-16) | Ijiami |
|---|---|---|---|---|---|---|---|
| Number of runs | 2 | 1 | 1 | 1 | 2 | 1 | 2 |

**Qihoo.** Since Qihoo packer parses the protected Dex file using native code in its library libjiagu.so instead of standard functions, *PackerGrind* locates the first Dex file when *dvmIntepret()* is invoked (i.e., the fourth Dex data collection point). From the tracking report, we notice that the `class_data_off` is changed to zero after its library `libjiagu.so` is loaded. Since *PackerGrind* does not find the Dex file at other collection points, we add a new data collection point right before *dvmLoadNative()* for the second run, and then the correct Dex file is recovered.

```
22 public void onCreate(Bundle bundle) {
23  String str = "LXXX;->onCreate001(Landroid/os/Bundle;)V";
24  A.d(str);
25  onCreate001(bundle);
26  A.e(str);
27 }
28
29 private void onCreate001(Bundle savedInstanceState) {
30  super.onCreate(savedInstanceState);
31  setContentView(C0000R.layout.main);
32  this.display = ((WindowManager)getSystemService("window")).getDefaultDisplay();
33  this.mLibrary = GestureLibraries.fromRawResource(this, C0000R.raw.gestures);
34  if (!this.mLibrary.load()) {
35   finish();
36  }
37 findViewById(C0000R.id.gestures)).addOnGesturePerformedListener(this);
38 }
```

Fig. 7: Content of *onCreate001()* after 2nd run.

**Baidu.** For the samples packed by the old version of Baidu packer (i.e., in DB-2015), we find that the method *onCreate001()*, which is recovered after the first run, is empty. Hence, we add a new data collection point after *A.d()* by analyzing the tracking report and the Dex file recovered in the first run. In the second run, the Dex file is successfully recovered (e.g., Fig. 7).

**Ijiami.** After the first run, we observe that all instructions of the methods in the MainActivity are zero. By analysing the tracking report, we find that the packed apps modify the instructions of `Method`s after method resolution. Moreover, the instructions of *Method*s are different from those of the corresponding `code_item` structure in Dex file. Therefore, we add a new data collection point after *dvmDefineClass()* for the second run, and then the Dex file is successfully recovered.

*2) Correctness of recovered Dex files:* We assess the correctness of recovered Dex files from three aspects. First, we apply five popular static analysis tools, which can reverse-engineer Dex files, to the recovered Dex files, because they adopt different verification strategies to check Dex files. These tools include *Baksmali* [40], *Dexdump* [41], *Dex2jar* [42], *Jadx* [43] and *IDA Pro* [44]. *PackerGrind* can successfully recover the Dex files of almost all samples. The only exception comes from *Dex2Jar* when it handles the recovered Dex files from Tencent samples (from DB-15). It failed to transform the Dalvik bytecodes into java bytecodes due to Dex optimization conducted by DVM.

TABLE IV: Difference between the original Dex file and recovered Dex file from the samples of DB-15/DB-16 ($\oplus$, $\ominus$ and $\odot$ represent the recovered Dex file has additional code, less code, and the same code compared with the original Dex file, respectively).

| Packer | Qihoo | Ali | Bangcle | Tencent | Baidu | Ijiami |
|---|---|---|---|---|---|---|
| DB-15 | $\odot$ | $\oplus$ | $\oplus$ | $\oplus$ | $\oplus$ | $\odot$ |
| DB-16 | $\odot$ | $\odot$ | $\oplus$ | $\odot$ | $\oplus\ominus$ | $\oplus$ |

Second, we compare the difference between the original Dex files and the recovered Dex files. We randomly select 60 packed samples (10 samples packed by each packer), decompile the Dex files into Java codes, and then *manually* compare the decompiled Java codes from the original Dex files and those from the recovered Dex files. The comparison results are summarized in Table IV. The recovered Dex files are the same as the original Dex files for Qihoo packer, Ijiami packer of DB-15, Ali packer of DB-16 , and Tencent packer of DB-16.

Ali packer of DB-15 adds two classes to the original Dex files, each of which has one field and three empty methods. It also inserts the invocation of "*Exit.b(Exit.a())*" to the beginning of every Java method. *Exit.a()* just returns *false* and *Exit.b()* is empty. Tencent packer of DB-15 adds two classes to each packed sample while Ijiami packer of DB-16 inserts five classes.

For Bangcle packer, there are six additional classes and twelve additional classes added to the packed samples of DB-15 and DB-16 respectively. In the main activity class, a method named *com_sec_plugin_action_APP_STARTED()* is inserted and invoked at the beginning of the method smallemphonCreate(). Bangcle packer of DB-15 creates an intent named *com.secneo.plugin.action.APP_STARTED* and broadcasts it in *com_sec_plugin_action_APP_STARTED()*. Bangcle packer of DB-16 further creates a new monitoring thread in *com_sec_plugin_action_APP_STARTED()*. Baidu packer of DB-15 adds two classes to each packed sample, and re-implements all the *onCreate()* methods using the dynamic code modification technique. Baidu packer of DB-16 inserts one additional class to each packed app but replaces the implementation of *onCreate()* methods with native codes. Therefore, for these methods, the recovered Java code is less than the original Java code.

***Answer to Q2:*** *PackerGrind* can correctly recover all Dex code that are not removed by packers as well as the additional classes/methods inserted by packers. Even for the methods that are re-implemented in native code, *PackerGrind* can still recover useful sematic information, based on which it is possible to regenerate the Dex code.

### D. Comparison

While two recent tools, DexHunter and AppSpear, claimed to be general unpackers, only DexHunter's source code is available. Hence, we only compare *PackerGrind* , DexHunter

and Android-unpacker [21] using 30 randomly selected samples from six packers (i.e., 5 samples from each packer). We perform two-step checking on the correctness of recovered Dex files. First, the Dex files can be disassembled by Baksmali. Second, we compare them with the Dex files of the original apps. The failure in any step will lead to $\times$ in Table V indicating unsuccessfully unpacking.

TABLE V: Comparison among Android-unpacker, DexHunter and *PackerGrind*.

| Packer | Qihoo | Ali | Bangcle | Tencent | Baidu | Ijiami |
|---|---|---|---|---|---|---|
| Android-unpacker [21] | × | × | × | × | × | × |
| DexHunter [10] | × | √ | √ | √ | × | × |
| *PackerGrind* | √ | √ | √ | √ | √* | √ |

Android-unpacker recovers Dex files by attaching to the app process through *ptrace* and dumping Dex files in memory. However, it cannot attach to packed apps with anti-debugging capability. Moreover, for packers that dynamically release and modify Dex files, Android-unpacker cannot obtain the valid Dex files because it does not know the proper dumping moment.

*PackerGrind* can successfully recover all Dex files from six packers. We mark Baidu samples with * because the Dex code in *onCreate()* of those samples has been re-implemented in native code as detailed in Section IV-E. Although *PackerGrind* is not designed to reverse-engineer native code, it can still provide very useful information about the native method as explained in Section IV-E.

DexHunter cannot correctly recover the Dex files for samples from Qihoo, Baidu and Ijiami. For Qihoo samples, the Dex files dumped by DexHunter only contain stub classes instead of real code, such as *com.qihoo.util.Configuration* and *com.qihoo.util.StubApplication*, because Qihoo packer uses its own functions instead of runtime methods monitored by DexHunter to load classes. For Baidu samples, the Dex files dumped by DexHunter cannot be disassembled because their Dex headers have been modified by the packed apps. Hence, they cannot be recognized by de-compilers. DexHunter also cannot recover the original Dalvik bytecodes of *onCreate()*. For Ijiami samples, DexHunter cannot unpack them successfully due to the time-out checking mechanism utilized by Ijiami. More precisely, the packed apps will check the existence of a long-running task, which exceeds a time threshold, and exit if found. Therefore, the process of DexHunter will stop because its unpacking operations takes such a long time that the packed app exits quickly.

***Answer to Q3:*** *PackerGrind* outperforms other available unpacking tools (i.e., Android-unpacker and DexHunter).

### *E. Unpacking Malware*

We apply *PackerGrind* to 200 malware samples packed by eleven popular packers and successfully recover all Dex files. By performing static analysis on these Dex files, we find that malware often employed packers to hide the invocations of sensitive APIs requiring permissions. Given a Dex file, we scan all sensitive APIs in it, and count how many permissions are required according to the mapping between permissions and APIs from PScout [45]. It is worth noting that many detection systems leverage sensitive APIs and permissions to discover mobile malware [46]–[50].

Let $P_p$ and $P_r$ denote the number of permissions required by a packed app and its recovered Dex file, respectively. We calculate the means of $P_p$ and $P_r$ for all malware samples packed by a packer. The result listed in Table VI shows that from the recovered Dex files we find more evidences (i.e., sensitive APIs requiring certain permissions) to explain why a malware sample needs certain permissions. For example, for Naga samples, before unpacking, we cannot find any API invocation requiring the permissions in the manifest (i.e., $P_p = 0$). Malware detection system may think that these samples just overclaim the permissions without using them. In contrast, after unpacking, we can identify API invocations that require 3 permissions in the manifest on average (i.e., $P_r = 3$). Let $A_p$ and $A_r$ indicate the number of sensitive API invocations in a packed app and its recovered Dex file, respectively. We compute the means of $A_p$ and $A_r$ for all malware samples packed by a packer. The result listed in Table VI obviously indicates that many more sensitive APIs can be found from the recovered Dex file. For example, no sensitive API invocation is found in malware samples packed by Ijiami whereas on average 45.88 sensitive API invocations can be found from the recovered Dex files.

***Answer to Q4:*** *PackerGrind* can facilitate malware detection by exposing the hidden malicious components.

### *F. Overhead*

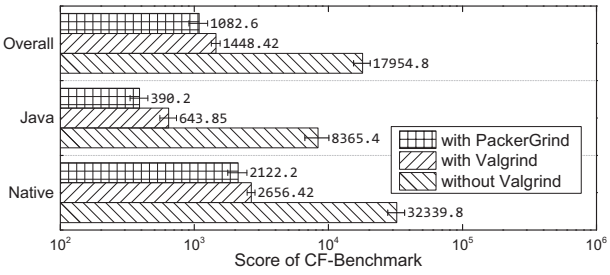


Fig. 8: CF-Benchmark results.

To evaluate the overhead introduced by *PackerGrind*, we run CF-Benchmark [51] 30 times on Nexus 5 without Valgrind, with Valgrind, and with *PackerGrind* , respectively. The scores of CF-Benchmark on the same smartphone without Valgrind serve as the baseline for comparison. Fig. 8 shows the results obtained in three scenarios, which include the overall scores, the scores of Java operations, the scores of native operations. We can see that Valgrind incurs 12.4 times slowdown and *PackerGrind* brings 17.6 times slowdown on average compared with the baseline. Since *PackerGrind* is based on Valgrind, it is still efficient because it is only 1.34 times slower than Valgrind. Compared with the dynamic analysis systems based on emulator that may introduce 11-34 times slowdown [15], *PackerGrind* has acceptable efficiency.

***Answer to Q5:*** *PackerGrind* introduces acceptable low overhead compared to Valgrind and emulator-based dynamic systems.

TABLE VI: Permissions and sensitive API calls in malware samples before and after unpacking by *PackerGrind*.

| Packer | Ali | Apkprotect | Baidu | Bangcle | Ijiami | Naga | Qihoo | Tencent | LIAPP | Netqin | Payegis |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Number of samples | 26 | 24 | 19 | 34 | 24 | 12 | 27 | 9 | 2 | 18 | 5 |
| Average value of $P_p$ | 0.00 | 2.65 | 0.21 | 0.24 | 0.00 | 0.00 | 2.07 | 7.22 | 0.00 | 4.28 | 1.80 |
| Average value of $P_r$ | 5.38 | 3.65 | 10.11 | 7.56 | 7.04 | 3.00 | 8.81 | 7.78 | 0.50 | 10.33 | 2.00 |
| Average value of $A_p$ | 0.00 | 5.62 | 0.95 | 0.88 | 0.00 | 0.00 | 6.30 | 40.67 | 0.00 | 19.56 | 5.80 |
| Average value of $A_r$ | 49.92 | 9.38 | 111.68 | 50.18 | 45.88 | 14.33 | 88.52 | 58.00 | 4.50 | 90.94 | 11.00 |

## VI. Discussion

*PackerGrind* can only recover the Dex data after the methods for releasing the real code are invoked. The majority of existing packers execute such methods, which are usually JNI methods, when a packed app is launched to avoid performance degradation. We also use IntelliDroid [25] to trigger the execution of such methods. Since packers may delay the execution of such methods after knowing the mechanism of *PackerGrind* we will leverage advanced input generator for Android [52], [53] to enhance *PackerGrind* in future work.

*PackerGrind* is based on Valgrind. Similar to the anti-emulator methods, packed apps may detect the existences of *PackerGrind* and then cease releasing the real code. For example, they could check the app starting command or count the time used to finish some operations. To address this issue, we could change the return value of selected APIs to hide the existence of *PackerGrind* or insert additional IR statements to modify the registers and force the app to execute forward.

*PackerGrind* currently focuses on the operations related to Dex files. Packed apps may hide the real code by modifying the compiled code in oat files directly. Moreover, if packed apps load different code into the same memory and execute them under different conditions, *PackerGrind* cannot decide which code is real. Since *PackerGrind* can trace such modifications and monitor code execution, we will address these issues by employing more semantic information in future work.

## VII. Related Work

Although there are already many studies on code packing/unpacking, almost all of them focus on x86 native codes [54]–[57]. The unpacking techniques for x86 binaries cannot be applied to packed Android apps because Android and the OSes running on x86 CPU have different architectures and execution models [10], [11], let alone the different formats of their executables. For example, Android packers need to protect both the Dex code and the native code if any, whereas traditional packers only hide native code [54]–[57].

Since mobile malware adopts packers to evade the detection, a few studies on unpacking apps were proposed recently from both academia [10], [11], [58] and industry [21], [39]. However, all of them adopt the one-pass strategy (i.e., dump the Dex data at fixed points), and therefore they can be easily evaded by the latest packers. For example, DWroidDump [58] only collects the Dex data in *dvmDexFileOpenFromFd()* when a Dex file is mapped to memory by the runtime. DexHunter [10] and AppSpear [11] are proposed to be general unpacker by customizing Android runtime. DexHunter inserts code in *defineClassNative()* to extract Dex files from memory. However, it may dump invalid Dex files since packers can release the real code after this function. AppSpear instruments the Dalvik interpreter to collect required data during method execution and then reconstruct the Dex file. Unfortunately, AppSpear may also dump invalid data because it relies on DVM's parsing methods to collect Dex data. Note that packers could make these methods return inaccurate results and use their own functions to parse Dex files. Moreover, AppSpear does not support ART. In contrast, PackerGrind adopts an iterative approach and conducts cross-layer monitoring so that it is adaptive to the changes of packers. Experimental results show that it outperforms existing approaches. Moreover, it supports both DVM and ART.

Existing cross-layer monitoring tools [15], [19], [59], [60] for Android cannot collect all necessary information and fulfill the requirement for handling packed apps. For example, ProfileDroid [59] cannot handle packed apps because it relies on apktool to conduct static analysis. TaintDroid [19] neither supports ART nor collects information at the runtime, system, and instruction layers. DroidScope [15] and NDroid [60] rely on QEMU, which can be detected by packers [61].

## VIII. Conclusion

The evolving app packers can easily circumvent existing unpackers because they adopt the one-pass strategy and hence are not adaptive to the changes of packers. To address this challenging issue, we propose a novel iterative process and develop *PackerGrind* to recover the Dex files from packed apps. With the capability of conducting cross-layer profiling in real smartphones, *PackerGrind* can effectively monitor the packing patterns and adapt to the evolution of packers for extracting Dex files. Our extensive experiments with real packed apps illustrate the effectiveness and efficiency of *PackerGrind*.

We will release *PackerGrind* to the community only for research purposes to prevent illegal use. Interested users please send an email to packergrind@gmail.com for the system using your university's email account. The first data set and the hash values of the samples in the second data set will be available at https://sites.google.com/site/packergrind.


## Acknowledgement

We thank the anonymous reviewers for their helpful comments. We appreciate the collaboration with mobile malware research team at Palo Alto Networks. This work is supported in part by the Hong Kong GRF (No. PolyU 5389/13E, 152279/16E), Hong Kong ITF (No. UIM/285), HKPolyU Research Grants (G-UA3X,G-YBJX), Shenzhen City Science and Technology R&D Fund (No. JCYJ20150630115257892), the US National Science Foundation (Grant No. CCF-1320605) and Office of Naval Research (Grant No. N00014-13-1-0175, N00014-16-1-2265, and N00014-16-1-2912).

## REFERENCES


[1] "Statistics and facts about Android," https://www.statista.com/topics/876/android/, 2016.

[2] G. Kelly, "97% of mobile malware is on Android. this is the easy way you stay safe," http://arcy24.blogspot.hk/2014/09/report-97-of-mobile-malware-is-on.html, 2014.

[3] CISCO, "Cisco 2014 annual security report," https://www.cisco.com/web/offers/lp/2014-annual-security-report/index.html, 2014.

[4] Y. Zhou and X. Jiang, "Dissecting Android malware: Characterization and evolution," in *Proceedings of the IEEE Symposium on Security and Privacy (IEEE S&P)*, 2012.

[5] K. Chen, P. Wang, Y. Lee, X. Wang, N. Zhang, H. Huang, W. Zou, and P. Liu, "Finding unknown malice in 10 seconds: Mass vetting for new threats at the Google-Play scale," in *Proceedings of the USENIX Security Symposium*, 2015.

[6] Y. Shao, X. Luo, C. Qian, P. Zhu, and L. Zhang, "Towards a scalable resource-driven approach for detecting repackaged android applications," in *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2014.

[7] M. Fan, J. Liu, X. Luo, K. Chen, T. Chen, Z. Tian, X. Zhang, Q. Zheng, and T. Liu, "Frequent subgraph based familial classification of android malware," in *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*, 2016.

[8] "OWASP mobile top 10 risks," https://www.owasp.org/index.php/OWASP_Mobile_Security_Project#tab=Top_10_Mobile_Risks, 2014.

[9] Gartner, Inc., "Debunking six myths of app wrapping," https://www.gartner.com/doc/3008117/debunking-myths-app-wrapping, 2015.

[10] Y. Zhang, X. Luo, and H. Yin, "DexHunter: toward extracting hidden code from packed Android applications," in *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*, 2015.

[11] W. Yang, Y. Zhang, J. Li, J. Shu, B. Li, W. Hu, and D. Gu, "AppSpear: Bytecode decrypting and DEX reassembling for packed Android malware," in *Proceedings of the International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2015.

[12] A. Apvrille and R. Nigam, "Obfuscation in Android malware, and how to fight back," *Virus Bulletin*, July 2014.

[13] S. Aimoto, "Five ways Android malware is becoming more resilient," https://www.symantec.com/connect/blogs/five-ways-android-malware-becoming-more-resilient, 2016.

[14] T. Vidas and N. Christin, "Evading Android runtime analysis via sandbox detection," in *Proceedings of the ACM Asia Conference on Computer and Communications Security (ASIACCS)*, 2014.

[15] L.-K. Yan and H. Yin, "DroidScope: Seamlessly reconstructing OS and Dalvik semantic views for dynamic Android malware analysis," in *Proceedings of the USENIX Security Symposium*, 2012.

[16] K. Tam, S. J. Khan, A. Fattori, and L. Cavallaro, "CopperDroid: Automatic reconstruction of Android malware behaviors." in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2015.

[17] M. Zheng, M. Sun, and J. C. Lui, "DroidTrace: a ptrace based Android dynamic analysis system with forward execution capability," in *Proceedings of the International Wireless Communications and Mobile Computing Conference (IWCMC)*, 2014.

[18] C. Yang, G. Yang, A. Gehani, V. Yegneswaran, D. Tariq, and G. Gu, "Using provenance patterns to vet sensitive behaviors in Android apps," in *Proceedings of the EAI International Conference on Security and Privacy in Communication Networks (SecureComm)*, 2015.

[19] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones," *ACM Transactions on Computer Systems*, vol. 32, no. 2, 2014.

[20] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2007.

[21] "Android-unpacker," https://github.com/strazzere/android-unpacker.

[22] "Dalvik executable format," https://source.android.com/devices/tech/dalvik/dex-format.html, 2016.

[23] C. Collberg and J. Nagra, *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Addison-Wesley, 2009.

[24] Baidu Inc., http://app.baidu.com.

[25] M. Y. Wong and D. Lie, "IntelliDroid: A targeted input generator for the dynamic analysis of Android malware," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2016.

[26] "F-Droid," https://f-droid.org/, 2015.

[27] Qihoo360 Inc., http://dev.360.cn/.

[28] Alibaba Inc., http://jaq.alibaba.com/.

[29] Bangcle Inc., http://www.bangcle.com/.

[30] Tencent Inc., https://www.qcloud.com/product/cr.

[31] Ijiami Inc., http://www.ijiami.cn/.

[32] Palo Alto Networks, "Wildfire[tm] cloud-based threat analysis service," https://www.paloaltonetworks.com/products/secure-the-network/subscriptions/wildfire.

[33] "Apk protect," https://sourceforge.net/projects/apkprotect.

[34] NAGA IN Inc., http://www.nagain.com/.

[35] T. Strazzere and J. Sawyer, "Android hacker protection level 0," *DEFCON*, 2014.

[36] Netqin Inc., https://www.netqin.com.

[37] PayEgis Inc., http://www.payegis.com/.

[38] "LG Nexus 5," http://www.gsmarena.com/lg_nexus_5-5705.php.

[39] "ZjDroid," https://github.com/halfkiss/ZjDroid, 2016.

[40] "Baksmali," https://github.com/JesusFreke/smali, 2016.

[41] "Dexdump," https://developer.android.com/studio/command-line/index.html, 2016.

[42] "Dex2jar," https://github.com/pxb1988/dex2jar, 2016.

[43] "Jadx," https://github.com/skylot/jadx, 2016.

[44] "IDA Pro," https://www.hex-rays.com/products/ida/, 2016.

[45] K. Au, Y. F. Zhou, Z. Huang, and D. Lie, "PScout: Analyzing the Android permission specification," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2012.

[46] M. Zhang, Y. Duan, H. Yin, and Z. Zhao, "Semantics-aware android malware classification using weighted contextual api dependency graphs," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2014.

[47] V. Avdiienko, K. Kuznetsov, A. Gorla, A. Zeller, S. Arzt, S. Rasthofer, and E. Bodden, "Mining apps for abnormal usage of sensitive data," in *Proceedings of the ACM/IEEE International Conference on Software Engineering (ICSE)*, 2015.

[48] L. Yu, X. Luo, X. Liu, and T. Zhang, "Can we trust the privacy policies of android apps?" in *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2016.

[49] M. Xu, C. Song, Y. Ji, M.-W. Shih, K. Lu, C. Zheng, R. Duan, Y. Jang, B. Lee, C. Qian, S. Lee, and T. Kim, "Toward engineering a secure android ecosystem: A survey of existing techniques," *ACM Computing Surveys (CSUR)*, vol. 49, no. 2, 2016.

[50] K. Tam, A. Feizollah, N. B. Anuar, R. Salleh, and L. Cavallaro, "The evolution of android malware and android analysis techniques," *ACM Computing Surveys (CSUR)*, vol. 49, no. 4, 2017.

[51] "CF-Bench," http://http://bench.chainfire.eu/, 2016.

[52] S. R. Choudhary, A. Gorla, and A. Orso, "Automated test input generation for android: Are we there yet?" in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2015.

[53] N. Mirzaei, J. Garcia, H. Bagheri, A. Sadeghi, and S. Malek, "Reducing combinatorics in gui testing of Android applications," in *Proceedings of the ACM/IEEE International Conference on Software Engineering (ICSE)*, 2016.

[54] F. Guo, P. Ferrie, and T.-C. Chiueh, "A study of the packer problem and its solutions," in *Proceedings of the International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2008.

[55] G. Bonfante, J. Fernandez, J.-Y. Marion, B. Rouxel, F. Sabatier, and A. Thierry, "CoDisasm: Medium scale concatic disassembly of self-modifying binaries with overlapping instructions," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2015.

[56] X. Ugarte-Pedrero, D. Balzarotti, I. Santos, and P. G. Bringas, "SoK: Deep packer inspection: A longitudinal study of the complexity of run-time packers," in *Proceedings of the IEEE Symposium on Security and Privacy (IEEE S&P)*, 2015.

[57] S. Schrittwieser, S. Katzenbeisser, J. Kinder, G. Merzdovnik, and E. Weippl, "Protecting software through obfuscation: Can it keep pace with progress in code analysis?" *ACM Computing Surveys (CSUR)*, 2016.

[58] D. Kim, J. Kwak, and J. Ryou, "DWroidDump: Executable code extraction from Android applications for malware analysis," *International Journal of Distributed Sensor Networks*, 2015.

[59] X. Wei, L. Gomez, I. Neamtiu, and M. Faloutsos, "Profiledroid: multi-layer profiling of android applications," in *Proceedings of the 18th annual international conference on Mobile computing and networking (Mobicom)*, 2012.

[60] C. Qian, X. Luo, Y. Shao, and A. T. Chan, "On tracking information flows through jni in android applications," in *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2014.

[61] Y. Jing, Z. Zhao, G.-J. Ahn, and H. Hu, "Morpheus: automatically generating heuristics to detect Android emulators," in *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2014.